Applicable and Sound Polyhedral Optimization of Low-Level Programs

Johannes Rudolf Doerfert

A dissertation submitted towards the degree

Doctor of Engineering (Dr.-Ing.)

of the

Faculty of Mathematics and Computer Science

of

Saarland University

Saarbrücken, 2018

UNIVERSITÄT
DES
SAARLANDES

Colloquium Details

Colloquium Date: December 19, 2018

Faculty Dean: Prof. Dr. Sebastian Hack

Committee Chairman: Prof. em. Dr. Dr. h.c. Reinhard Wilhelm

Examiners:

Prof. Dr. Sebastian Hack, Saarland University, Saarbrücken
Prof. Dr. Jan Reineke, Saarland University, Saarbrücken
Dr. Fabrice Rastello, Research Director, Inria, Grenoble

Academic Member: Dr. Roland Leißa

Zusammenfassung

Anwendbarkeit und Korrektheit Polyhedraler Optimierung Hardwarenaher Programme

von Johannes Rudolf Doerfert

Compiler Design Lab
Universität des Saarlandes

Aktuelle und zukünftige Computersysteme zeichnen sich durch verschiedenartige Mehrkernprozessoren, sowie programmierbare und spezialisierte Hardwarebeschleuniger aus. Zudem wird die Speicherhierarchie tiefer und oft durch Speicher mit niedriger Latenz, oder hoher Bandbreite, erweitert. Der Programmierer kann dieses enorme Potential allein nicht nutzen. Übersetzer müssen Einblick in das Programmverhalten geben, oder sogar Berechnungen und Daten selbst verwalten. Für beides brauchen sie eine ganzheitliche Sicht auf das Programm, da auch lokale Transformationen, die die Ausführungsreihenfolge, die Recheneinheit und das Speicherlayout unverändert lassen, nicht ausreichen um vielfältige Systeme auszulasten.

Das Polyedermodell, eine mathematische Programmdarstellung und ein Transformationsrahmenwerk, hat große Erfolge bei der Bewältigung verschiedener Probleme im Kontext vielfältiger Systeme erzielt. Obwohl die Analyse- und Transformationsfähigkeiten weithin anerkannt sind, wird auch allgemein angenommen, dass es zu restriktiv ist für Programme aus der Praxis.

In dieser Arbeit verbessern wir die Anwendbarkeit und Rentabilität von Techniken basierend auf dem Polyedermodell. Unsere Bemühungen garantieren eine korrekte Programmdarstellung und führen neue Anwendungen ein um die verfügbaren Informationen in der polyedrischen Programmdarstellung zu nutzen. Diese sind eigenständige Optimierungen und Techniken zur Ableitung von abstrakten Programmeigenschaften.

Diese Arbeit ist in englischer Sprache verfasst.

Abstract

Applicable and Sound Polyhedral Optimization of Low-Level Programs

by Johannes Rudolf Doerfert

Compiler Design Lab
Saarland University

Computers become increasingly complex. Current and future systems feature configurable hardware, multiple cores with different capabilities, as well as accelerators. In addition, the memory subsystem becomes diversified too. The cache hierarchy grows deeper, is augmented with scratchpads, low-latency memory, and high-bandwidth memory. The programmer alone cannot utilize this enormous potential. Compilers have to provide insight into the program behavior, or even arrange computations and data themselves. Either way, they need a more holistic view of the program. Local transformations, which treat the iteration order, computation unit, and data layout as fixed, will not be able to fully utilize a diverse system.

The *polyhedral model*, a high-level program representation and transformation framework, has shown great success tackling various problems in the context of diverse systems. While it is widely acknowledged for its analytical powers and transformation capabilities, it is also widely assumed to be too restrictive and fragile for real-world programs.

In this thesis we improve the applicability and profitability of polyhedral-model-based techniques. Our efforts guarantee a sound polyhedral representation and extend the applicability to a wider range of programs. In addition, we introduce new applications to utilize the information available in the polyhedral program representation, including standalone optimizations and techniques to derive high-level properties.

Acknowledgments

It has been over a year since I finished my thesis, at least all but this page. Consequently, I write these words with some figurative distance to the process and physical distance to the balcony on which most of the actual write up happened.

The first thing that comes to mind when I think back is being part of the Compiler Design Lab for many years. For me, it was both an honor and a joy to work as part of this group, or, put differently, an experience I am profoundly grateful for.

What eventually became this thesis was worked on over many years and in close collaboration with many people. The support that I experienced over these years, from my collaborators, from my colleagues in the research group and beyond, from friends inside and outside the university, as well as from all people I consider family, was crucial to overcome the hurdles I had to face. I want to use this opportunity to say *thank you* to everyone that supported me during this time.

I want to acknowledge my colloquium committee, and especially my reviewers, who did not only have to listen to my talk and read thoroughly through this thesis, but who also helped me to get everything done in a particularly short and specific time frame. I sincerely appreciate the effort put into the correction, the feedback at the end, and during the course of my dissertation.

Finally, I want to thank my advisor Sebastian Hack directly. Sebastian, you always gave me your time, support, and especially leniency throughout these years. Thank you. I doubt that without your help I would have found my research interests the same way and I most certainly would not have been able to explore them as I did. Beyond that, you always encouraged me when I looked into new areas. While many of them are not properly reflected in this thesis, they already turned out to be particularly advantageous for my professional life. I am happy we had many successful and interesting results, I am proud to tell people I worked with you and was advised by you, but honestly, I am most glad we had a great time.

Contents

Dedicated to my father whose never ending patience is priceless.

Chapter 1

Introduction & Motivation

> *"Begin at the beginning," the King said gravely,*
> *"and go on till you come to an end; then stop."*
>
> Lewis Carroll, *Alice in Wonderland*, 1899

Two of the main challenges for current and future computing systems [Luc+14] are energy efficiency [FM11] and performant computations on increasingly growing amounts of data [VOE11]. As a consequence, we need to take full advantage of the available hardware, especially because CPU frequencies stopped their decade long exponential growth [Sut05]. Even though hardware advances continue, the performance improvement per year for single-threaded floating-point SPEC benchmarks, shown in Figure 1.1, sunk 2004 from former 64% to 21% afterwards [Pre12].

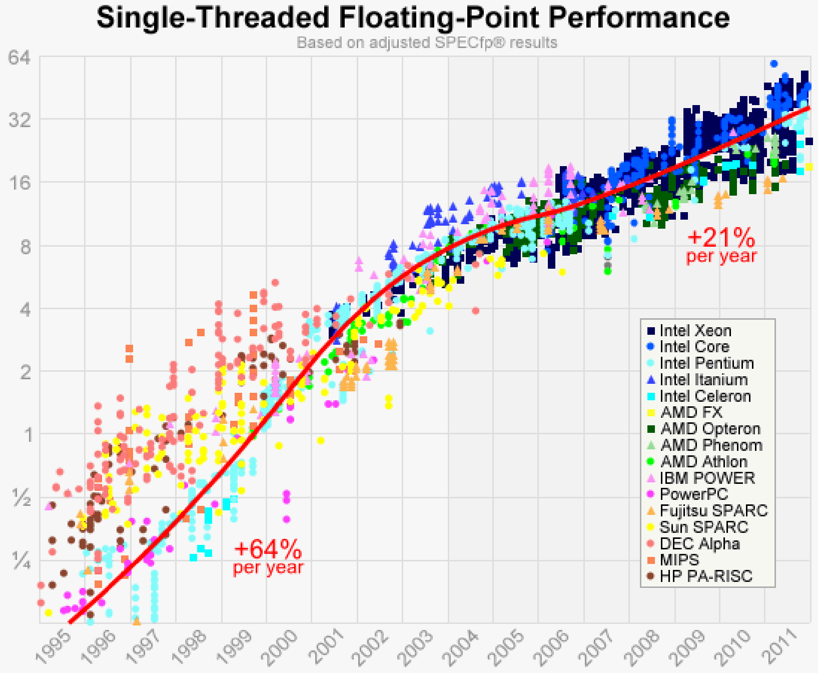


Figure 1.1: Floating point performance of single-threaded SPEC benchmark results reported between 1995 and 2011. Source: Preshing [Pre12].

While we need to maximise hardware utilization, a study by Prabhu et al. [Pra+11] revealed that in practise programs often "*run unaltered after an initial implementation on multiple machines with widely varying architectures*" [Pra+11]. This inevitable result of the cumulative complexity developers face, when confronted with heterogeneous machines [LUH18; Nou+17], highlights the importance of improved automatic solutions. Program performance prediction, a research field on its own, is a hard task for experts, and often infeasible for average programmers. Even after the right algorithm was selected, the implementation can easily underperform if the availability of a single hardware resource, from floating-point units to memory bandwidth, or the interplay between them, is limiting the overall throughput [Abe+13; Kol+13]. For a highly-tuned program, developers need to identify and remedy performance bottlenecks. While performance models and tooling support ease this burden, it is unrealistic to assume programmers could alone bridge the gap between actual and potential performance on a modern system. Instead, compilers have to increase their effort too. They have to augment local transformations with high-level schemes that automatically distribute and manage both computation and data [GH16; MF18].

To achieve high utilization on a complex and diverse system, compilers have to aide developers in various tasks. Parallelization, the distribution of computations across multiple processing elements, is only one of them. However, given the enormous and continuously increasing body of compiler research on this topic, see Streit [Str17, Chapter 3], as well as the quantity of language and library based solutions, it seems clear that solving this task is hard for humans [Pre12] and compilers alike[1]. Nevertheless, automatic, semi-automatic, and manual parallelization techniques allowed us to transition our programs from fast single-core processors, to systems with multiple, less-powerful CPUs, to the massively parallel accelerators which are nowadays available in commodity machines. While we could make a similar argument for schedule optimizations, data layout transformations, and other program modifications, we want to stress that the ever growing requirements on computational power, in high-performance but also general purpose computing, can only be satisfied through continuous improvement of our tools and techniques. To facilitate this development, we have to improve the existing state-of-the-art on multiple fronts, including productivity and applicability.

In this thesis we will take a closer look at the *Polyhedral Model* [FL11], a promising candidate to address the performance challenges that come with complex, heterogeneous, and massively parallel systems. As a high-level program representation and transformation framework, and with a powerful mathematical foundation, the polyhedral model allows for both structural and also fine-grained program transformations. Consequently, it is regularly used in automatic and semi-automatic approaches to improve the utilization of a system through distribution and reorganization of computation, data, or both.

[1] Though, in all fairness, expectations, architectures, and requirements are, and will always continue to be, moving targets for any kind of optimization. This means that even the best techniques for today's systems and important problems might be retired with the next programming trend or advance in hardware.

Based on decades of research with highly successful results, polyhedral-model-backed systems are on the rise to satisfy the increasing demand for fast computation pipelines in the context of deep learning and image processing [Bag+18; JB18; Vas+18]. Additionally, polyhedral-model-based techniques found their way into production compilers for general purpose languages such as IBM-XL [Bon+10], GCC (through GRAPHITE [Pop+06]), and LLVM [LA04] (through POLLY [GGL12]). While this certainly improved the outreach of such techniques, we regularly observe reservations when it comes to the application. While the reasons are manifold, there are several that are widely acknowledged:

- Several components of polyhedral-model-based techniques have a worst case doubly exponential complexity in various input characteristics (ref. [Bas04a, Section 4.3] and [Upa13]). While this theoretical upper bound is not necessarily reached, we have to ensure that the time investment is proportional to the benefit for the user. This is especially important for actively developed projects in which vast amounts of code are continuously compiled to ensure correctness after each modification.
- Polyhedral-model-based techniques and tools are often complex and hard to comprehend for non-expert users [Bag+16]. While ignorance can be bliss, a black-box optimization that cannot be understood, guided, and improved by the user is generally not. It is therefore crucial to improve feedback and guidance possibilities in a way that is custom to an average programmer [DGP15].
- Applicability, robustness, and correctness issues are another hurdle when it comes to polyhedral optimizations, especially for general purpose applications written or represented in low-level languages [DGH17]. Only if the code adheres to all syntactic and semantic restrictions of the polyhedral model [CGT04; Fea91], which might or might not be checked prior to the application, the desired transformations are performed and the result is correct. However, any subtle change to the program, even if it is semantic preserving, can cause us to miss out on optimizations for seemingly amenable code regions. Even worse, if implicit assumptions about the input do not hold, silent miscompilations can follow.

In the remainder of this thesis we will present extensions to classical polyhedral tooling which try to mitigate all of the above reasons that hold back the adaption of polyhedral-model-based techniques. We will mostly focus on the last one, thus *applicable*, *robust*, and *sound* application on programs in a low-level representation. Improvements in this area are important as they not only allow to apply polyhedral techniques to the enormous existing codebase written in low-level languages like C and C++, but also enable the same analyses and transformations for higher-level languages which are at some point lowered during their compilation. In addition to these extensions, the thesis also contains extensive evaluation results, usability and optimization improvements, as well as short introductions into our ongoing research.

1.1 Contributions & Structure

The main contributions of this thesis are techniques to improve polyhedral optimization of low-level programs. The enhancements we describe allow us to reliably create a sound and concise polyhedral representation of general purpose code written in low-level languages such as C/C++. In summary, we make the following contributions:

Measuring Applicability, Profitability, and Limitations

In Section 3.1, we discuss new metrics to summarize the applicability of polyhedral approaches on a large corpus of programs without the shortcomings of existing metrics. Additionally, we provide a profitability heuristic that reliably determines if a program region is actually amenable to polyhedral optimizations, prior to the costly polyhedral analyses. In Section 3.2, we determine and classify applicability limitations of a modern polyhedral optimizer through a large study on general purpose code.

Enhancing Applicability and Robustness

We describe several enhancements for polyhedral modeling of low-level programs in Sections 3.3 – 3.7. For each, we evaluate the applicability effect on a large selection of general purpose benchmarks (ref. Section 2.3). The proposed techniques eliminate syntactic and semantic limitations through the use of algorithms tailored to low-level inputs, and runtime checks synthesized from automatically derived preconditions. The result of our efforts is a significantly improved applicability on low-level programs.

Ensuring Correct and Concise Representations

In Sections 4.1 – 4.3, we present practical ways to bridge common semantic mismatches between the high-level polyhedral program repsentation and low-level languages such as C/C++, or the LLVM intermediate representation (LLVM-IR). The proposed techniques ensure a correct polyhedral representation of low-level programs through automatically synthesized runtime checks. Additionally, our modeling facilitates fast processing by favoring a concise representation of expected behaviors, over a complete representation of all possible program behaviors.

Improving Polyhedral Optimizations

Finally, we introduce new applications and optimizations for polyhedral model based tools on low-level programs. This chapter includes improvements with regards to user interaction in Section 5.1, techniques to determine high-level program properties from the low-level inputs in Section 5.2, as well as a standalone program optimization to improve the organization of computations in low-level programs in Section 5.3. The evaluation for the latter shows that we are on par, or even better, than dedicated state-of-the-art optimization approaches which are tailored towards the benchmarks we use.

The thesis is structured as follows: In Chapter 2, necessary background information is presented. This includes an introduction of the polyhedral model, some mathematical concepts, as well as details on the polyhedral representation and optimization of programs. Afterwards, we present the contributions of this thesis in Chapters 3 – 5. The first, Chapter 3, describes our work on applicability and profitability of polyhedral-model-based techniques. Beside metrics for both and an elaborate evaluation on current applicability limitations, we introduce and evaluate different extensions towards an applicable and robust polyhedral representation of low-level programs. In Chapter 4, the emphasis is put on correctness issues. While these are often similar to applicability problems, they can be more subtle. In this chapter we explain how a sound polyhedral program representation is possible without sacrifices due to unlikely corner cases in the semantics of the low-level input language. Afterwards, we present new applications and optimizations for polyhedral techniques in Chapter 5. This chapter includes standalone optimizations as well as techniques to deduce, and communicate, high-level program properties from low-level programs. Before we conclude this thesis in Chapter 7, we present ongoing work and interesting directions for future research Chapter 6.

1.2 Publications

This thesis is based on the following works presented at conferences and workshops. Each section additionally describes the implementation availability and relevant publications at its beginning.

SPolly: Speculative Optimizations in the Polyhedral Model [Doe+13]

Johannes Doerfert, Clemens Hammacher, Kevin Streit, and Sebastian Hack. International Workshop on Polyhedral Compilation Techniques (IMPACT), 2013.

The content of this publication overlaps with the bachelor thesis of Johannes Doerfert.

Polly's Polyhedral Scheduling in the Presence of Reductions [Doe+15]

Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. International Workshop on Polyhedral Compilation Techniques (IMPACT), 2015.

Runtime Pointer Disambiguation [Alv+15]

Péricles Alves, Fabian Gruber, *Johannes Doerfert*, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. International Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM, 2015.

Generalized Task Parallelism [Str+15]

Kevin Streit, *Johannes Doerfert*, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. ACM Transactions on Architecture and Code Optimization (TACO), 2015.

Assumption Tracking for Optimistic Optimizations [DGP15]

Johannes Doerfert, Tobias Grosser, and Sebastian Pop. LLVM-HPC Workshop, 2015.

Input Space Splitting for OpenCL [MDH16]

Simon Moll, *Johannes Doerfert*, and Sebastian Hack. International Conference on Compiler Construction (CC). ACM, 2016.

Polyhedral Driven Optimizations on Real Code [DH17a]

Johannes Doerfert and Sebastian Hack. LLVM Performance Workshop at CGO, 2017.

Optimistic Loop Optimization [DGH17]

Johannes Doerfert, Tobias Grosser, and Sebastian Hack. International Symposium on Code Generation and Optimization (CGO). IEEE/ACM, 2017.

Polyhedral Value & Memory Analysis [DH17b]

Johannes Doerfert and Sebastian Hack. LLVM Developers Meeting (LLVM Dev), 2017.

Polyhedral Expression Propagation [DSH18]

Johannes Doerfert, Shrey Sharma, and Sebastian Hack. International Conference on Compiler Construction (CC). ACM, 2018.

Chapter 2

Background

> *My definition of an expert in any field is a person who knows enough about what's really going on to be scared.*
>
> P.J. Plauger, *Computer Language*, 1983

This chapter provides the necessary background knowledge to comprehend the remainder of this work. Especially the *Polyhedral Model* is detailed in Section 2.1, including notation, underlying concepts, and an introduction into the polyhedral representation and optimization of programs. In addition, we provide an extensive glossary in Appendix B. It contains definitions of common terms from the field of compiler construction, (polyhedral) program optimization, as well as general terminology we use throughout this thesis. Whenever a term that is listed in the glossary is used for the first time in a section, it is highlighted by a dotted line, e.g., control flow graph (CFG). Note that the glossary only provides a brief description of potentially unclear terminology; we invite the interested reader to consult the provided literature for further information.

In addition to conceptual background, this chapter also contains details on the implementation and availability of the presented techniques in Section 2.2. The evaluation setup used throughout Chapter 3 and Chapter 4 is discussed in Section 2.3. The evaluation of the optimization techniques, presented in Section 5.2 and Section 5.3, is detailed in their respective sections.

Note that we start each section with an introduction of the problem which will generally include dedicated background information, examples, and additional references to the relevant literature.

2.1 The Polyhedral Model

Throughout this thesis we consider the *Polyhedral Model* [FL11] mostly as a mathematical program abstraction and optimization framework based on Presburger Arithmetic [Pre31][1]. As such it has been used for decades to analyze and optimize programs with static affine control flow conditions and access relations [Fea92a; Fea92b; Len93; Pug91a]. Programs, or better program parts, that fulfill these and other requirement imposed by the polyhedral model (ref. Section 2.1.3 and Section 3.2) are known as static control parts, or SCoPs [CGT04; Fea91]. The polyhedral representation of a SCoP is oblivious to the actual computation performed by the program, but instead concerned with the execution order of statement instances. A statement instance is a pair containing a statement and an iteration vector that identifies iterations of the loops surrounding the statement. The extent of a statement in the polyhedral representation depends on the statement granularity. Common choices for "higher-level" programs, written in Fortran and C-like languages, are source level statements [BRS10; Bon+08; Fea92b; GL96; VG12]. Low-level polyhedral tools that work on the control flow graph (CFG) of a program generally use basic blocks as statements [GGL12; Pop+06]. There are however alternatives which aim to improve applicability [MDH16], optimization freedom [Sto+14], or to reduce compile time costs, which generally grows exponentially [Upa13] with the number of the statements [MY15]. The set of instances for which a statement $\mathtt{S}$ is executed, is defined by its iteration domain $\mathcal{D}_\mathtt{S}$ in terms of surrounding loop iterations and parameter values. The order in which statement instances are executed is determined by the schedule θ of the SCoP. Both are derived from the input code as explained in Section 2.1.3 and Section 3.3. Polyhedral statements contain a representation of all (scalar and memory) accesses that were also contained in the program part represented by the statement. A memory access m is defined by the accessed array (or pointer) and the piecewise, quasi-affine access relation f_m. This relation maps an iteration vector $\mathbf{i} \in \mathcal{D}_\mathtt{S}$ of the statement $\mathtt{S}$ to the array element(s) accessed by m in the loop iteration described by $\mathbf{i}$. Note that the access relations, the iteration domains, and the schedule may contain references to symbolic, thus unknown but fixed, parameters of the SCoP. These parameters are commonly function arguments or scalar variables defined outside of the SCoP. In Section 3.6 we introduce an extension that also allows global variables to be considered parameters.

The remainder of this section briefly introduces background literature to the polyhedral model in Section 2.1.1, before Presburger Arithmetic, the mathematical foundation on which modern implementations of the polyhedral model are rooted, is described in Section 2.1.2. Afterwards, Section 2.1.3 explains the polyhedral representation of programs in more detail. Readers with a background in polyhedral optimization may safely skip these parts and consult the glossary in Appendix B in case terms are unknown or unclear.

[1] An annotated English translation of the originally German paper was made available by Stansifer [Sta84].

2.1.1 Polyhedral Model Literature

With roots dating back more than five decades [CH78; KMW67], and improved by equally seminal contributions since then [AI91; Fea91; Fea92b; GL96], it is safe to say that the polyhedral model has successfully stood the test of time. The copious body of research on polyhedral-model-based techniques includes everything from schedule optimizations [Bon+10; DI15; Fea92a; Fea92b], which change the order of program statement instances, over data-layout transformations [BBC16; DSV05; DIY16; QR00; YR13], to techniques that distribute both computation and data across processing units [Dam+15; GH16; MF18; Mik+14; Vas+12; Ver+13]. There is also an active research community that is continuously improving polyhedral-model-based techniques and adapting them to new applications. Notable active research directions include speculative approaches [Caa+17; Doe+13; Jim+13a; Suk+14; SC16], applicability improvements [Bag+13; DGH17; Gro+15; KG18; ZKC18], as well as the application to new problems, programming models, and architectures [Cla+11; DH17b; GH16; Ham+18; Kur17; MF18; MDH16; Pra+17; SHS17; SKF18; ZHB18].

2.1.2 Presburger Predicates, Sets, and Relations

Presburger arithmetic is a decidable first-order logic over integers with addition [Pre31]. In the context of this work, we express Presburger formulae through piecewise defined, quasi-affine expressions, as well as as logical combinations of equalities and inequalities over such expressions [Ver16, Chapter 4]. A piecewise quasi-affine expression is a set of affine expressions where each one is defined over a disjunct part of the input space. The input space spans all possible valuations of the variables in the expressions. Formula 1 defines the general shape of an affine expression. The variables $\mathbf{x} = (x_1, \dots, x_n)$ are universally quantified, the variables $\mathbf{p} = (p_1, \dots, p_m)$ are existentially quantified, and $\mathbf{c} = (c_0, \dots, c_{n+m}) \in \mathbb{Z}^{n+m}$ are constant integers. As shown here, we use bold letters to denote vectors and italic letters to indicate variables.

$$\begin{aligned} \exists\, \mathbf{p} : \forall \mathbf{x} : \mathbf{c}^\intercal \times (1, \mathbf{x}, \mathbf{p}) &= \mathbf{c}^\intercal \times (1, x_1, \dots, x_n, p_1, \dots, p_m) \\ &= c_0 + \sum_{1 \leq i \leq n} (c_i * x_i) + \sum_{1 \leq i \leq m} (c_{i+n} * p_i) \end{aligned} \tag{1}$$

Quasi-affine expressions additionally allow integer divisions and modulo operations with a constant divisor. Note that both can be reduced to additional existentially quantified variables and constraints. This reduction is exemplarily shown in the translation from Formula 4 to Formula 5. The structure of affine, quasi-affine, and piecewise quasi-affine expressions is provided in Figure 2.1. To improve readability, we generally use the term affine expression, even if we deal with piecewise defined, quasi-affine expression. The distinction is made explicit only if necessary.

$$\begin{array}{lll}
\langle \mathit{aexp} \rangle ::= & \langle \mathit{cnst} \rangle \mid \langle \mathit{ident} \rangle \mid (\langle \mathit{aexp} \rangle) \mid \langle \mathit{aexp} \rangle\ (\,+ \mid -\,)\ \langle \mathit{aexp} \rangle & \text{affine expression} \\
\quad\mid & \langle \mathit{cnst} \rangle * \langle \mathit{aexp} \rangle \mid \langle \mathit{aexp} \rangle * \langle \mathit{cnst} \rangle & \\
\langle \mathit{qaexp} \rangle ::= & \langle \mathit{aexp} \rangle \mid \langle \mathit{qaexp} \rangle\ (\,/ \mid \mathrm{mod}\,)\ \langle \mathit{cnst} \rangle & \text{quasi-affine expr.} \\
\langle \mathit{pqaexp} \rangle ::= & \langle \mathit{qaexp} \rangle \mid \langle \mathit{pqacond} \rangle\ \mathbf{?}\ \langle \mathit{pqaexp} \rangle\ \mathbf{:}\ \langle \mathit{pqaexp} \rangle & \text{pw. quasi-affine expr.} \\
\langle \mathit{pqacond} \rangle ::= & \langle \mathit{pqacond} \rangle\ (\,\wedge \mid \vee\,)\ \langle \mathit{pqacond} \rangle & \text{pw. quasi-affine condition} \\
\quad\mid & \langle \mathit{pqaexp} \rangle\ (\,< \mid > \mid = \mid \neq \mid \leq \mid \geq\,)\ \langle \mathit{pqaexp} \rangle &
\end{array}$$

Figure 2.1: Grammar for affine, quasi-affine, and piecewise quasi-affine expressions.

For integer sets and relations, we adopt the notation of the integer set library ISL [Ver10]. Though, we omit the definition of existentially quantified variables. All unbound variables are implicitly existentially quantified. As an example, the variable N is existentially quantified in the predicate $\forall i : 0 \leq i \leq N$ since it is not otherwise bound.

A Presburger set, from here on mostly denoted as *integer set*, is a subset of the n-dimensional integer space $\mathbb{Z}^n$ that is defined by a Presburger condition. A value $\mathbf{x} \in \mathbb{Z}^n$ is part of the set, if and only if there exists an assignment for the unbound, and thereby existentially quantified variables in the condition, that extends $\mathbf{x}$ to a fulfilling assignment of the predicate. An example for a three-dimensional integer set is given in Formula 2. Note that in this notation universal quantification is assumed implicitly for all variables left of the bar |. Thus, the three variables i, j, and k are universally quantified while the two unbound variables N and M are existentially quantified. Formula 3 shows the explicit representation of the same set.

$$\{\,(i,j,k) \mid \qquad 0 \leq i < N \wedge 0 \leq j < M \wedge i < k < j\,\} \subseteq \mathbb{Z}^3 \tag{2}$$

$$\{\,(i,j,k) \mid \exists\, N, M : 0 \leq i < N \wedge 0 \leq j < M \wedge i < k < j\,\} \subseteq \mathbb{Z}^3 \tag{3}$$

An *integer relation* maps n dimensional integer points to m dimensional integer points and it is also defined by a Presburger condition. While we generally interpret them as mappings, thus with type $\mathbb{Z}^n \to \mathbb{Z}^m$, we will equivalently treat them as integer sets over the combined space $\mathbb{Z}^{n+m}$. As an example consider the relation shown in Formula 4 which translates two-dimensional points into a three-dimensional space. Similar to ISL, we denote the two-dimensions on the left as *input dimensions* and the three on the right as *output dimensions*. Also note the simplified notation to express the equivalence of the first input and the second output dimension. The only other constraint for this relation states that the sum of j and k modulo two is equal to m. The last output dimension is consequently bounded while all other dimensions are not. The relation is shown as an equivalent integer set, with explicit quantification and a "desugared" modulo operation, in Formula 5.

$$\{\,(i,j) \to (k,i,m) \mid (j+k) \bmod 2 = m \,\} \subseteq \mathbb{Z}^2 \to \mathbb{Z}^3 \tag{4}$$

$$\{\,(i,j,k,l,m) \mid \exists\, e : l = i \wedge j + k = 2e + m \wedge 0 \leq m \leq 1\,\} \subseteq \mathbb{Z}^5 \tag{5}$$

The ISL library provides algorithms to effectively handle integer sets and relations. For a detailed introduction to the underlying concepts, as well as the capabilities of ISL, we recommend the tutorial by Verdoolaege [Ver16]. In this work we mostly use common mathematical concepts from set and relation theory explained in the following.

Union and Intersection

For two integer sets or relations of equal dimensionality we construct the *union* or *intersection* by building the conjunction, or, respectively, disjunction of the defining Presburger predicates. If we take the general integer relations $r_1 \coloneqq \{\, \mathbf{i} \to \mathbf{j} \mid c_1(\mathbf{i},\mathbf{j}) \,\}$ and $r_2 \coloneqq \{\, \mathbf{i} \to \mathbf{j} \mid c_2(\mathbf{i},\mathbf{j}) \,\}$, both subsets of $\mathbb{Z}^n \to \mathbb{Z}^m$, we define the union and intersection as shown in Formulae 6 and 7.

$$r_1 \cup r_2 \coloneqq \{\, \mathbf{i} \to \mathbf{j} \mid c_1(\mathbf{i},\mathbf{j}) \vee c_2(\mathbf{i},\mathbf{j}) \,\} \tag{6}$$

$$r_1 \cap r_2 \coloneqq \{\, \mathbf{i} \to \mathbf{j} \mid c_1(\mathbf{i},\mathbf{j}) \wedge c_2(\mathbf{i},\mathbf{j}) \,\} \tag{7}$$

Complement and Projection

For integer sets and relations we define the *complement* and *projection* as shown in Formula 8 and 9. Given an integer set $s \subseteq \mathbb{Z}^n$ of n-dimensional vectors, the complement set $\neg s$ contains all elements of the same dimensionality that were not contained in s. The projection eliminates dimensions of s such that elements of the result can always be extended to elements of the original set. Except for Algorithm A.1 on Page 201, we always project onto the parameter space ρ. This operation, denoted as $\pi_\rho(\circ)$, will eliminate *all* dimensions and the result will only constrain the existentially quantified variables that were constrained in the original set. The definitions for integer relations, which can also be interpreted as sets, are similar to the ones shown.

$$\neg s \coloneqq \{\, \mathbf{i} \in \mathbb{Z}^n \mid \mathbf{i} \notin s \,\} \subseteq \mathbb{Z}^n \tag{8}$$

$$\pi_m(s) \coloneqq \{\, \mathbf{i} \in \mathbb{Z}^m \mid \exists \mathbf{j} \in \mathbb{Z}^{n-m} : (i_1, \ldots, i_m, j_1, \ldots, j_{n-m}) \in s \,\} \subseteq \mathbb{Z}^m \text{ with } m < n \tag{9}$$

Domain, Range, Inversion, Application, and Composition

The common definitions for the *domain* and *range* of an integer relation $r \subseteq \mathbb{Z}^n \to \mathbb{Z}^m$ are provided in Formula 10 and 11. The *inversion* r^{-1} of r is defined as the reverse mapping shown in Formula 12. The *application* of the relation r onto a set $s \subseteq \mathbb{Z}^n$ is defined in Formula 13. In the result, the elements of s are translated according to the mapping provided by r. Finally, two compatible relations can be *composed*, through application of one relation to the domain or range of the other. This is denoted as $\circ_{dom}$ or $\circ_{rng}$ and defined in Formula 14 and 15.

$$dom(r) := \{ \mathbf{i} \in \mathbb{Z}^n \mid \exists \mathbf{j} \in \mathbb{Z}^m : (\mathbf{i},\mathbf{j}) \in r \} \tag{10}$$

$$rng(r) := \{ \mathbf{j} \in \mathbb{Z}^m \mid \exists \mathbf{i} \in \mathbb{Z}^n : (\mathbf{i},\mathbf{j}) \in r \} \tag{11}$$

$$r^{-1} := \{ \mathbf{j} \rightarrow \mathbf{i} \mid (\mathbf{i},\mathbf{j}) \in r \} \subseteq \mathbb{Z}^m \rightarrow \mathbb{Z}^n \tag{12}$$

$$r(s) := \{ \mathbf{j} \in \mathbb{Z}^m \mid \mathbf{i} \in s \subseteq \mathbb{Z}^n : (\mathbf{i},\mathbf{j}) \in r \} \tag{13}$$

$$r \circ_{dom} f := \{ \mathbf{k} \rightarrow \mathbf{j} \mid (\mathbf{i},\mathbf{j}) \in r \wedge (\mathbf{i},\mathbf{k}) \in f \} \subseteq \mathbb{Z}^p \rightarrow \mathbb{Z}^m \qquad f \subseteq \mathbb{Z}^n \rightarrow \mathbb{Z}^p \tag{14}$$

$$r \circ_{rng} f := \{ \mathbf{i} \rightarrow \mathbf{k} \mid (\mathbf{i},\mathbf{j}) \in r \wedge (\mathbf{j},\mathbf{k}) \in f \} \subseteq \mathbb{Z}^n \rightarrow \mathbb{Z}^p \qquad f \subseteq \mathbb{Z}^m \rightarrow \mathbb{Z}^p \tag{15}$$

2.1.3 Polyhedral Representation And Optimization of Programs

Polyhedral-model-based approaches commonly use the term static control part (SCoP) to refer to a (maximal) program part that can be represented in the polyhedral model [CGT04; Fea91]. We additionally use it to denote the polyhedral representation that is consequently constructed for that program part. Note that common restrictions of SCoPs are discussed in Section 3.2. A SCoP conceptually comprises a list of statements as well as a schedule relation θ. The statements in a SCoP are named and contain a description of the accesses as well as an iteration domain. This domain describes the surrounding loop iterations for which the statement is executed in terms of iteration vectors. The statement (name) together with an iteration vector is also called a statement instance. The iteration domain $\mathcal{D}_{\mathtt{S}}$ of a statement S describes all dynamically executed instances with regard to the iterations of surrounding loops. Formula 16 and 17 show the (simplified) iteration domains constructed for the statements P and Q defined in Figure 2.2b.

$$\mathcal{D}_{\mathtt{P}} = \{ (n,m) \mid 0 \leq n < N \wedge 0 \leq m < M \} \tag{16}$$

$$\begin{aligned} \mathcal{D}_{\mathtt{Q}} &= \{ (n,m) \mid 0 \leq n < N \wedge 0 \leq m < M \wedge n = m \} \\ &= \{ (n,n) \mid 0 \leq n < \min(N,M) \} \end{aligned} \tag{17}$$

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    for (k = i + 1; k < j; k++)
      S(i, j, k);
```

(a) Three dimensional loop nest with a single "abstract" statement S. The iteration domain $\mathcal{D}_{\mathtt{S}}$ of S is shown in Formula 2 on Page 10. Abstract statements are used whenever the actual accesses and computations are not important but only the iteration domains and schedules are.

```
    for (n = 0; n < N; n++)
      for (m = 0; m < M; m++) {
P:      A[n][m] = B[n] + C[m];
        if (n == m)
Q:        A[n][m] *= -1;
      }
```

(b) Two dimensional loop nest with two "concrete" statements P and Q. We use examples with concrete statements if the access relations or the computations are important.

Figure 2.2: The two code styles used for example programs provided in this thesis.

The schedule θ is an integer relation which maps iteration vectors of statement instances to multidimensional timestamps. At runtime, the statement instances are executed according to the lexicographic order of the timestamps associated to them by the schedule. Thus, for two statements $\mathtt{S}$ and $\mathtt{P}$, with respective iteration vectors $\mathbf{i} \in \mathcal{D}_{\mathtt{S}}$ and $\mathbf{j} \in \mathcal{D}_{\mathtt{P}}$, the constraint $\theta\left(\mathtt{S}(\mathbf{i})\right) \ll_{lex} \theta\left(\mathtt{P}(\mathbf{j})\right)$ implies that $\mathtt{S}(\mathbf{i})$ is executed (or scheduled) before $\mathtt{P}(\mathbf{j})$. To further improve readability, we use the names of the loop iteration variables surrounding the statements to denote the dimensions of the iteration domain as well as the (input) dimensions of the schedule. Given the definition of $\mathbf{i}$ and $\mathbf{j}$ above, we would omit the names of the statements and simply write $\theta\left(\mathbf{i}\right) \ll_{lex} \theta\left(\mathbf{j}\right)$ to express the same lexicographic order between the statement instances. Our examples are chosen such that this naming scheme will usually suffice to uniquely identify statement instances. If not, we explicitly name all statements and use their respective names as well. To distinguish consecutive statement instances by their multidimensional timestamps, additional constant dimensions are introduced [KP95]. Due to these additional dimensions, a general schedule for a SCoP with d nested loops requires up to $2d + 1$ output dimensions [CGT04]. However, detailed knowledge about the generation of schedules will not be required in the remainder of this thesis. Depending on the use case, we either show only the schedule part relevant to a single statement, or the entire schedule with embedded statement names. To express the original execution order for the statements in Figure 2.2b, we could therefore write $\theta_{\mathtt{P}} = \left\{ (i, j) \rightarrow (i, j, 0) \right\}$ and $\theta_{\mathtt{Q}} = \left\{ (i, j) \rightarrow (i, i, 1) \right\}$, or alternatively $\theta = \left\{ \mathtt{P}(i, j) \rightarrow (i, j, 0)\, ;\, \mathtt{Q}(i, j) \rightarrow (i, j, 1) \right\}$.

An access m is identified by the accessed array, pointer, or scalar variable and the access relation f_m. Since scalars are accessed without an offset, we often omit the access relation. For memory accesses, the array (or pointer) needs to be statically known and invariant in the SCoP. If that is not the case, or the access relation contains dynamic values, e.g., function calls or memory loads, the memory access is considered dynamic, or simply non-affine. While the polyhedral model generally only allows static affine accesses, we discuss extensions for dynamic and non-affine accesses in Section 3.4 and 3.6. Multidimensional accesses are also (partially) supported if the dimension sizes can be statically determined [Gro+15]. Finally, we categorize accesses into write and read as well as may or must accesses. Since the later distinction is a consequence of approximations, it is further discussed in Section 3.4.

The statement descriptions, and the initial schedule, are derived from the input code. As this thesis describes work in the context of the LLVM/POLLY optimizer, our implementations generally assume statements to be basic blocks. However, the techniques are not conceptually restricted to this choice. To simplify the presentation of our examples, we generally use a C/C++ -like syntax, potentially with explicitly labeled statements. Depending on the context, we use one of two naming styles depicted in Figure 2.2. The first one (part 2.2a) is used when the memory accesses and computations are irrelevant while the second (part 2.2b) makes both explicit.

Polyhedral optimization is classically concerned with the computation of a new schedule relation for a given SCoP [Bon+08; Fea92a; Fea92b]. To this end, the polyhedral representation is first used to derive dependences Γ between statement instances [Fea91]. We generally distinguish three kinds of dependences[2]: *read-after-write* (*RAW*), *write-after-read* (*WAR*), and *write-after-write* (*WAW*). The schedule optimizer, or scheduler, then uses the dependences Γ to compute a new schedule θ', which defines a different execution order of the statement instances. The new schedule θ' is *valid* if the order it defines adheres to the dependence relation Γ. Thus, dependent statement instances $(\mathbf{i},\mathbf{j}) \in \Gamma$ have to keep their original order: $\theta'(\mathbf{i}) \ll_{lex} \theta'(\mathbf{j})$. Finally, the code structure, e.g., the abstract syntax tree (AST) or the control flow graph (CFG), of the underlying program is changed according to the new schedule θ' [AI91; Bas04a; GVC15; QRW00]. Depending on the objective function and the optimization algorithm, the new schedule might minimize the reuse distance between dependent accesses [Fea92a; Fea92b], expose loop parallelism and tileable dimensions [Bon+08], or improve vectorization [Kon+13; MDH16]. However, only the techniques described in Section 5.2 and 5.3 influence, or are influenced by, the actual objective function and optimization algorithm. Hence, the majority of presented techniques is oblivious and can be used with any existing or emerging polyhedral optimization scheme.

In addition to dependences between statement instances, LLVM/POLLY can also compute dependences between instances of accesses. While generally more expensive to compute, these fine-grained relations are beneficial if the statements are not considered atomic entities. To denote a dependence between two specific accesses a_1 and a_2, we write $a_1 \to a_2$. Thus, $w{\to}r$ describes the read-after-write (*RAW*) dependence between the read r and the write w. If we are interested in all dependences that are emanating from an access, we substitute the target by an asterisk, e.g., $r{\to}*$ denotes all write-after-read (*WAR*) dependences originating from the read r.

The interested reader can find a graphical illustration of iteration domains and dependences in Figure 5.8 on Page 126. We additionally provide a short introduction to the polyhedral optimizer LLVM/POLLY in Section 2.2.2. It features a schematic overview of the optimization pipeline in Figure 2.3. Readers that look for a more in-depth introduction to the polyhedral model may consult the Ph.D theses of Bastoul [Bas04b, Part I] and Pouchet [Pou10, Chapter 2].

2.2 Implementation Notes

In this thesis we describe several enhancements over the state-of-the-art in polyhedral program analysis and optimization. All techniques are publicly available as part of LLVM's polyhedral optimizer POLLY or in our research prototype. For each enhancement we also discuss the availability and associated literature in a dedicated box on the first page of the respective section.

[2] Depending on the field, *RAW* dependences are also referred to as *flow* dependences, similarly *WAR* dependences are known as *anti* dependences, and *WAW* dependences are also called *output* dependences.

2.2.1 The LLVM Compiler Framework

The LLVM compiler framework [LA04] is an open source, industry-strength compiler written in modern C++. Like most compilers, it consists of front-ends that parse the input languages, a middle-end that applies (mostly target independent) optimizations, and back-ends that emit the target dependent machine code. There are several benefits of LLVM that made it very popular in the research community, including a modular implementation, various front-ends and back-ends, a single intermediate language (LLVM-IR), and the availability of dozens of analysis passes.

2.2.2 The Polyhedral Optimizer Polly

Polly is the polyhedral loop optimizer of the LLVM compiler framework [GGL12]. It was initially a port of Graphite [Pop+06], the polyhedral loop optimizer in the GCC compiler suite. However, over the last couple of years Polly evolved separately [Doe+15; DGH17; Gro+15; GH16; KG18; Rag11]. The general structure of LLVM/Polly is shown in Figure 2.3. The entry point for LLVM-IR functions is the SCoP detection. It identifies (maximal) valid single-entry single-exit (SESE) regions for which Polly can build a polyhedral representation (ref. Section 3.2). We denote both, the region and the polyhedral representation thereof, as static control part (SCoP). During SCoP modeling, the code in the valid SESE regions is analysed more thoroughly and the polyhedral representation, here denoted as "SCoP object", is built. From this polyhedral representation, the dependence analysis [Fea91] will derive read-after-write (*RAW*), write-after-read (*WAR*), and write-after-write (*WAW*) dependences between statement instances. Afterwards, the schedule optimizer will compute a new schedule θ' that adheres to all dependences, but is optimal according to some objective function [Fea92a]. LLVM/Polly uses the integer set library isl [Ver10] to represent, handle, and modify integer sets and relations. isl is also used to compute dependences, derive the optimized schedule, and to generate the optimized AST. The optimization algorithm employed by isl is similar to the one described by Bondhugula et al. [Bon+08]. The polyhedral code generation implemented in isl follows the approach by Grosser, Verdoolaege, and Cohen [GVC15]. Finally, a code generation step will generate LLVM-IR based on the optimized AST structure that was computed from the optimized schedule. The LLVM-IR will also contain alias and dependence annotations that are usable by later passes, e.g., the LLVM loop vectorizer. If requested, Polly will additionally try to parallelize loops using the OpenMP [Cla98] runtime library [Rag11]. We refer the interested reader to the Diploma Thesis of Grosser [Gro11] which contains a slightly outdated, but detailed introduction of LLVM/Polly as well as some relevant parts of LLVM and LLVM-IR.

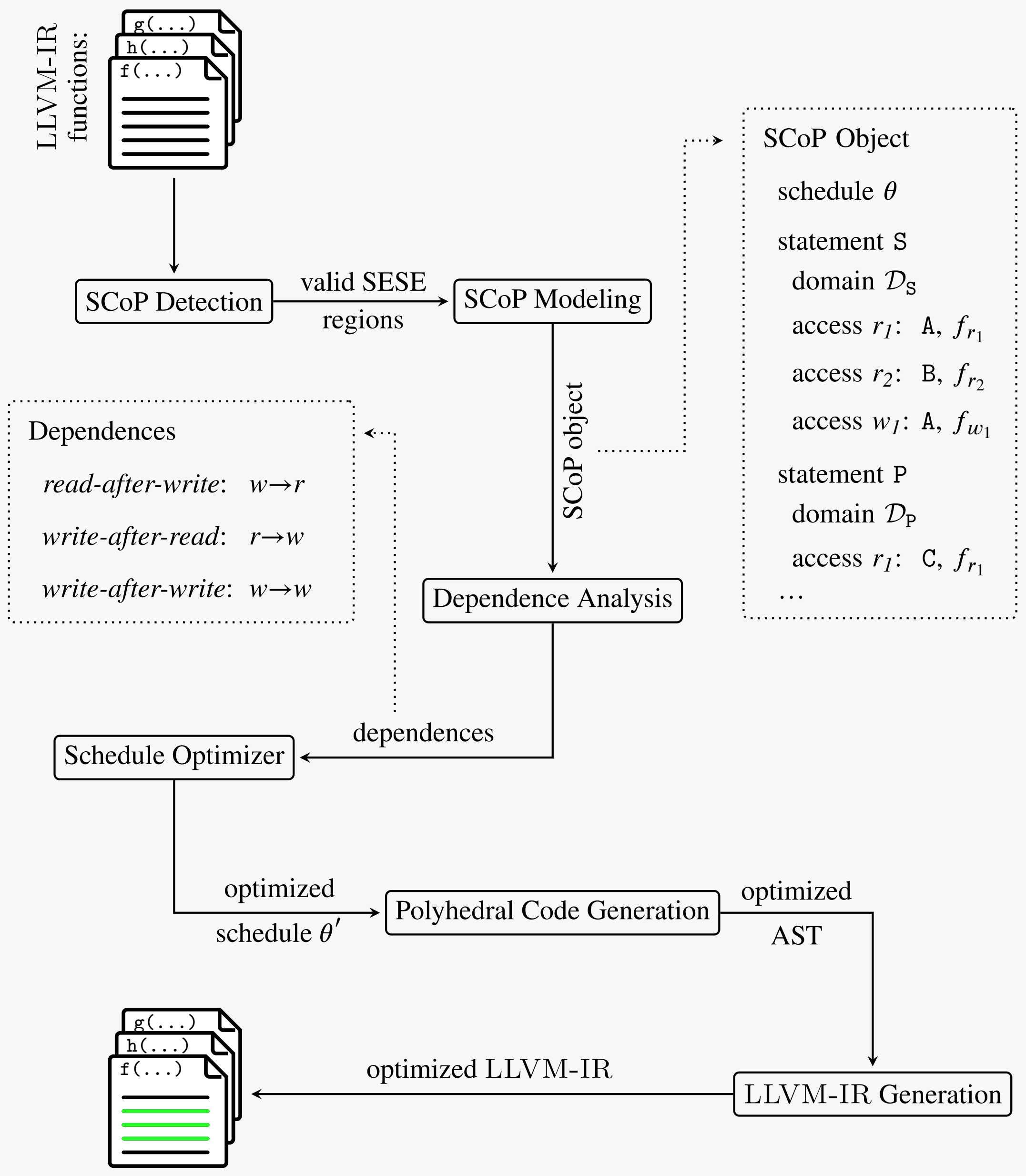


Figure 2.3: Schematic overview of LLVM/POLLY's optimization pipeline. Note that we only depict the major passes; others like dead iteration elimination, are left out.

At the beginning of this work, LLVM/POLLY was only able to recognize affine expressions ($\langle$*aexp*$\rangle$ in Figure 2.1), as well as minimum and maximum computations which are special piecewise affine computations. Also conditionals were limited to equalities and inequalities over affine expressions. We added support for quasi-affine expressions ($\langle$*qaexp*$\rangle$ in Figure 2.1), conjunctions $\wedge$, and disjunctions $\vee$. Our *Polyhedral Value Analysis* (ref. Section 6.4) additionally supports general piecewise defined expressions $\langle$*pqaexp*$\rangle$ and it is worth to note that all described techniques are conceptually capable of dealing with these as well.

The domain generation we integrated into LLVM/POLLY is described in Section 3.3. It generates precise iteration domains for any reducible [HU74] control flow graph (CFG) with static affine control flow conditions. For dynamic or non-affine conditions, we introduced control flow approximations (ref. Section 3.4.2). While this eliminates common syntax restrictions, e.g., the use of `goto`, LLVM/POLLY is still limited by the detection of affine expressions. To identify affine expressions for access relations and control flow conditions, it relies on the SCALAR EVOLUTION analysis [BWZ94; PCS05]. With our *Polyhedral Value Analysis* [DH17b] we proposed a polyhedral-model-based alternative to SCALAR EVOLUTION that is discussed in Section 6.4.

2.2.3 Our Polly Research Prototype

While most techniques presented in this thesis are integrated into the open source version of LLVM/POLLY, some experimental extensions and prototype implementations are only available in our research prototype. These features are in separate branches of our research prototype and publicly available online: `https://github.com/jdoerfert/polly`. The branches are mostly based on the 6.0 release version of LLVM/POLLY and listed in Table 2.4, together with a reference to the section(s) in which they are mentioned.

Name	Section	Name	Section
`metrics`	Section 3.1	`min_dep_dis`	Section 5.1.1
`rejection_reasons`	Section 3.2	`reductions`	Section 5.2
`approximation`	Section 3.4	`expression_propagation`	Section 5.3
`bitops`	Section 3.2.2	`interprocedural`	Section 6.1 – 6.3

Table 2.4: Feature branches of our research prototype.

2.3 Evaluation Notes

For the evaluation of the presented techniques we used the LNT tool as a driver for the LLVM Test Suite (LLVM-TS)[3] as well as different versions of the SPEC benchmark suites[3]. With this setup we generated the statistics used to evaluate all techniques presented in Chapter 3 and Chapter 4, as well as Section 5.1.3. As these statistics are deterministic, we only performed a single run. The benchmarks that were executed in this configuration are listed in Table 2.5 and 2.6. The POLLY version we executed was based on the 6.0 release. Modifications were only applied to collect statistics or to evaluate the respective enhancement. However, to improve our baseline we enabled invariant load hoisting, as described in Section 3.6, for all runs. In case the evaluation setup differed, e.g., for the optimization techniques in Section 5.2 and 5.3, as well as the compile time evaluation presented in Section 3.1.6, it is detailed in the respective section.

[3] LLVM-TS git version: f9c975a1, SPEC2000/2006/2017 with respective versions: 1.3.1/1.1/1.0.1

SPEC2000

164.gzip	175.vpr	176.gcc	177.mesa	179.art	181.mcf
183.equake	186.crafty	188.ammp	197.parser	252.eon	253.perlbmk
254.gap	256.bzip2	300.twolf			

SPEC2006

400.perlbench	401.bzip2	403.gcc	429.mcf	433.milc
444.namd	445.gobmk	447.dealII	450.soplex	453.povray
456.hmmer	458.sjeng	462.libquantum	464.h264ref	470.lbm
471.omnetpp	473.astar[a]	482.sphinx3	483.xalancbmk	

SPEC2017

500.perlbench_r	502.gcc_r	505.mcf_r	508.namd_r
510.parest_r[a]	511.povray_r	519.lbm_r	520.omnetpp_r
523.xalancbmk_r	525.x264_r	526.blender_r	531.deepsjeng_r
538.imagick_r	541.leela_r	544.nab_r	557.xz_r
600.perlbench_s	602.gcc_s	605.mcf_s	619.lbm_s
620.omnetpp_s	623.xalancbmk_s	625.x264_s	631.deepsjeng_s
638.imagick_s	641.leela_s	644.nab_s	657.xz_s

LLVM Test Suite — Polybench

correlation	covariance	atax	bicg
cholesky	doitgen	gemver	gesummv
mvt	symm	syr2k	syrk
trisolv	trmm	durbin	dynprog
gramschmidt	lu	floyd-warshall	reg_detect
adi	fdtd-2d	fdtd-apml	jacobi-1d-imper
jacobi-2d-imper	seidel-2d		

Table 2.5: Tested C/C++ benchmarks in the SPEC [b] benchmark suites as well as the Polybench v3.2 benchmarks[c] that are part of the LLVM Test Suite (LLVM-TS)[d].

[a] The compilation of `Library.cpp` (473.astar) and `parameter_handler.cc` (510.parest_r) produced an error that prevented compile time statistics for this file in the evaluations shown in Chapter 3, Chapter 4, and Section 5.1.3.

[b] From the SPEC suites we only used the benchmarks written in C/C++ but not the ones written in Fortran. There are various technical reasons for this choice but also the believe that the C/C++ programs provide better insights into general purpose programming. The Table does consequently not list Fortran code.

[c] Online available at: `https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`

[d] Online available at: `https://github.com/llvm-mirror/test-suite`

LLVM Test Suite — TSVC

ControlFlow-dbl	ControlFlow-flt	ControlLoops-dbl
ControlLoops-flt	CrossingThresholds-dbl	CrossingThresholds-flt
Equivalencing-dbl	Equivalencing-flt	Expansion-dbl
Expansion-flt	GlobalDataFlow-dbl	GlobalDataFlow-flt
InductionVariable-dbl	InductionVariable-flt	LinearDependence-dbl
LinearDependence-flt	LoopRestructuring-dbl	LoopRestructuring-flt
Symbolics-dbl	Symbolics-flt	

LLVM Test Suite — Benchmarks

Halide/bilateral_grid	Halide/blur	Halide/local_laplacian
ALAC/decode	ALAC/encode	ClamAV
JM/ldecod	JM/lencod	SIBsim4
SPASS	d	kimwitu++
lua	obsequi	oggenc
sgefa	sqlite3	viterbi
7zip	ASCI_Purple/SMG2000	Bullet
ASC_Sequoia/AMGmk	ASC_Sequoia/CrystalMk	ASC_Sequoia/IRSmk
ProxyApps/CLAMR	ProxyApps/PENNANT	ProxyApps/RSBench
ProxyApps/SimpleMOC	ProxyApps/miniAMR	ProxyApps/miniGMG
FreeBench/analyzer	FreeBench/neural	FreeBench/pifft
MallocBench/espresso	MallocBench/gs	SciMark2-C
McCat/04-bisect	McCat/05-eks	McCat/18-imp
MiBench/automotive-susan	MiBench/consumer-jpeg	MiBench/consumer-lame
MiBench/consumer-typeset	MiBench/security-sha	MiBench/telecomm-gsm
NPB-serial/is	PAQ8p	Prolangs-C/agrep
Prolangs-C/gnugo	Ptrdist/bc	Ptrdist/yacr2
VersaBench/beamformer	VersaBench/bmm	mafft
mediabench/gsm/toast	mediabench/jpeg-6a	mediabench/mpeg2dec
nbench	sim	tramp3d-v4
Adobe-C++	BenchmarkGame	CoyoteBench
Linpack	Misc	Misc-C++
Shootout	Shootout-C++	Stanford

Table 2.6: C/C++ benchmarks in the LLVM Test Suite (LLVM-TS)[a] when executed with the `lnt --benchmarking-only` option. The Polybench benchmarks that are part of the LLVM Test Suite are shown in Table 2.5.

[a] Online available at: `https://github.com/llvm-mirror/test-suite`

Chapter 3

Applicability & Profitability

> “ C *is quirky, flawed, and an enormous success.* ”
>
> Dennis M. Ritchie, *The Development of the C Programming Language*, 1996

Polyhedral program optimization techniques have been around for decades [Fea92b; Len93]. However, only recently we began to apply these techniques automatically on (a large corpus of) real-world programs [Bas+03; Bon+10; Doe+13; DGH17; Sim+13]. With the development of GRAPHITE [Pop+06] in GCC and POLLY [GGL12] in LLVM, the two foremost open source C/C++ compilers are nowadays equipped with polyhedral optimization capabilities. In addition, the LLVM compiler framework has become a very popular back-end for emerging languages such as Julia, Swift or Rust. Furthermore, existing languages, including but not limited to Ada, Fortran, Java and Haskell, can be lowered to the LLVM intermediate representation (LLVM-IR). While this eliminates the need for a dedicated polyhedral optimizer per language, it creates new challenges due to the input diversity that stems from various programming styles and different code structures.

Even though GCC and LLVM make it easier to enable polyhedral optimization during the compilation of existing and emerging software projects, the lacking applicability as well as robustness of available tools severely limits the actually observed results. This is especially problematic as it encourages programmers to manually optimize their loop-heavy code in order to get an instant, yet short term, benefit. Later on, optimized code increases the maintenance burden, the adaption cost for new hardware, as well as the analysis efforts required to understand it again.

In this Chapter we will focus on applicability and profitability of polyhedral-model-based tools. In Section 3.1 we define both terms in detail, discuss the differences and the usefulness in the context of automatic polyhedral optimization. We describe the limitations of polyhedral techniques, as well as the effect of the here presented advances, in Section 3.2. The remaining Sections 3.3 – 3.7, will introduce several extensions over classical polyhedral modeling techniques that are designed to improve applicability, precision, and robustness for general purpose applications.

3.1 Applicability & Profitability Metrics

To evaluate new techniques, and compare them to existing ones, metrics are defined and then evaluated in a fixed environment. For compiler optimizations the most important evaluation criteria are arguably: performance, code size, and other resource requirement of the compilation. However, new advances will not always be able to deliver improvements in these metrics, especially not for all benchmarks. While their contribution is visible to certain metrics, others might only be affected after several more pieces have been put in place. Consequently, it is not always beneficial to look only at the immediate impact on e.g., performance. Instead, we might want to consider other criteria that are impacted now and potentially improve other metrics later on.

	No.	Metric	Description
Static / Compile Time	(1)	# SCoPs	The number of SCoPs in the program. [Bas+03; Doe+13; DGH17; GH16]
	(2)	# depth-n SCoPs	The number of SCoPs containing loops of a certain depth. [Bas+03; Ben+10; Gir+06; GH16]
	(3)	# n-statement SCoPs	The number of SCoPs containing a certain number of statements. [Bas+03; Gir+06]
Dynamic / Runtime	(4)	# executed SCoPs	The number of SCoPs that were executed. [DGH17]
	(5)	# SCoP executions	The number of times a SCoP was executed. [DGH17]
	(6)	SCoP coverage	The time spend in SCoPs. [Gir+06; Sim+13]

Table 3.1: Static and dynamic applicability metrics for polyhedral-model-based approaches. Note that the definition of SCoPs varies between approaches. Some might not require statically affine accesses while others do only count “rich” or “profitable” SCoPs which contain a minimum number of statically affine loops or statements enclosed in such.

Polyhedral optimizations are generally assessed like other compiler techniques, thus by their impact on performance [AB15; DSH18; Pan+15; Ven+14] or resource requirements [BBC16; DIY16; Zuo+13]. Though, extensions to polyhedral techniques might not directly impact these metrics but target, and consequently measure, *applicability* instead [Bas+03; Ben+10; Doe+13; DGH17; GH16; Sim+13]. However, there is no de-facto standard for measuring applicability. Different approaches use different metrics, as listed and described in Table 3.1. Due to the use of

In this section we discuss existing applicability and profitability metrics for polyhedral-model-based tools. In addition we present new ones that avoid problems we encountered evaluating several own extensions to polyhedral applicability [Alv+15; Doe+13; DGH17; DSH18]. The new metrics are available in the `metrics` branch of our research prototype.

various metrics it is hard to compare different applicability enhancing techniques. To make things harder, there is no single definition for basic underlying concepts. A static control part (SCoP) might for example be allowed to contain unknown or non-affine memory accesses [Bas+03]. SCoPs might also be required to contain a minimal number of instructions [GH16], at least one affine loop [Bas+03], or multiple transformable statements or loops [DGH17].

The task to define a single *applicability* metric is complex due to the amount of characteristics to choose from. All static characteristics can be extracted from either the input program or the optimized version, both with regards to the actual source code or the polyhedral representation thereof. In addition, there are dynamic characteristics, e.g., the time spent in SCoPs, that can be used. While most choices have their merit, it is important to consider the problems that come with them: Statistics gathered from the source code are sensitive to the programming language, syntactic constructs as well as canonicalizations and transformations applied prior to the polyhedral modeling. Similarly, the polyhedral representation vastly differs depending on the statement granularity [MY15], e.g., C statements [Bon+08] vs. basic blocks in a CFG [GGL12]. Furthermore, there are other representation choices that heavily impact the structure of the polyhedral representation, e.g., dynamic single assignment form (DSA) [Fea88a] or an alternative elimination of scalar variables [DSH18; KG18]. The optimized program is sensitive to the employed transformations and code generation schemes. Scheduling choices, e.g., emphasis on loop fusion vs. fission, loop tiling, as well as complementary optimizations (ref. Section 5.3) have a vast impact on characteristics of the resulting code, e.g., the number of loops and accesses. Finally, dynamic statistics do heavily depend on the chosen input data set.

Profitability metrics, in contrast to applicability metrics, determine the optimization potential of a polyhedral program representation. With regards to polyhedral performance optimization, *profitable* SCoPs [DGH17] are code regions that can be optimized in a beneficial way by polyhedral techniques. While the optimization potential differs depending on the optimization scheme, e.g., schedule optimizations [Bon+08; Bon+10; Fea92b], parallelization [Doe+15; Jim+13b; PKC12], accelerator offloading [GH16; SHS17; Sud+14; Zuo+13] or memory requirement minimization [BBC16; DIY16; QR00], there are always more and less promising candidate regions. Especially for the automatic application of polyhedral scheduling techniques, profitability metrics play a crucial role. If they are expensive or to lenient, compile time is wasted and performance is likely to suffer as well. If they are to restrictive, optimization opportunities are missed. To this end, profitability metrics play an important role in identifying candidates that are likely to be optimized given the capabilities and optimization goals of the polyhedral tool.

Profitability metrics are important to predict the impact of automatic polyhedral optimization. However, this does not render applicability metrics obsolete, especially not if polyhedral techniques do not have to justify program transformations (on their own). If they are used to support other optimization schemes [Atz+16; Jun15; Str+15], or are augmented by non-polyhedral information, e.g., domain [MDH16] or runtime knowledge [BA13; CWC16; Doe+13], applicability improvements can have a positive impact which is missed by (common) profitability metrics.

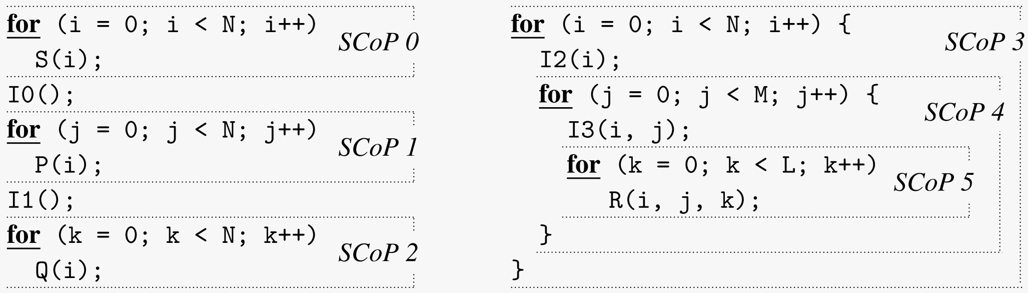
3.1.1 Monotone Applicability Metrics

Applicability metrics should be monotone in the sense that an increased applicability will result in a higher metric score. Though, none of the metrics listed in Table 3.1 does fulfill this requirement.

To illustrate the problems we provide two examples in Figure 3.2. The first, in part 3.2a, features three SCoPs, each enclosing one loop (nest)[1]. If we now increase applicability, we could end up with a single SCoP that covers all three loops. Assuming we did not give up precision, e.g., no approximations were used, a single SCoP is arguably better as it allows for more transformations, e.g., loop fusion and expression propagation (ref. Section 5.3). However, metrics (1) - (5) will yield higher, thus better, scores for three smaller SCoPs and the part of the runtime spend in SCoPs, metric (6), would remain almost the same. In the second example, part 3.2b, three potential SCoPs are indicated. Using metric (5) will result in more executions if fewer loops are covered. Metric (6) will again not vary much for all three choices since each contains the innermost loop. Though, transformations like interchange or tiling are not possible for *SCoP 5* and other optimizations, e.g., parallelization or offloading, might be more profitable for *SCoP 3*.

Existing applicability metrics are not monotone since they treat SCoPs as atomic entities, metrics (1), (4) - (6), or consider only the maximum value of a property, metrics (2) and (3). In the first case, changes to properties like the number of affine loops are missed and SCoP expansion (ref. Figure 3.2a) can consequently harm the score. In the second case, only changes that alter the maximum value of a property are accounted for. A similar situation arises when we simply sum up the number of affine loops or statements enclosed in SCoPs. The problems illustrated by the examples in Figure 3.2 will then still exist, assuming we first remove the intermediate statements `I0` to `I3`. The number of statements enclosed in affine loops does not change for the code shown in Figure 3.2b and there is no increase in the number of affine loops if a single SCoP would cover all three loops in Figure 3.2a. Alternatively, a metric could use a mean to summarize a certain property, e.g., the number of statically affine loops, with regards to the number of SCoPs. However, these metrics will decrease for newly detected SCoPs that are below the current average.

[1] For simplicity only one-dimensional loops with single statement loop bodies are shown. However, with regards to metrics (2) and (3), the same situation can appear for higher-dimensional loops that contain multiple statements.



```
for (i = 0; i < N; i++)
  S(i);
I0();
for (j = 0; j < N; j++)
  P(i);
I1();
for (k = 0; k < N; k++)
  Q(i);
```

(a) Multiple SCoPs that cover one loop each but in total all three loops.

```
for (i = 0; i < N; i++) {
  I2(i);
  for (j = 0; j < M; j++) {
    I3(i, j);
    for (k = 0; k < L; k++)
      R(i, j, k);
  }
}
```

(b) Multiple SCoPs that all include the innermost statement `R` and up to 3 surrounding loops.

Figure 3.2: Two fabricated examples that show the problems with both static and dynamic SCoP applicability metrics (ref. Table 3.1). The static loop coverage for the left example is 100% since all loops are covered in SCoPs[1]. The dynamic coverage for the SCoPs in the right example is similar if metrics (4) and (6) are used. For metric (5) it is even better to detect a smaller, inner SCoP than a larger one that contains more loops and consequently allows more transformations.

A good applicability metric should reward SCoP expansion, avoid regression due to new SCoPs, and additionally take the actual optimization potential into account. To this end, we propose two symbolic polyhedral applicability metrics, a static one C_α and a dynamic one R_β.

The piecewise affine metric C_α is defined in Formula 1. It determines a score for a given set of SCoPs based on the number of statically affine loops that are contained in each individual one.

$$\mathsf{C}_\alpha : 2^{\mathrm{SCoP}} \longrightarrow \mathbb{N}$$
$$\mathsf{C}_\alpha(S) = \sum_{s\in S} \max\left(0,\ \#\mathrm{loops}(s) - \alpha\right) \qquad \text{with } \alpha > 0 \tag{1}$$

The maximum guarantees that no SCoP can regress the overall result and that SCoP expansion will never decrease the score. If a single SCoP s subsumes[2] a set of SCoPs S we know that $\mathsf{C}_\alpha(\{s\}) \geq \mathsf{C}_\alpha(S)$. The parameter $\alpha > 0$ accounts for the optimization potential of a single statically affine loop. Given a SCoP with n loops, each will contribute $1 - \frac{\alpha}{n}$ to the final score. If $\alpha = 0$, the score would be similar to metric (1) in Table 3.1, except that SCoPs are weighted by the number of contained loops. For $\alpha = 1$, the metric will count how many loops can be combined with one fixed loop in each SCoP. For $\alpha > 1$, only SCoPs of a certain size will improve the score. This is similar to metrics (2) and (3) shown in Table 3.1, but it also accounts for transformation opportunities within one SCoP.

The dynamic metric R_β is defined in Formula 2. Similar to metrics (4) to (6) in Table 3.1, it measures the potential runtime improvement of polyhedral optimization. Such improvements are limited by many factors, including the number of statically affine loops. The parameter β therefore allows to adjust the score, which looks at the loop count, based on empirical data. As

[2] This implicitly assumes that no approximations (ref. Section 3.4) were used to increase the SCoP size.

an example, β should be set to a reasonable SCoP size that might allow for a twofold speedup. The scaling factor for loops of this size is then $\frac{1}{2}$, and the overall metric result is the sum of the runtime fraction of each SCoP that optimization could reasonably eliminate.

$$\begin{aligned} \mathsf{R}_\beta &: 2^{\text{SCoP}} \longrightarrow \mathbb{Q} \\ \mathsf{R}_\beta(S) &= \sum_{s \in S} \frac{\#\text{loops}(s)}{\#\text{loops}(s) + \beta} * \text{runtime}(s) \qquad \text{with } \beta > 0 \end{aligned} \tag{2}$$

Note that for both metrics it is plausible, and potentially even beneficial, to replace, or augment, the number of statically affine loops with the number of represented statements, affine memory accesses, or other static properties.

3.1.2 Applicability Evaluation

We evaluated the different static applicability metrics listed in Table 3.1 as well as the monotone one proposed above. The results for the first two metrics ((1) and (2)) are shown in Figure 3.3.

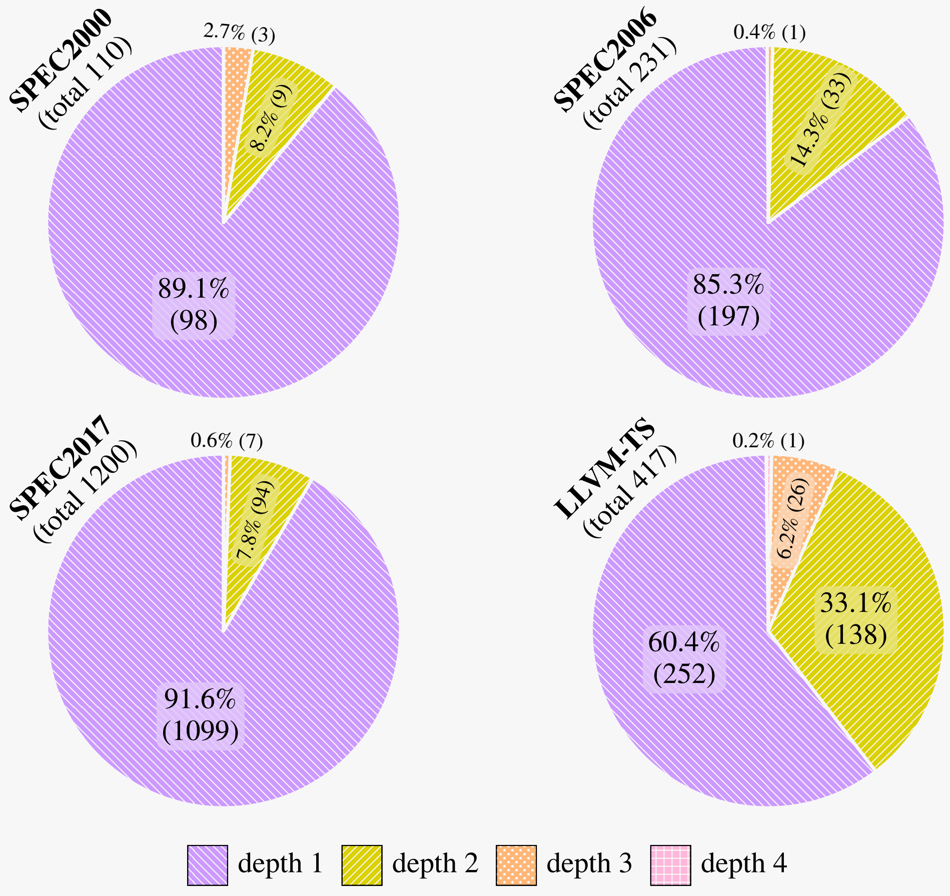


Figure 3.3: Number of feasible[3] SCoPs (upper left) and their respective maximal loop depth. The benchmark suites as well as the evaluation setup is described in Section 2.3.

The total number of feasible[3] SCoPs per benchmark suite is indicated below the name. For SPEC2000, there are 110 feasible SCoPs while SPEC2017 contains 1200. In Table 3.4 we put these absolute numbers into perspective and also show our proposed scores (ref. Formula 1) for two values of α. While the number of single-entry single-exit (SESE) regions, which are analyzed and optimized by LLVM/Polly or GCC/Graphite, varies a lot between the benchmark suites, for each $\approx 0.2\%$ of them are valid SCoPs. In contrast to valid regions, SCoPs are maximal [CGT04]. Thus, expanding the SCoP is either impossible or would cause invalidation.

Benchmarks	**# regions**	**valid regions**	**# SCoPs**[3]	C_0	C_1
SPEC2000	47312	662 (1.40%)	110 (0.23%)	125	15
SPEC2006	126124	1269 (1.01%)	231 (0.18%)	266	35
SPEC2017	584884	5401 (0.92%)	1200 (0.21%)	1308	108
LLVM-TS	163974	29203 (17.81%)	417 (0.25%)	610	193

Table 3.4: The number of valid SESE regions and SCoPs compared to the total number of SESE regions in the different benchmark suites. The last two columns show the static score according to Formula 1 with $\alpha = 0$ and 1.

Figure 3.6 shows the number of SCoPs with regards to the number of contained statements. To evaluate metric (3), namely the number of $n-$statement SCoPs, all values right of the chosen size n have to be summed up. The small number of single statement SCoPs can be explained by the profitability heuristics employed by Polly (ref. Section 3.1.4). Namely, SCoPs that contain only a single statement and a single loop are discarded (reason SSL in Table 3.19 on Page 44).

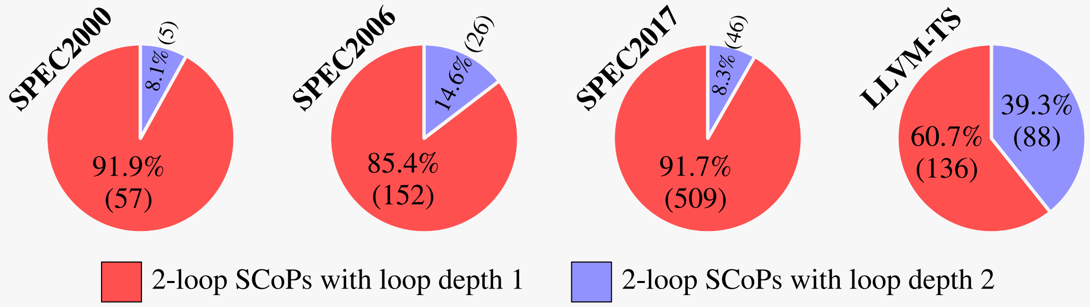


Figure 3.5: Loop depth distribution for SCoPs with exactly two loops.

Similar to the number of statements, we visualized the number of SCoPs for a specific loop count in Figure 3.7. While the small number of single loop SCoP was already explained above, it is worth to note that the loop distribution is concentrated around SCoPs with two loops. As illustrated by Figure 3.5, most of the 2-loop SCoPs, especially in the SPEC benchmarks, have only loop depth one. Thus, the two loops are most often not nested but in sequence. This does restrict the set of applicable loop transformations as listed in Table 3.8.

[3] Only valid *and* statically feasible SCoPs were counted. Figure 3.43 on Page 76 shows feasibility of all valid SCoPs.

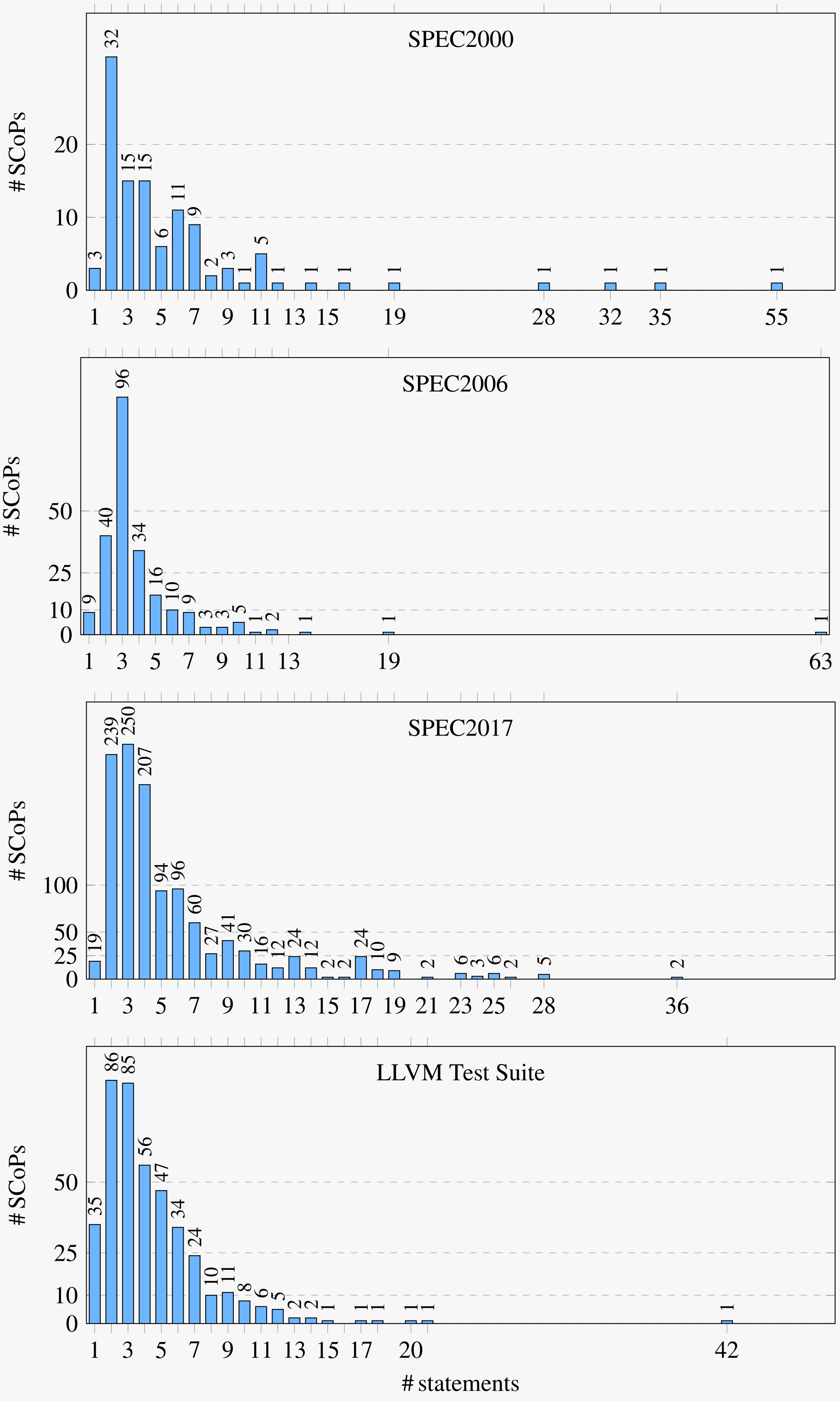


Figure 3.6: Number of SCoPs with regards to the number of contained statements.

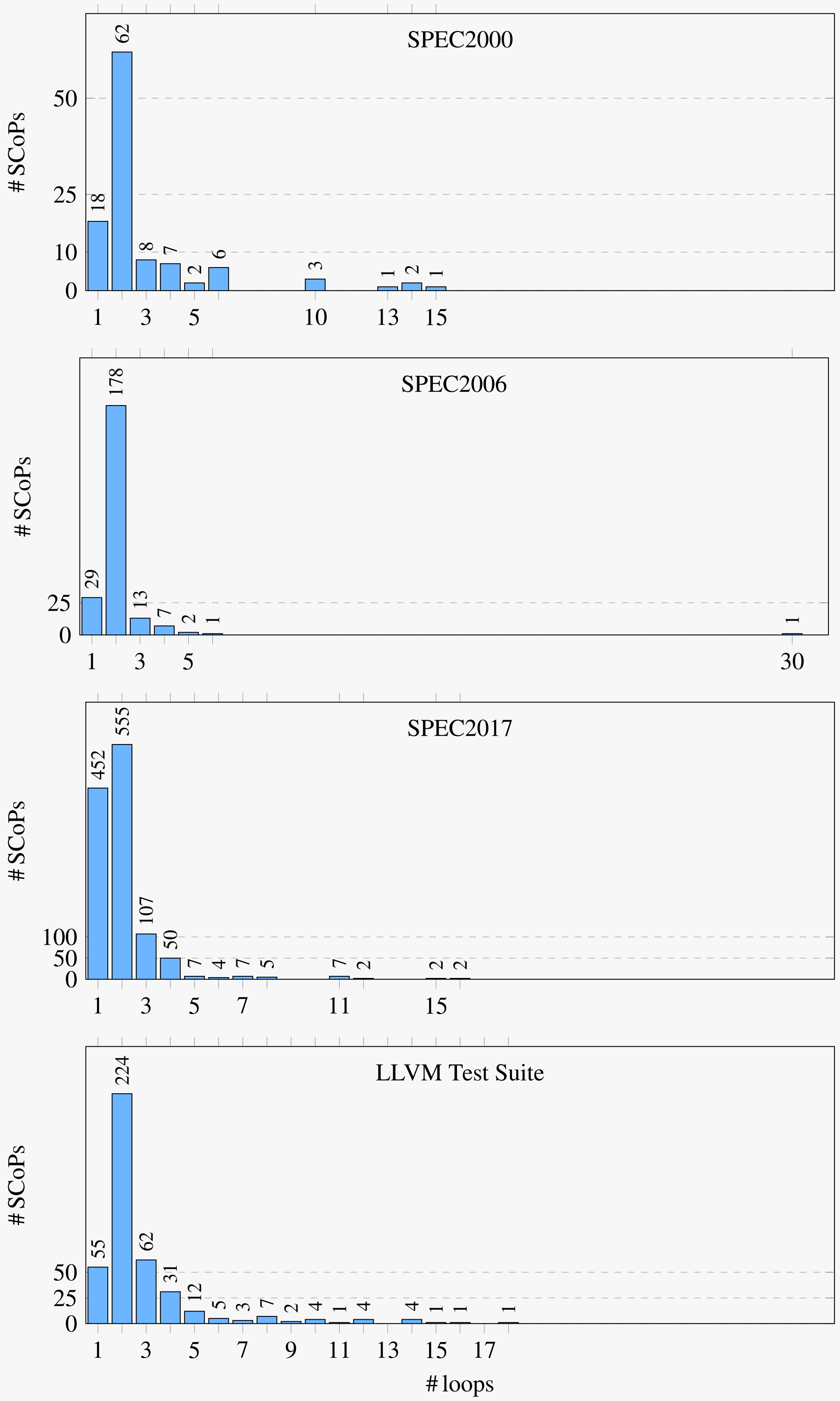


Figure 3.7: Number of SCoPs with regards to the number of contained loops.

3.1.3 Profitability Metrics for Automatic Polyhedral Optimization

Profitability metrics for automatic optimization need to consider the achievable benefits but also the compile time costs as well as alternative optimizations. Table 3.8 lists (mostly loop) optimizations that can be performed by the polyhedral optimizer Polly[4] and natively by LLVM.

Optimization	**lda** [a]	**# loops**	**# stmts**	**Support in** LLVM	**Type**
loop rotation	✗	1	1+	full	A
loop peeling	✓	1	1+	full	
loop unroll	✓	1	1+	full	
loop fission	✓	1	2+	experimental, innermost	
loop reordering [b]	✓	2+	1+	none	B
loop unswitching	✓	1+	1+	invariant conditions	
invariant load hoisting	✓	1+	1+	full, but in isolation [c]	C
runtime alias checks	✓	1+	1+	innermost	
dead store elimination	✗	1+	1+	straight line code + patterns	
memory propagation [d]	✓	1+	2+	limited [e]	
loop vectorization	✗	1	1+	innermost [f]	D
loop parallelization	✗	1	1+	none	
loop interchange	✗	2+	1+	experimental	E
loop fusion	✓	2+	2+	none	
loop skewing	✓	2+	1+	none	
loop tiling/blocking	✗	2+	1+	none	

Table 3.8: Classical (loop) transformations that can be performed by the Polly loop optimizer. The center columns provide a brief summary of the application requirements. The second to last column describes the support available in the LLVM compiler toolchain.

[a] Low-dimensional write accesses, e.g., scalar writes, will inevitably cause loop carried dependences without the use of elaborate techniques [DSH18; Fea88a; VC16; ZKC18] (see also Section 5.3). To indicate the sensitivity, we use ✗ (sensitive → not applicable) and ✓ (not sensitive → potentially applicable).

[b] Loop reordering refers to a change in the order of a sequence of loops not their iteration order.

[c] In Section 3.6 we show that load hoisting is in practise far more powerful when performed in combination with runtime alias checks (ref. Section 4.1) as there is a cyclic dependence between both optimizations.

[d] Memory propagation will forward expression from a store to a load that would otherwise read the expression result (ref. Section 5.3). The store and the involved memory can be eliminated if they are not live-out (ref. Section 5.3.6.1).

[e] Memory propagation, or "store-load-forwarding", is only performed for dependences of length one in the innermost loop and for straight line code.

[f] Extensions for *real* outer loop vectorization are available [NZ08]. Transformations like unroll-and-jam [CCK90] allow to emulate outer loop vectorization.

[4] Not all optimizations listed in Table 3.8 are available in the upstream version of Polly. Similarly, we did not consider all extensions and optimizations proposed for Polly or other polyhedral optimizers.

The transformations are shown with the number of involved loops (# loops) and statements (# stmts). We grouped them by type, A to E, depending on their complexity, optimization potential and availability in LLVM. The second column indicates if POLLY is impacted by low dimensional accesses (lda), i.a., scalars. However, it is important to note that there are evolved techniques to eliminate such low dimensional access dependences [DSH18; Fea88a; KG18; VC16] which need to be taken into account (ref. Section 3.1.4). The table illustrates that LLVM is tuned to optimize single loops in isolation (type A), especially innermost ones. Both observations are not surprising given that these cases allow for performance benefits with relatively little compile time overhead and optimization complexity. Polyhedral optimizations classically achieve the most impressive performance improvements with transformations that involve multiple loops (type E) or if the precision of data-dependences is crucial for the effectiveness and applicability (type C to E). The efficacy of the remaining transformations (type B) is hard to determine, especially since loop unswitching can be subsumed by loop peeling and loop fission/distribution.

A profitability metric for automatic polyhedral optimization should dismiss SCoPs that can be otherwise optimized e.g., innermost loops, as well as SCoPs that exclude most transformations based on their characteristics and the available modeling and optimization techniques [KP16]. Loops with low dimensional write accesses are generally not directly amenable to transformations that change the iteration order, e.g., parallelization. Transformations like interchange, skewing or tiling will additionally require nested loops. Other optimizations e.g., fusion or reordering, are only applicable to a sequence of loops that are executed under a common control predicate. Given these and other constraints it remains to determine how many loops and statements in a SCoP are amenable to transformations.

3.1.4 Profitability Checks In Polly

In LLVM/POLLY, profitability of a SESE region, and later of the generated SCoP, is checked multiple times throughout the transformation pipeline. Regions without loops, regions with a single loop and a single statement as well as regions that only read or write memory are dismissed early. This eliminates ≈ 47% of all SESE regions prior to polyhedral modeling (ref. Figure 3.20). After the polyhedral model (or SCoP) is built, the simple heuristic shown in Algorithm 3.9 will check if multiple loops or statements are amenable to transformations. This step is done twice. First, scalar accesses are ignored as they might be eliminated during the following SCoP canonicalization phase. This phase aims to remove scalar phi nodes and simple scalar accesses. For the latter, the used technique [KG18] is however not as powerful as our polyhedral expression propagation [DSH18] explained in Section 5.3. SCoPs that were deemed profitable prior to canonicalization are checked again afterwards, this time low-dimensional write accesses, though only *scalars*, are taken into account. However, the simple heuristic we introduced into upstream POLLY will not consider control conditions nor dependences caused by low-dimensional *memory* write accesses. As both are important for the optimization potential, many SCoPs will be deemed profitable even though no beneficial transformation can be applied. The two final profitability checks will determine if the newly computed schedule differs from the original one and if any beneficial transformation was applied prior to the code generation step. However, these checks are often too simple to determine canonicalizations that do not necessarily improve performance but alter the schedule and potentially the generated abstract syntax tree (AST).

```
procedure BASICPROFITABILITYCHECK(S : SCoP, PreCanonicalization : bool)
    Initialize the counter for the number of optimizable statement dimensions (OSD).
  OSD ← 0
    Prior to canonicalization all loops are considered potentially profitable. After canonicalization
    only the loops surrounding statements with array but no scalar writes are counted as optimizable.
  for stmt in S do
    if PreCanonicalization then
      OSD ← OSD + stmt.getNestingDepth()
    else if stmt.hasArrayWrites() and not stmt.hasScalarWrites() then
      OSD ← OSD + stmt.getNestingDepth()
    end if
  end for
    Return profitable only if a minimum of two optimizable statement dimensions was found.
  return OSD > 1
```

Algorithm 3.9: The simple SCoP profitability check performed by LLVM/POLLY that is run before and after canonicalization passes tried to eliminate scalar accesses.

3.1.5 Piecewise Polyhedral Profitability Heuristic

The profitability heuristic proposed in the following is based on the piecewise defined, affine profitability function generated by Algorithm 3.11. This function relates parameter conditions to the number of loop dimensions, thus loops per statement, that can potentially be transformed under these conditions. The uncertainty comes from the lack of dependences. Since they are expensive to compute, we believe it is prudent to construct them only for SCoPs with a reasonable optimization potential. To this end, we count only transformable loops that might not carry dependences[5]. Depending on the configuration of the polyhedral optimizer and the expected benefit from different transformations, the least number of loops required to apply a beneficial transformation varies. However, we can specify a lower bound to limit the costly dependence analysis and scheduling step. Given that LLVM performs well for single loops it is reasonable to

[5] If tiling/blocking is enabled and considered a worthwhile optimization for the input code, the piecewise profitability heuristic will never consider low-dimensional accesses (lda) as problematic.

require at least two transformable loop dimensions. Hence, only SCoPs that contain a statement which is surrounded by two transformable loops, or multiple statements executed under (partially) the same control conditions surrounded by one transformable loop, are deemed profitable.

To showcase differences between the two heuristics we provide four examples in Figure 3.10. Assuming that none of the statements contain scalar accesses, all four examples are deemed profitable by LLVM/POLLY. However, even if all contained memory accesses are dependent on the loop counters, only two of the four example are profitable according to the piecewise heuristic. The examples in parts 3.10a and 3.10d are dismissed as there are no parameter values (for `N` and `p`) for which both `S` and `P` would be executed. For the other two examples such a context exists. Regarding the loop fission/distribution opportunity for the last two examples, it is worth to note that LLVM can natively perform loop unswitching for loop invariant conditions. Thus, LLVM can unswitch the conditional in the unprofitable example (part 3.10d), but not in the example deemed profitable (part 3.10c).

```
for (i = 0; i < N; i++)
  S(i);
for (j = N; j < 0; j++)
  P(j);
```

(a) Unprofitable SCoP, according to the piecewise heuristic, with the profitability function $(N > 0\,?\,1 : 0) + (N < 0\,?\,1 : 0)$.

```
for (i = 0; p && i < N; i++)
  S(i);
for (j = 0; q && j < M; j++)
  P(j);
```

(b) Profitable SCoP, according to the piecewise heuristic, with the profitable context $p \wedge N > 0 \wedge q \wedge M > 0$.

```
for (i = 0; i < N; i++)
  if (i < p)
    S(i);
  else
    P(i);
```

(c) Profitable SCoP, according to the piecewise heuristic, with the profitable context $0 < p < N \wedge N > 0$.

```
for (i = 0; i < N; i++)
  if (p)
    S(i);
  else
    P(i);
```

(d) Unprofitable SCoP, according to the piecewise heuristic, with the profitability function $(N > 0 \wedge p\,?\,1 : 0) + (N > 0 \wedge \neg p\,?\,1 : 0)$.

Figure 3.10: Four simple examples that, assuming `S` and `P` contain only array accesses, are considered profitable by LLVM/POLLY (ref. Algorithm 3.9), but not necessarily by the piecewise profitability heuristic. The shown profitability functions and profitable contexts are generated by Algorithm 3.11.

While we evaluated the piecewise profitability heuristic as a replacement of the one currently used in LLVM/POLLY, which is a boolean predicate, it is more than that. Especially for larger SCoPs, the heuristic can identify profitable parts which should consequently be optimized. However, it also determines non-profitable parts, or more precise parameter conditions, under which no profitable optimization is possible. If a SCoP is specialized to profitable inputs (ref. Section 3.5 and Section 6.3), later steps, e.g., dependence analysis, can become less costly.

```
procedure ProfitabilityFunctionGeneration(S : SCoP)
```

Determine the profitability of each statements in the SCoP and summarize the result in the piecewise defined, affine profitability function Ψ that determines profitability for a parameter context. Initially Ψ will return 0 for all parameter configurations.

```
  $\Psi \leftarrow \lambda c : 0$
  for stmt in S do
```

All loops surrounding the statement and part of the SCoP are initially potentially transformable. If tiling is enabled and considered a worthwhile optimization, these loops are all counted. Otherwise, loops that are likely to carry dependences are excluded in the following.

```
    Loops ← S.getSurroundingLoopsInSCoP(stmt)
    if not consider tiling then
```

Exclude loops that will probably carry dependences induced by a write access. To this end, we only consider loops transformable that are involved in the access function f_w of the write access.

```
      foreach write access $w$ in stmt do
        VaryingLoops ← S.getInvolvedLoopsInSCoP($f_w$)
        Loops ← Loops.intersect(VaryingLoops)
      end foreach
```

Exclude loops that probably carry dependences induced by a dependent write access. If the defining write of a read access is not known we optimistically assume no dependences to preserve the optimization potential.

```
      foreach read access $r$ in stmt do
        if defining write $w_r$ is unknown then continue
        foreach loop l in Loops do
          if l contains $w$ then
            Loops ← Loops.remove(l)
          end if
        end foreach
      end foreach
    end if
```

Add the number of potentially transformable loops under the statement's execution context.

```
    $\Psi \leftarrow \lambda c : \big((\Psi\, c)$ + $(c \in \pi_\rho(\mathcal{D}_{\texttt{stmt}})$ ? |Loops| : 0$)\big)$
  end for
```

Finally, restrict the profitability function to the assumed context (ref. Section 3.5) which will exclude all parameter combinations that would violate an assumption taken in order to represent the SCoP.

```
  $\Psi \leftarrow \lambda c : (c \in$ S.getAssumedContext() ? $(\Psi\, c)$ : 0)
  return $\Psi$
```

Algorithm 3.11: Algorithm to determine the profitability function of a SCoP. The resulting piecewise affine function Ψ maps parameter conditions to the number of potentially transformable loops based on the iteration contexts and access functions of the SCoP statements as well as the assumptions taken to model the SCoP (ref. Section 3.5).

3.1.6 Profitability Evaluation

We evaluated the profitability algorithms by inspecting the schedule [Bon+08] and the final AST [GVC15] generated by LLVM/Polly. A good metric should result in a high percentage of optimized, or better transformed, SCoPs. For the former we used the same (simple) check as Polly to determine if two schedules differ. However, canonicalizations of the internal representation [Ver+14] can cause differences even though there is no functional change. For ASTs, a syntactic comparison would yield similar results mostly due to benign variations. To this end, we compare the AST generated for the original and optimized schedule not just for syntactic equality but we allow reorderings of loops and statements inside a loop or at the outermost level.

Difference	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
Schedule	76.0%/72.0%	48.2%/46.4%	78.4%/77.1%	68.2%/63.7%
AST Structure	18.4%/13.9%	28.7%/26.0%	54.0%/52.8%	43.3%/38.1%

Table 3.12: Percentage of SCoPs with an altered schedule and AST structure after LLVM/Polly performed schedule optimizations. The two numbers per benchmark show the impact tiling has on the result (first with, then without tiling).

Table 3.12 shows the percentages of SCoPs with an altered schedule and AST structure after LLVM/Polly performed schedule optimizations on them. The results are collected with and without tiling, a transformation which is applied after the common polyhedral schedule optimizations. Due to the native profitability checks of LLVM/Polly (ref. Section 3.1.4), only SCoPs with an altered schedule were considered for the AST structure test. The percentage of SCoPs with an altered AST structure can be as low as 18.4%/13.9%, with and without tiling respectively. Hence, there is much room for improvements with regards to the profitability heuristic. Early identification of unprofitable loops will save compile time, but falsely pruned SCoPs will decrease the overall optimization potential. To this end, we want to carefully dismiss more unprofitable SCoPs early but ensure profitable ones are kept.

For the piecewise profitability heuristic we also relate the prediction for a SCoP with the actual change in schedule and AST. To make the comparison fair, we evaluated the prediction only for SCoPs deemed profitable by LLVM/Polly. However, it is important to note that our heuristic is strictly more restrictive than the one currently used (ref. Algorithm 3.9). In Figure 3.14 the results for schedules are shown. Because schedules are tested for equality, the number of mispredicted SCoPs is high. Depending on the benchmark suite and configuration, it varies between 34.2% and 58.5%. Though, only SCoPs classified as unprofitable ▦ but with an actual different schedule/AST are problematic since conservative mispredictions[6] 🟧 are not different from the status quo. For schedules, this number is still high and between 9.8% and 33.0%. The same

[6] We denote incorrectly predicted, actually equal schedules/ASTs as conservative mispredictions. Since we focus on preserving optimization potential, our conservative check will generally assume profitability.

evaluation for the AST structure is shown in Figure 3.15. Note that we do not test for equality here but for "functional changes" as explained above. The number of problematic mispredicted SCoPs, thus the ones we classified as unprofitable but that exhibited non-trivial changes, is now down to less than 2% in all SPEC configurations and 13.1% for the LLVM Test Suite. Since the absolute number of these misclassification was only 23 (with and without tiling), we were able to analyze the AST of each misprediction manually. All but the single one in the SPEC2006 benchmark suite stem from general loop fission/distribution (type A in Table 3.8). The exception was a loop unswitching transformation that hoisted code out of the loop which was only executed for the first iteration (type B in Table 3.8). If multiple mispredictions occurred in a test suite (SPEC2017 and LLVM-TS), their ASTs were all identical. This is a strong indicator that it was initially one piece of code that was duplicated either manually or through automatic canonicalization techniques, e.g., inlining or loop unrolling.

Our experiments show that 20.3% to 45.8% of all SCoPs, depending on the configuration, would be additionally dismissed early as there is little to no chance for an optimized AST. As discussed above, this would cause only a marginal reduction in optimization potential. Note that the prediction was performed right after the SCoP modeling and canonicalization step, thus before polyhedral dependence analysis, schedule, or AST generation. Hence, the piecewise profitability heuristic would prevent these costly steps completely for a much larger number of unprofitable SCoPs without loosing significant optimizations in the process.

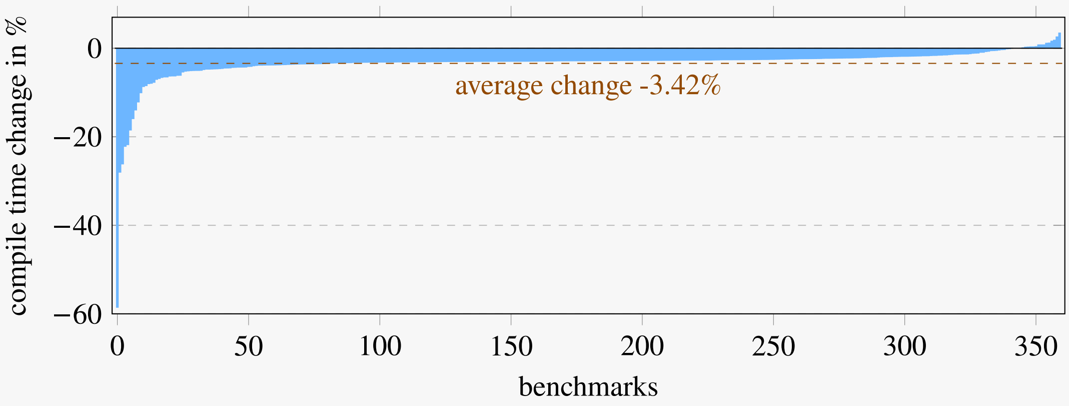


Figure 3.13: Compile time impact of the piecewise profitability metric presented in Algorithm 3.11. The baseline, i.e., 0%, is LLVM/POLLY and we compared the median of 51 runs.

To quantify the benefit of the piecewise profitability heuristic we measured the impact on compile time for all our benchmark suites (ref. Table 2.5 and 2.6). The results are illustrated in Figure 3.13 as percentage differences compared to our baseline version of POLLY (ref. Section 2.3). We observed an average compile time save of 3.42% and ten benchmarks are improved by more than 10%. There were only 16 out of the 360 benchmarks for which the results deteriorated. In the worst case, compile time was increased by 3.54%. We believe these exceptions were caused by the cost of the more evolved profitability checks in a short running compilation process.

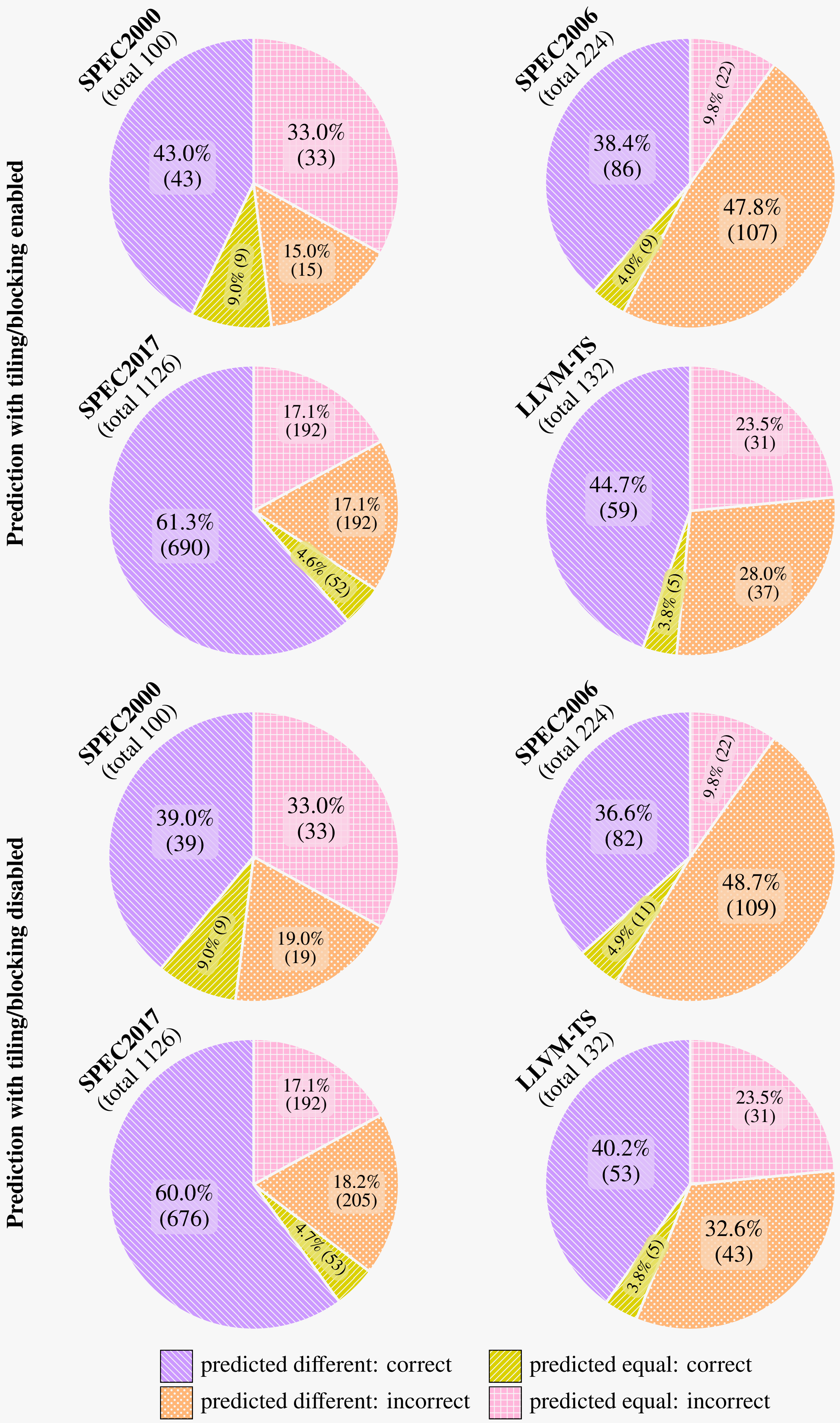


Figure 3.14: Piecewise profitability heuristic results that predicted the equality of the original and generated schedules. The upper part shows the results with, the lower part without tiling.

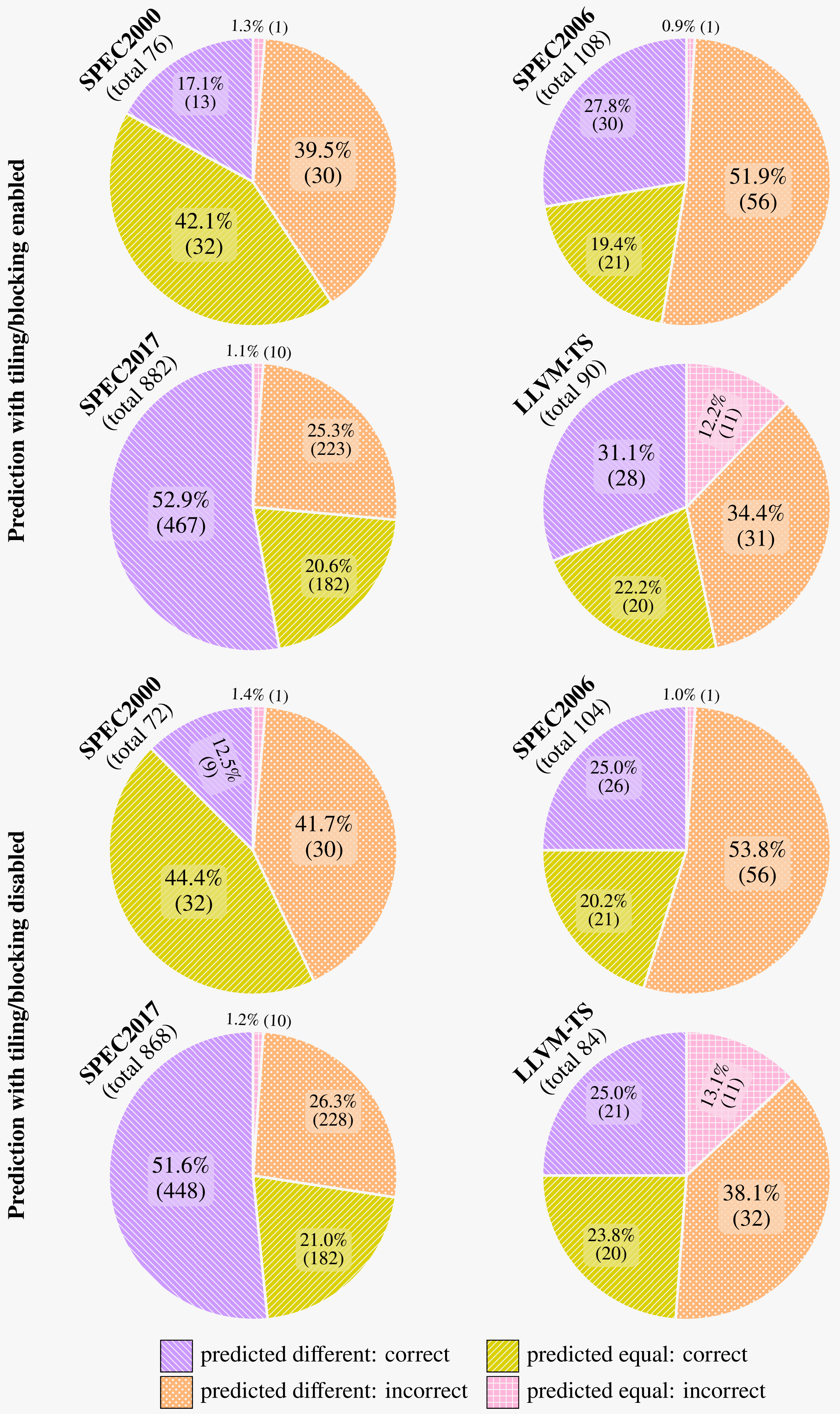


Figure 3.15: Piecewise profitability heuristic results that predicted the equality of the original and generated AST. The upper part shows the results with, the lower part without tiling.

3.2 Applicability Limitations

Applicability, profitability, and performance are crucial in the evaluation of existing polyhedral approaches. However, we need to understand their limitations to predict the potential for future extensions and guide the development of polyhedral tools. Only if the applicability restrictions are well understood, we can determine if and how polyhedral-model-based tools will become more applicable, leading to automatic and robust optimization of general purpose code. To this end, we conducted two studies, an early one in 2013 [Doe+13], and a second one in the course of this thesis, almost five years later. This section describes and interprets the results in order to provide a better understanding of the limitations of automatically applied polyhedral techniques.

To argue about applicability limitations we require two kinds of information. First, we need to explore which low-level rejection reasons prevented regions in the input program to be valid SCoPs. Second, we have to identify in which high-level limitation category these rejection reasons fall, thus if they are caused by implementation artifacts or are inherent to the underlying model. If used together, this knowledge can guide the development of future extensions to polyhedral tools such that the cost-benefit ratio is optimal. Hence, applicability is improved with minimal developing effort and as little as possible complexity increase.

In addition to our studies there is one by Simbürger et al. [Sim+13] that focuses on the impact of certain applicability limitations with regards to dynamic SCoP coverage (ref. Table 3.1). While we are not aware of other dedicated studies, extensions to the polyhedral model do often provide some insights about applicability limitations by measuring their specific impact.

3.2.1 Rejection Reasons

Rejection reasons are low-level problems that prevented the applicability of polyhedral modeling techniques, e.g., dynamic or non-affine memory accesses. Our first study [Doe+13] was focused on understanding these problems and measuring their impact on general purpose code. To this end, we modified the LLVM/POLLY polyhedral optimizer to collect the causes for a SESE region to be deemed invalid for polyhedral representation. The results of this study are shown in Table 3.16. It is a ranking of the rejection reasons observed for regions in some SPEC2000 benchmarks. In addition to the number of occurrences of a specific cause as part of the rejection reasons for a region (# Occur.) and the number of regions only rejected because of a specific

> This section discusses and compares two applicability studies we performed with LLVM/POLLY. The first in 2013 [Doe+13], and the second in the context of this thesis, almost five years later. The detailed enumeration of rejection reasons shown here can be reproduced using the `rejection_reasons` branch of our research prototype.

cause (# Only Reason), we derived accumulative numbers (last column). These allow to reason about the interplay of the most often occurring rejection reasons. As an example we can look at potential aliasing, the second to most important rejection reasons. For 18.9% of all regions rejected due to aliasing it was the sole problem that prevented polyhedral modeling. In an additional 27.7%, aliasing and non-affine expressions were cause for dismissal. Based on these numbers we concluded that non-affinity[7], aliasing, and function calls were the biggest limiting factors with 45.7%, 24.1%, and 11.7% of the counted rejection reasons respectively. However, while the remaining reasons did not occur as often, they are more clearly defined and thereby easier to remedy in practise. Thus, if feasibility and the development cost is taken into account, the most often occurring reasons might not necessarily have the best cost-benefit ration. In the remainder of this thesis we present extensions to the polyhedral model that tackle all of the rejection reasons collected in this first study. Non-affinity is addressed in Section 3.6, potential aliasing in Section 4.1, function calls in Section 6.1, complex loop counters in Section 6.4, complex control flow in Section 3.3, and unsigned comparisons in Section 4.3. In addition, we present different approximation schemes in Section 3.4 and Section 6.3 which can eliminate various applicability limitations.

i	**Rejection Reason**	**# Occur.**	**# Only Reason**	**# Only Reasons** (1 to *i*)
1	non-affine expression	1230	84	84
2	potential aliasing	1093	207	510
3	non-affine loop bounds	840	6	660
4	function call	532	72	928
5	complex loop counter[a]	384	0	1174
6	complex CFG[b]	253	31	1387
7	unsigned comparison	199	0	1586
8	others	1	0	1587

Table 3.16: Rejection reasons for code regions in SPEC2000 benchmarks[c] as reported by Doerfert et al. [Doe+13]. The first two columns enumerate and name the reasons in descending order of their occurrence (# Occur.). If a region was only rejected due to a single cause, it is included in the count presented in column four (# Only Reason). The last column shows the accumulated effect of a rejection reason and all preceding ones.

[a] LLVM/Polly did consider all loop counters that could not be canonicalized as too complex. A canonical loop counter would start at zero and be incremented by one in each iteration.

[b] LLVM/Polly did consider non-structured control flow (which is not representable with for/while/if) as too complex. This included switches and loops with multiple back edges (continue) or exits (break).

[c] The evaluation environment was based on the Sambamba framework [Str+12] which could only handle a subset of all SPEC2000 benchmarks at the time, namely: ammp, art, bzip2, crafty, equake, gzip, mcf, mesa, and twolf. Thus the numbers are only extracted from these benchmarks.

[7] We use non-affinity to summarize reasons 1 and 3, thus non-affine or dynamic memory accesses (1) as well as non-affine or dynamic control flow conditions (3).

In Section 3.1 we established that the number of SCoPs, thus valid code regions, is not necessarily a good indicator for applicability. A similar reasoning holds for occurrences of rejection reasons, i.a., a single problem is counted multiple times if it is nested in many SESE regions. Though, since such nested regions do almost always correspond to nested loops, the multiple counts could be interpreted as weight for the lost optimization potential. In addition, we believe that rejection reason counts do still allow to determine general trends. This hypothesis was independently validated by the study of Simbürger et al. [Sim+13]. That study came to a consistent conclusion with regards to the importance of our top four rejection reasons. In contrast to our work, they measured the dynamic SCoP coverage (ref. Table 3.1) on a large corpus of real world applications. Their measurements were taken with and without the assumption that the top four rejection reasons could be eliminated, thus ignored for the purpose of their experiment.

3.2.2 Limitation Categories

In our experience, polyhedral applicability limitations can be divided into the four categories: implementation, representation, conceptual, and profitability limitations. In contrast to low-level applicability issues (also referred to as rejection reasons), the applicability categories shown in

(I)mplementation limitations are caused by incomplete or not well integrated parts in the polyhedral toolchain. This category also includes complex data types, e.g., library collections, and situations where information is known but not yet available to the tool, e.g., the possible side-effects of library functions.

(R)epresentation limitations are caused by inputs that do not have an equivalent polyhedral representation but for which the semantic differences can be overcome. This category includes certain function calls, integer arithmetic, signedness, aliasing arrays as well as certain kinds of dynamic and non-affine expressions.

(C)onceptual limitations are inherent to the (static) polyhedral model itself which makes them hard to overcome. Instead, approximations as well as non-polyhedral extensions and dynamic techniques are required. This category includes calls to unknown functions as well as various forms of dynamic and non-affine expressions.

(P)rofitability limitations indicate valid SCoPs that were chosen to be rejected for profitability reasons. In LLVM/Polly, this category includes regions without loops but also SCoPs with only one kind of memory instruction, either load or store and SCoPs with a single polyhedral statement contained in a single loop.

Table 3.17: Distinct categories of limitations that require different kinds of future extensions to improve applicability and robustness of polyhedral tools.

Table 3.17 are high-level abstractions of the underlying problems. As such, they help to reason about the *kind* of work necessary to remedy existing limitations and thereby improve polyhedral applicability, robustness and profitability. As an example we take a look at non-affine expressions. If a polyhedral optimizer, e.g., LLVM/POLLY, reports a non-affine expression in an access relation or control flow condition, it is not immediately clear if the problem is an implementation limitation, a representation shortcoming or caused by a semantically non-affine computation, hence if it is conceptual. Examples for these different kinds of expressions that are, or were, reported by LLVM/POLLY as non-affine are shown in Figure 3.18. The first one (ref. 3.18a) is actually a *quasi affine* computation, well in the reach of modern polyhedral tools [Ver10]. It is not detected and represented as such because the handling of bitwise logical operations (involving a constant) was never integrated into POLLY's codebase[8]. This implementation limitation would be easily corrected as soon as code containing such a pattern become interesting. In the second example (ref. 3.18b), the load of the index variable from memory is generally a dynamic and thereby non-affine value. However, since the address is a constant global variable it can be treated as a parameter, thus a symbolic constant. LLVM/POLLY is able to recognize and model this situation due to the representation extension we present in Section 3.6. Finally, the third example (ref. 3.18c) shows a piecewise defined, partially affine access function. While piecewise defined expressions are generally supported by polyhedral analyses, optimizations and code on schemes, there is little front-end support for them. Thus, the input code is not properly translated to a piecewise defined polyhedral representation but instead classified as non-affine (ref. Section 6.4). Even if piecewise affine expressions would be recognized as such, without further knowledge of the function `foo`, the expression would still be dynamic/non-affine for part of the inputs, namely for `i` $\geq$ `M`. As dynamic expressions are a conceptual limitation we would need to employ approximations (ref. Section 3.4) or represent only the statically affine part (ref. Section 6.3). These examples illustrate that a single rejection reason can fall into either category and even multiple ones depending on the input. While we classify actual rejection reasons into limitation categories in the following evaluation of our second study, we realize that more fine-grained reason tracking is necessary to determine an injective mapping.

```
idx = i & 3;
A[idx]++;
```

(a) A bitwise logical operation with a constant is a quasi-affine operation[8] but often considered non-affine.

```
idx = *ConstGlobalVar;
A[idx]++;
```

(b) A load of a constant global variable is known to be invariant it can consequently be treated as unknown but fixed value, thus a parameter (ref. Section 3.6).

```
idx = i < M ? i : foo(i);
A[idx]++;
```

(c) A piecewise defined expression with a statically affine part and a potentially dynamic part, depending on the function `foo` (ref. Section 6.1 and Section 6.4).

Figure 3.18: Three access functions that are or were recognized as non-affine by LLVM/POLLY. Note that each example falls into a different limitation categories (ref. Table 3.17). This indicates that at least this particular reason is too coarse grained to determine how future extensions have to look like in order to remedy the problem.

[8] Support of bitwise logical operations with constants is available in the `bitops` branch of our research prototype.

3.2.3 Evaluation

In our second and more elaborate study we answer four specific questions aimed to reveal the limits of polyhedral tooling, or better LLVM/POLLY, on low-level programs today:

1. How did improvement to POLLY, partially presented in this thesis, impact rejection causes?
2. Which rejection reasons do provide the best cost-benefit ratio for future extensions?
3. Which combination of rejection reasons prohibit polyhedral modeling the most?
4. How many distinct rejection reason do prevent polyhedral modeling for an average SCoP?

Question 1. To answer the first questions we collected rejection reasons and their occurrences as we did for the first study. Though, the number of benchmarks, as well as the granularity of the rejection reasons, is now significantly higher. The results, ordered by occurrences, are shown in Table 3.19. The first column is short name of the problem while the fourth column provides a brief description. Columns two and three show the number of affected regions as well as how many of them were only rejected because of the particular cause. In the last column we placed the rejection reason into one or more of the limitation categories described in Table 3.17. Note that LLVM/POLLY has preconditions and profitability conditions that have to be fulfilled for a region to be considered valid. If not, the region is dismissed and the condition is counted as cause. However, only preconditions are checked in the very beginning while profitability conditions are checked after a region was deemed valid. Thus, rejections due to preconditions might contain other violations while regions dismissed due to profitability conditions are in-fact valid regions.

While our early results, shown in Table 3.16, do not account for preconditions and profitability rejections, we can compare the remaining ones. The former top rejection cause, non-affinity, still has the most impact today. Though, the new results describe the location of non-affinity better and they differentiate different kinds of problems that were summarized as non-affine before, i.a., variant base pointers (VBP). First note that most instances of unknown loop bounds (LB) had to be caused by non-affine control flow conditions as the other reasons (IT, IC, and MP) could only account for $\approx 15\%$ of the instances. This also means that most non-affine control conditions ($\approx 84\%$) are part of a loop control branches, thus back edge and exit conditions. The reason for this is twofold. First, the no loop precondition (NL) prevents non-affinity tests for SESE regions that do not contain loops. Second, our evaluation setup (ref. Section 2.3) does allow control flow approximations for innermost non-affine conditionals but approximations for non-affine loops are disabled (ref. Section 3.4.2). After non-affine (loop) control flow, the next most often occurring rejection reasons are concerned with non-affine and dynamic memory accesses (NAA and VBP). Since neither function call handling described in Section 6.1 nor function call approximation presented in Section 3.4.3 were enabled in this experiment, it is hardly surprising that they occupy a top slot with regards to the number of occurrences. Rejections because of a single reason are by far lead by function calls, if we exclude preconditions and profitability rejection reasons. The runtime alias checks shown in Section 4.1 reduced the number of rejections due to potential aliasing pointers (PA) significantly. Similarly, the extensions to the control flow representation described in Section 3.3, as well as the handling of unsigned expressions explained in Section 4.3, basically eliminated rejection reasons 5 to 7 shown in Table 3.16. However, some complex loop counters will nevertheless cause non-affinity (NAB and NAA).

Abbr.	# Occur.	# Only Occ.	Description	Cat.[a]
NL[b,c]	396869	396869	no loop in region	(P)
NAB	326563	138	non-affine control flow condition	(I,R,C)
LB	310649	0	unknown loop bound (due to IT, IC, MP, or NAB in a loop control flow condition)	(I,R,C)
NAA	224630	11759	non-affine access (ref. Figure 3.18)	(I,R,C)
VBP	188412	3210	variant memory access base pointer	(I,R,C)
FC	185142	23204	function call (ref. Section 3.4.3 & 6.1)	(I,R,C)
PA	96720	4663	pot. aliasing pointers (ref. Section 4.1)	(I,R)
EO[c]	84359	218	operand in error block (ref. Section 6.3)	(I)
TL[b]	61051	61051	top level region (covering the function)	(I)
IC	30993	376	invalid control flow (e.g., cmp. of floats)	(R,C)
MP[c]	19939	1472	multiple pointers in control flow condition (often implies actual aliasing)	(I,R)
ITP	15575	1564	integer to pointer instruction	(I)
NLS[d,c]	13666	13666	either no loads or no store instructions	(P)
SSL[d]	13125	13125	only single statement in a single loop	(P)
EP[c]	12256	2	phi node operand in error block (see EO)	(I)
EI[c]	1006	0	exception handling	(I,R)
AVM	832	0	atomic/volatile memory access	(I,R)
NBP	655	24	unknown memory access base pointer	(I,R,C)
SA	640	6	stack allocation	(I)
UIE[b]	89	89	unreachable region exit (i.a., `exit()`)	(I)
IR	87	0	irreducible control flow	(R)
UI	42	0	unknown instruction	(I,R,C)
IT	37	1	no branch nor switch control flow	(I)
INF	19	0	obvious infinite loop (ref. Section 5.1.3)	(R)

Table 3.19: Rejection reasons collected for the benchmarks shown in Table 2.5 and 2.6. The number of SESE regions rejected because of them is shown as well as a description and the limitation categories they fall into. In total 915422 regions were rejected.

[a] Multiple limitation categories if no single one is plausible (ref. Figure 3.18).

[b] Precondition for SCoPs enforced by LLVM/POLLY. Occurrence implies only rejection reason.

[c] Not explicitly reported by vanilla LLVM/POLLY but only in the `rejection_reasons` branch.

[d] Profitability condition for SCoPs enforced by LLVM/POLLY. Occurrence implies only rejection reason.

Question 2. The second question is concerned with the cost-benefit ratio for future extensions. To this end, we should look at the most frequent rejection causes only categorized as implementation limitations, namely error blocks operands (EO and EP) and top level regions (TL). As such, they are perfect candidates for simple, cost effective, applicability extensions. Though, the latter (TL) is a precondition of LLVM/POLLY (ref. Section 6.1) which means the actual benefit cannot be predicted from these results. Other beneficial improvements that fall either in the implementation or representation category can be found by examining the limitations of the alias checks (ref. Section 4.1.3), e.g., adding the possibility to represent presumably aliasing pointers (MP), as well as various forms of non-affine expressions (ref. Figure 3.18 and Section 6.4). Rather simple but less beneficial extensions could introduce support for certain integer to pointer conversions (ITP), stack allocations (SA), as well as atomic and volatile accesses (AVM).

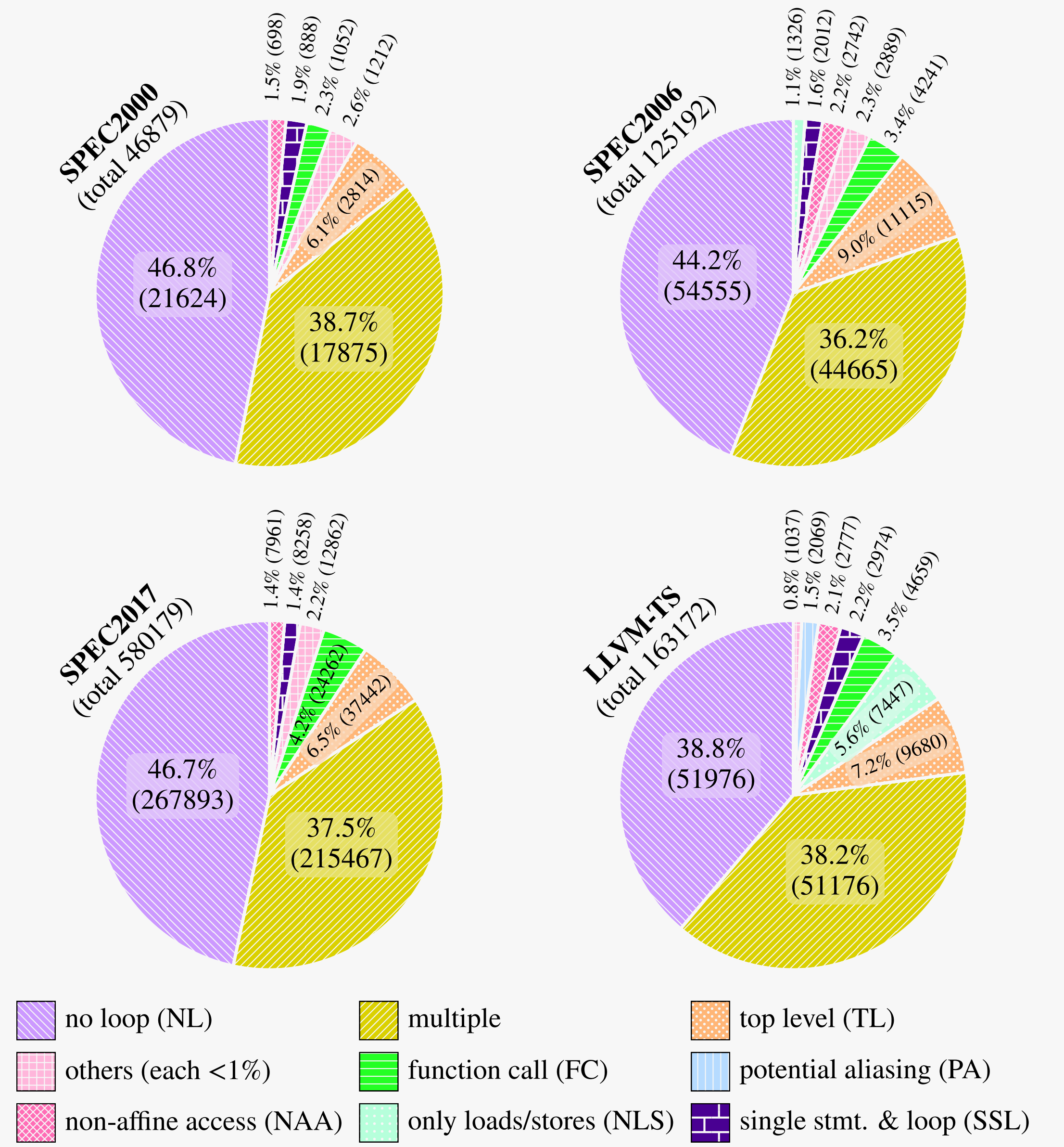


Figure 3.20: Rejection reasons for all analyzed SESE regions. The most often occurring combinations (part of multiple reasons ▨) are shown in Figure 3.21.

Question 3. To answer question three we have to look at the rejection reasons that prohibit polyhedral modeling on their own, or in combination with other reasons. To this end, we show the distribution of unique causes for dismissal in Figure 3.20. The unique reasons are dominated by the requirement for loops in the SESE region (NL) with 38.8% to 46.8%. The second most often occurring unique reason are top level regions (TL) which is also a precondition of LLVM/POLLY. Afterwards, there are profitability limitations, e.g., the requirement for loads and stores (NLS) as well as multiple statements or loops (SSL), trailing with 1.4% to 7.8%. The remaining unique rejection reasons are function calls (FC) with 2.3% to 3.5%, non-affine accesses with 1.4% to 2.2%, and potentially aliasing pointers with up to 1.5%. Since combinations of rejection reasons account for ≈ 37% of all dismissals across the benchmark suites, we broke them down in Figure 3.21.

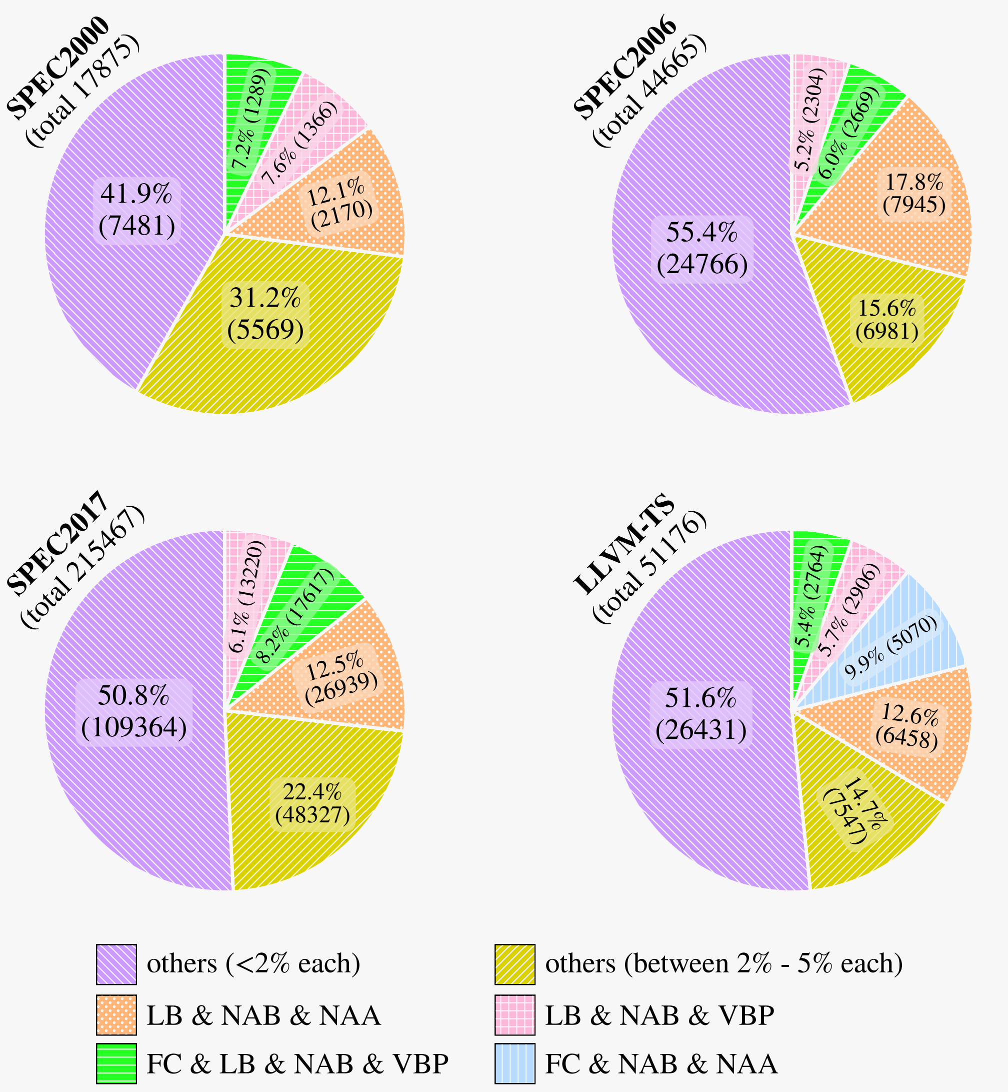


Figure 3.21: Combinations of rejection reasons (ref. Table 3.19) that caused an analyzed SESE region not the be a valid SCoP. Combinations that cause less than 2% of the regions to be rejected are summarized as ▨ and the ones that rejected between 2% and 5% are shown as ▨.

The variety of different combinations identified in the different benchmark suites is quite large. Each one features ≈ 1100 different combinations that each caused less than 2% of regions to be rejected ▧. The number of combinations that rejected between 2% and 5% of the regions ▨ is relatively low and varies between 5 (LLVM-TS) and 11 (SPEC2000). Explicitly listed, and thereby most important, are the following combinations of rejection reasons:

- 🟧 Non-affine conditions in loop control flow in conjunction with non-affine accesses with a rejection rate between 12.1% and 17.8%.
- ▦ Non-affine conditions in loop control flow together with varying base pointers accounting for 5.2% to 7.6% of the rejected regions.
- ▤ Non-affine conditions in loop control flow in addition to varying base pointers and function calls dismiss 5.4% to 8.2% of all regions.
- ▥ Non-affine conditions in conjunction with function calls and non-affine accesses invalidate 9.9% of the regions found in the LLVM Test Suite (LLVM-TS).

Question 4. Our last evaluation question is concerned with the number of distinct rejection reasons per region. To this end, we plotted the amount of rejected regions with regards to the number of distinct causes that were responsible for their dismissal in Figure 3.22. All combinations of rejection reasons that did occur less than 1000 times are uniformly colored 🟦 at the top of the bars. Combinations that were present in Figure 3.21, and therefore in the list above, did retain their coloring and are stacked at the bottom of the bars. The remaining combinations are randomly but consistently colored and placed in-between.

The distribution for all four benchmark suites is centered around three distinct rejection reasons. Thus, most regions are dismissed because of a relative low number of reasons. There are only very few regions that contained eight or more distinct problems and exactly one, in SPEC2006, with eleven. Instead, the number of different combinations is significant and most of them dismiss less than two percent of the regions (ref. Figure 3.21). This fragmentation can be seen for SPEC2017 in Figure 3.22. Even less significant combinations, which are not explicitly listed in Figure 3.21, dismiss more than 1000 regions and are therefore uniquely colored (not 🟦).

Conclusion There are several conclusions we can draw from the two studies on the rejection reasons of LLVM/Polly. First, there are several rejection causes, namely 5 to 7 in Table 3.16, that our applicability extensions completely eliminated. Second, there are rejection causes, especially aliasing and non-affine expressions, that were mitigated by our work but not completely eliminated. Third, advanced applicability can expose new problems that have to be handled, e.g., control flow conditions involving multiple pointers (MP in Table 3.19) often imply aliasing pointers and thereby contradic the assumption used to justify runtime alias checks (ref. Section 4.1). Lastly, we need to improve and incorporate extensions to deal with function calls (ref. Section 6.1), as well as expressions currently classified as non-affine (ref. Section 6.4). These two rejection reasons are, alone and in combination, the most frequent.

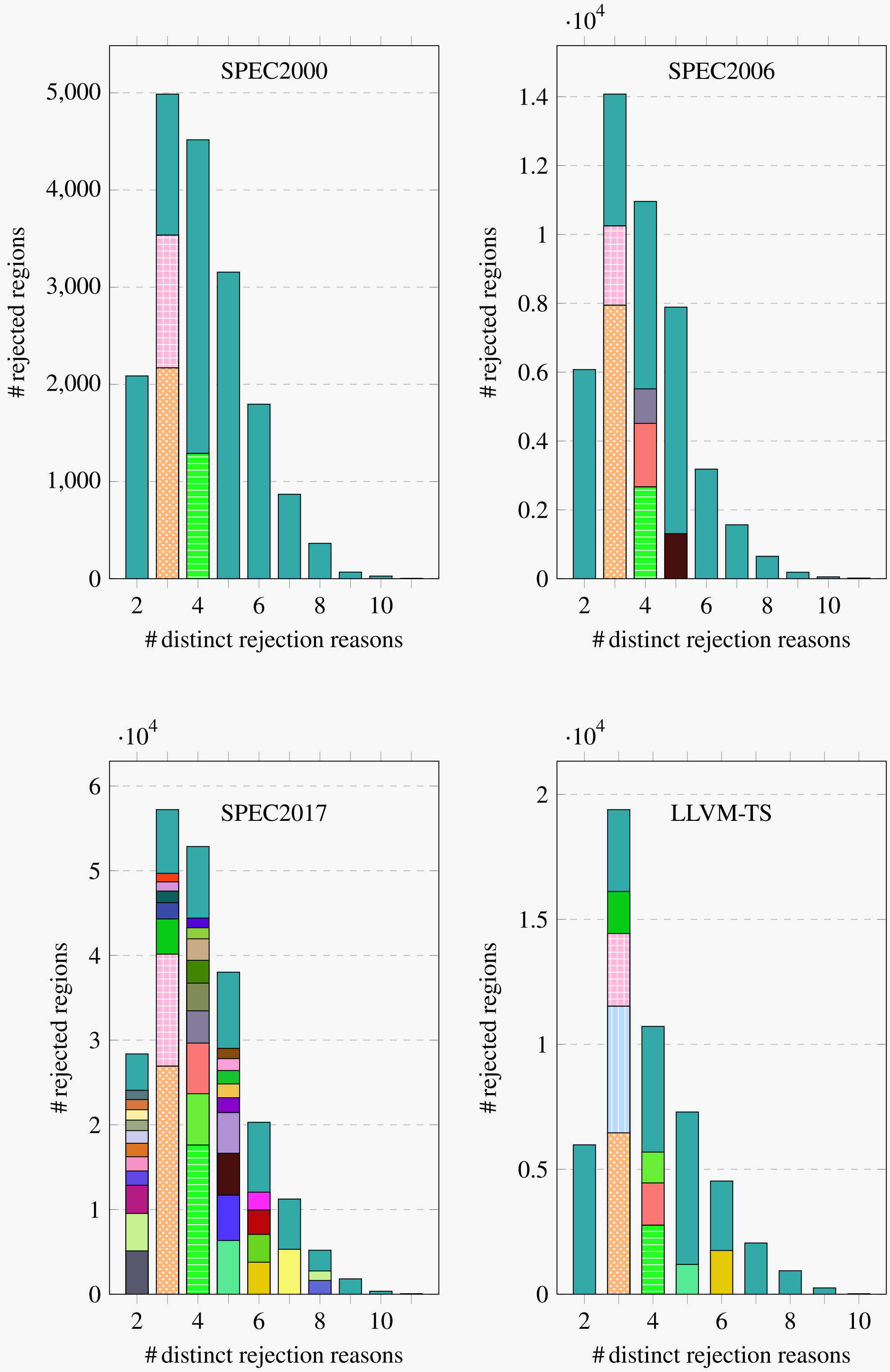


Figure 3.22: Distribution of rejection reason combinations (ref. Table 3.19). Significant combinations (> 2% in any test suite) match the color used in Figure 3.21, other combination are colored randomly but consistent in all four plots. Combinations that cause less than 1000 regions to be rejected are colored uniformly 🟦 at the top of the bars.

3.3 Control Flow Representation

Polyhedral-model-based tools historically required structured control flow [Bas+03; Fea92a]. Thus, a series of nested for loops and conditionals, which additionally adhere to severe syntactic requirements. The use of **`goto`**, **`continue`**, **`break`**, and **`switch`** statements as well as while loops[9] was generally not supported [PCL11]. With the introduction of the polyhedral extraction tool [VG12] various limitations, including **`continue`**, **`break`**, and **`while`**, have been overcome. Though the application is limited to high-level languages that are converted to an abstract syntax tree (AST) and for which syntactic requirements are still in place (ref. Section 6.4). While most restrictions might not limit expert users that want to apply polyhedral optimizations on properly written program parts, they do hinder automatic polyhedral optimization integrated in a general purpose optimization pipeline as performed by LLVM/Polly, Graphite [Pop+06] in GCC or the polyhedral pass in the IBM XL compiler [Bon+10]. The reasons are manifold, reaching from coding styles, over differences in the input representation, to the effects of other code canonicalizations/transformations which are potentially oblivious to the user. The input and output of low-level polyhedral optimizers is not an abstract syntax tree (AST) but instead a low-level control flow graph (CFG) with basic blocks connected though conditional and unconditional edges/branches. In this representation, different syntactic loop forms are indistinguishable, and it is not necessarily possible to determine the presence (or absence) of certain syntactic constructs in the input program. However, one can determine if a part of the CFG is expressible as a sequence of perfectly nested loops and conditionals, hence without the need for **`goto`**, **`break`** or **`continue`** statements. This is the case if each loop has a unique exiting and back edge. Additionally, each conditional branch has to start a single-entry single-exit (SESE) region. Though, especially in advanced stages of the compilation pipeline, canonicalization and optimization passes might (even actively) break these properties as they are restrictive and not necessarily useful. In addition, there are various inputs that do not have the required properties to begin with begin with. It is consequently worthwhile to eliminate these left-over syntactic requirements that stem from the handling of (structured) ASTs in order to improve applicability and robustness on all kinds of inputs.

> This section describes the control flow generation algorithm we integrated[a] into LLVM/Polly to eliminate syntactic and structural requirements on the input control flow graph (CFG). A slightly more general version of this algorithm is also an integral part of our dedicated polyhedral value analysis [DH17b] described in Section 6.4.
>
> [a] There are outstanding patches that have not been merged yet, see LLVM bugs 35465, 37576, and 35434.

[9] The term “while loop” does in the polyhedral community generally refer to a loop with an unknown trip count and not the syntactic statement.

In the remainder of this section, we describe a slightly generalized version of the domain generation algorithm we integrated into LLVM/Polly. It is free of syntactical requirements on the structure of the input CFG except, that the SCoP is built for a SESE region and that there is no irreducible control flow [Hav97] in the represented parts. Note that these two remaining requirements are benign as Table 3.19 clearly shows that reducibility is practically always given[10] and Section 3.3.3 describes how multi-entry-multi-exit regions can be (virtually) reduced to SESE regions. We also support all non-indirect control flow constructs that do not involve exceptions, thus conditional and unconditional branches, **`switch`** statements, as well as **`return`** statements. The domain generation allows for precise modeling as long as the control flow conditions of conditional branches and **`switch`** statements are static-affine. In the presence of non-affine conditions control flow approximations, as described in Section 3.4.2, are employed. To simplify the algorithms we omitted the explicit handling of switch instructions. However, they can be be seen as a cascade of conditionals and `goto`s, thus unconditional branches, as illustrated in Figure 3.23. In addition to the eliminated syntactic and structural requirements, our modeling does not need a dedicated loop trip count analysis. Though, it is important to note that LLVM/Polly still relies on an external, non-polyhedral-model-based analysis to identify affine expressions [BWZ94; PCS05], e.g., for control flow conditions. Consequently, the limitations imposed by this analysis are still decreasing the applicability, especially in the presence of complex, e.g., piecewise defined, iteration counters. To eliminate this limitation we propose a dedicated polyhedral value analysis in Section 6.4 which in turn relies on the domain generation presented here.

```
    switch(cond) {
      case c0:
A:      StmtA();
        break;
      case c1:
        StmtB();
        // fall-through
      case c2:
        StmtC();
        break;
      default:
        StmtD();
        goto A;
    }
```

(a) Original code version using `switch`, `break`, `goto` and a fall-through case.

```
    if (cond == c0) {
A:    StmtA();

    } else if (cond == c1) {
      StmtB();
      goto FT;
    } else if (cond == c2) {
FT:   StmtC();

    } else {
      StmtD();
      goto A;
    }
```

(b) Transformed code version that uses only conditionals and `goto`s.

Figure 3.23: Syntactic conversion of a `switch` statement (left) to a cascade of conditionals and `goto`s (right). Fall-through cases get appended with a `goto` statement to the following case, others do not need to be changed.

[10] The irreducible control flow rejection reason (IR) shown in Table 3.19 has overall only 87 occurrences.

3.3.1 Iteration Domain Generation

To generate iteration domains for a reducible sub graph of a CFG we will traverse it twice in reverse post order. The first traversal callback is illustrated in Algorithm 3.24. The procedure propagates control conditions to all statically affine successor blocks. Non-affine control regions (ref. Section 3.4.2) are treated as a single, atomic block (or statement). Thus, we only need to compute the domain of the entry block which is then used for all blocks inside the non-affine control region. Iteration domains are collected in the `DomainMap`. It is initialized with a universe (=unconstrained) domain of appropriate dimensionality for the entry block of the SESE region that bounds the SCoP. The iteration domains of all other blocks are initially empty. After the first CFG traversal all iteration domains have been initialized with straight line control flow constraints. Loop dimensions are still unbounded at this point. While the reverse post order traversal ensures that predecessors have already been visited, loop latch blocks are obviously excluded. The circular dependence between the domain constraints of a loop header and a loop latch block cannot be resolved in one step. Instead, a second traversal is performed using the callback procedure illustrated in Algorithm 3.25. The domain constraints of the predecessor blocks are unified and then combined with the initial domain built in the first traversal. In addition, loop exiting constraints are specialized for the iteration that will eventually exit the loop. This cannot be done in the first step because it requires control flow information from all loop exits.

3.3.2 Related Work

Enhancements for the control flow representation in the polyhedral model follow two distinct paths. In this section we present a way to remedy implementation limitations (ref. Table 3.17) that prohibited certain syntactic constructs, e.g., **`switch`** statements. In addition, we provide a iteration domain generation algorithm that eliminates representation limitations caused by all varieties of reducible, static-affine but unstructured control flow. The closest to our work is the polyhedral extraction tool by Verdoolaege and Grosser [VG12]. As it is syntax guided and works with an AST, it is not well suited for low-level polyhedral tools. In addition, the support for **`break`** and **`continue`** statements might not be sufficient if the programmer, or canonicalization passes, introduced **`goto`** based control flow, e.g., to exit multiple loops at once. An alternative kind of control flow representation improvements is concerned with dynamic or non-affine conditions. Different approximation schemes are known for the general case [Ben+10; GC95; MDH16] as well as for special loop forms [ZKC18]. Details on these techniques are provided in Section 3.4.

Kumar and Pop [KP16] showed that a different approach to determine and extend analyzed regions can significantly improve the profitability of SCoPs and the compile time. Since the control flow representation capabilities do limit SCoP detection, it is worthwhile to revisit some initial preconditions on SCoPs, especially the need for SESE regions (ref. Section 3.3.3), in the future.

```
procedure PROPAGATEBRANCHCONSTRAINTS(Block : BB, DomainMap : BB → ISet)
  Domain ← DomainMap[Block]
```

Block that are part of a non-affine control region (ref. Section 3.4.2) do not have an associated domain.

```
  if isEmpty(Domain) then return
```

Determine the set of statically affine *control flow successors which either contains the actual successor blocks or it is the end of the non-affine control region that hides the emanating control flow.*

```
  ExitCondition ← getExitControlFlowCondition(Block)      ⊆ ISet → {true,false}
  if isStaticAffine(ExitCondition) then
    SuccessorBlocks ← Block.getSuccessors()
  else
    SuccessorBlocks ← {Block.getSESE().getExit()}
  end if
```

Compute the domain constraints and propagate them to the successor(s). Since a successor might be surrounded by different loops, the domains have to be adjusted (ref. Algorithm A.1 on Page 201).

```
  switch SuccessorBlocks.size() do
    case 0: return                                    skip return instructions
    case 1:                               handle unconditional branch instructions
```

Skip the block if it is outside the SCoP or reached via a loop back edge.

```
      if SCoP.contains(SuccBB) and not SuccBB.dominates(Block) then
        DomainMap[SuccBB] ← ADJUSTDIMENSIONS(Domain, Block,
                                             SuccessorBlocks[0], false)
      end if
      return
    default:                                handle conditional branch instructions
      assume SuccessorBlocks.size() == 2
      SuccBB0, SuccBB1 ← SuccessorBlocks[0], SuccessorBlocks[1]
```

Handle blocks inside the SCoP which are not reached via a loop back edge.

```
      if SCoP.contains(SuccBB0) and not SuccBB0.dominates(Block) then
        TrueCondition ← Domain ∩ getTrueDomain(ExitCondition)
        DomainMap[SuccBB0] ← DomainMap[SuccBB0] ∪
                        ADJUSTDIMENSIONS(TrueCondition, Block, SuccBB0, false)
      end if
      if SCoP.contains(SuccBB1) and not SuccBB1.dominates(Block) then
        FalseCondition ← Domain ∩ getFalseDomain(ExitCondition)
        DomainMap[SuccBB1] ← DomainMap[SuccBB1] ∪
                        ADJUSTDIMENSIONS(FalseCondition, Block, SuccBB1, false)
      end if
  end switch
```

Algorithm 3.24: First domain generation traversal callback: Forward propagation of control flow constraints to all statically affine control flow successor blocks. As back edges are skipped, only straight line control flow constrains are collected.

```
procedure FINALIZEITERATIONDOMAINS(Block : BB, DomainMap : BB → ISet)
    We skip all blocks with an empty initial domain as they are either dead or in a non-affine control region.
    InitialDomain ← DomainMap[Block]
    if isEmpty(InitialDomain) then return
    If the SCoP entry block is not also a loop header we can skip it as well.
    PredBlocks = getBlockPredecessorsInScop(Block);
    if PredBlocks.isEmpty() then return
    Collect the union of all predecessor domains and intersect it with the initial domain to obtain an updated iteration domain.
    IncomingDomain ← getEmptyDomainForBlock(Block);
    for PredBB in PredBlocks do
        PredDomain ← ADJUSTDIMENSIONS(DomainMap[PredBB], PredBB, Block, false)
        IncomingDomain ← IncomingDomain ∪ PredDomain
    end for
    DomainMap[Block] ← InitialDomain ∩ IncomingDomain;
    If the block is not a loop header the updated domain is at this point final.
    if not isLoopHeaderInSCoP(Block) then return
    A loop header is executed until control flow leaves the loop via an exiting edge. To this end, we unify all exiting conditions of the loop and remove these iterations, as well as all later ones, from the header domain.
    AllExitIterations ← getEmptyDomainForBlock(Block);
    Loop ← getSurroundingLoopInSCoP(Block)
    for (ExitingBB, OutsideBB) in Loop.getExitingEdges() do
        Get the constrains under which the last execution of ExitingBB transfers control to OutsideBB.
        EdgeConstraints ← getEdgeConstraints(ExitingBB, OutsideBB)
        ExitConstraints ← ADJUSTDIMENSIONS(EdgeConstraints, ExitingBB,
                                           OutsideBB, /* Last */ true)
        AllExitIterations ← AllExitIterations ∪ ExitConstraints
    end for
    While exit iterations are still part of the loop, all later iterations are not. To compute them we use a forward relation that maps an iteration at a specific loop depth to all later iterations in that depth, e.g., for loop depth 3 the forward relation looks like $\{[i,j,k] \to [i,j,k'] : k < k'\}$ (see [VG12]).
    ForwardRelation ← getForwardRelation(Loop.getDepth())
    AllExitedIterations ← ForwardRelation(AllExitIterations)
    DomainMap[Block] ← DomainMap[Block] ∩ ¬ AllExitedIterations
```

Algorithm 3.25: Second domain generation traversal callback: Propagation of domain constraints and loop exit conditions across iterations and out of loops.

3.3.3 Multi-Entry Multi-Exit SCoPs

To be applicable for polyhedral-model-based analysis and optimization the input has to form a single-entry single-exit (SESE) region. While this is the case for loop nests that only contain structured control statements, it does not necessarily hold for complex inputs or after transformations, e.g., jump-threading, have altered the CFG. Since the SESE region restriction does not stem from limitations of the polyhedral model but is merely enforced to simplify the analysis and transformation, it can be dropped altogether. This is especially interesting because the modeling of multi-entry multi-exit regions can be mapped to the modeling of SESE regions as illustrated in Figure 3.26. The virtual nodes "single entry" and "single exit" are introduced to collect all incoming and outgoing edges of the SCoP. The control transfer of both nodes can be realized with a switch-like construct which can then be represented using conditionals and gotos as illustrated in Figure 3.23. The input edge conditions shown in Figure 3.26 can only depend on parameters of the SCoP and are consequently statically affine. The output edge conditions can however be dynamic or non-affine but this does not prevent the application of the presented scheme.

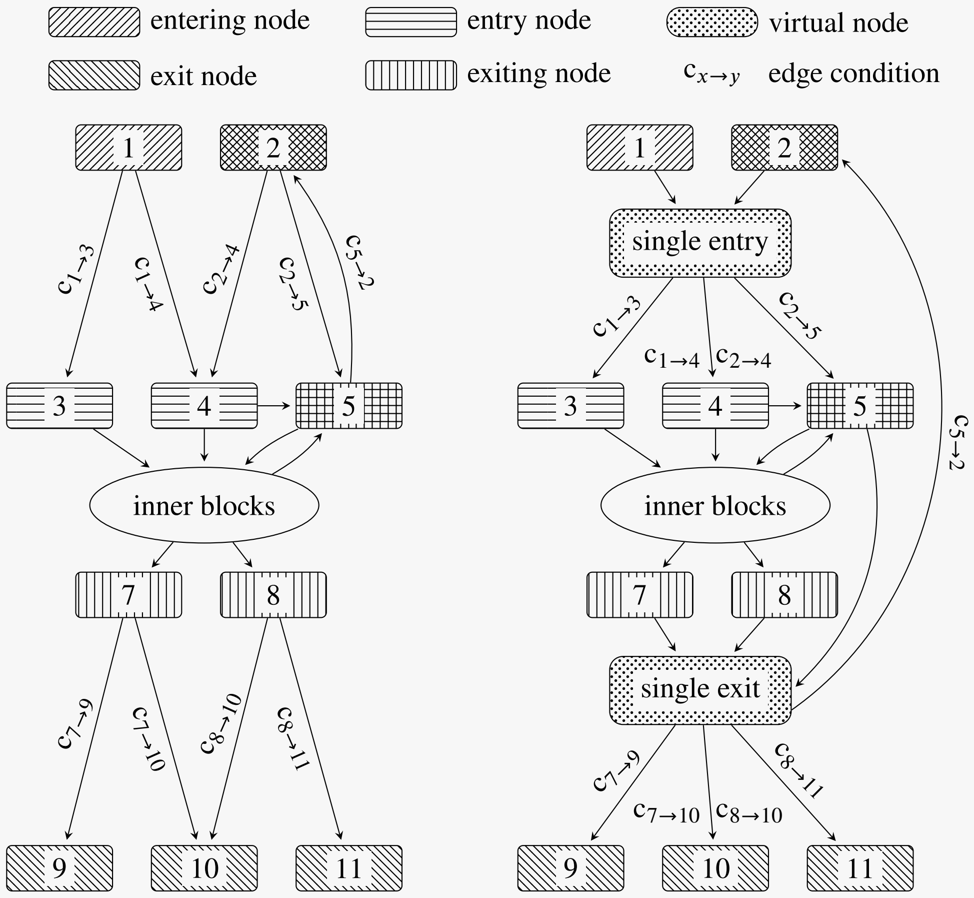


Figure 3.26: Encoding of a multi-entry multi-exit region (MEME) (left) as a single-entry single-exit (SESE) region (right) with two virtual blocks ("single entry" and "single exit"). In the polyhedral representation these blocks bundle all control transfer edges into and out-of the MEME region to create a virtual SESE region.

3.3.4 Evaluation

To evaluate the extended iteration domain generation, we compared our default configuration (ref. Section 2.3), which allows unstructured control flow, to one that does not. Thus, the latter enforces a single loop exit and back edge, as well as a SESE region around each conditional. To ease the interpretation of the obtained metric scores, we show them in Table 3.27 relative to our baseline results provided in Figure 3.3 and Table 3.4 on Page 26 and 27. Hence, percentages less than 100% represent decreased metric scores due to rejected unstructured control flow.

In the current setting, and for most benchmark suites, non-structured control flow has only little impact on the number of SCoPs, which is only down by 1.7% to 15.5%. Similarly, the monotone applicability metrics (ref. Section 3.1.1) are decreased by at most 14.7%. One reason for this is the lacking support for piecewise defined expressions that generally depend on control flow (ref. Section 6.4). To verify this hypothesis we determined the probability of a region to contain non-affine branches, depending on the presence of unstructured control flow. Overall, 27.6% of all regions did contain non-affine branches as as a rejection reason and 11.7% did contain at least one loop with multiple exits. The probability of a region to contain a non-affine branch if it already contained an unstructured loop was 82.9% while the probability was only 20.3% if no unstructured loop was present. Even though not all of these cases will benefit from a more powerful analysis to determine affine expressions (ref. Section 6.4), it is plausible that such an analysis will improve the beneficial effect of our extended iteration domain generation.

Unstructured Control Flow As Rejection Reason We also considered structured control flow as a SCoP requirement and counted the number of SESE regions that violated it. Since this changed the set of valid regions which the SCoP detection tries to expand, the numbers are not fully compatible with the rejection reason results shown in Table 3.19. Though, the order and magnitudes did not significantly change. In this experiment, unstructured control flow due to multiple loop exits is the seventh's most often occurring rejection reason. It is trailed by unstructured conditionals that do not form a SESE region and potentially aliasing pointers.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	94.5%	97.0%	84.5%	98.3%
# depth 1 SCoPs	93.9%	97.4%	83.6%	97.2%
# depth 2 SCoPs	100.0%	93.9%	93.6%	100.0%
# depth 3 SCoPs	100.0%	100.0%	100.0%	100.0%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	95.2%	96.6%	85.3%	98.9%
C_1 score	100.0%	94.3%	94.4%	100.0%

Table 3.27: Applicability results in various metrics (ref. Section 3.1) when unstructured control flow was forbidden. The percentages are shown relative to the baseline results provided in Figure 3.3 and Table 3.4 on Page 26 and 27.

3.4 Approximative Polyhedral Program Representation

Two major strengths of a polyhedral-model-based program representation are iteration-wise control flow information and precise memory access descriptions. Though, both are only possible if the control flow conditions and access relations can be represented with statically affine expressions. If that is not the case, polyhedral tools commonly give up on the analyzed code region. As an alternative, specialized modeling techniques can derive conditionally valid representations (ref. Section 3.5), or program parts that do not comply with the strict requirements can be excluded completely (ref. Section 3.7). However, static modeling techniques will always be limited, and it is often impossible to find static conditions that prevent only the execution of certain program parts. Instead, an approximative polyhedral representation is often the only practical alternative, especially for applications which do not only use polyhedral information [Atz+16; Cha+16], or those that always require a polyhedral representation [MDH16].

Approximations allow polyhedral representation and optimization of partially statically affine programs [Ben+10; MDH16; PC10; ZKC18]. While sufficient approximation allows to represent any program, the precision of the representation, and thereby the benefit of having one, is sensitive to the amount and kind of approximations employed. To enable schedule optimizations in the presence of approximations it is often necessary to utilize non-polyhedral analyses or alternatively domain knowledge (ref. Section 5.1.2). While the latter is available if domain specific languages are used [Bag+15; MDH16; MVB15], the former is especially interesting if the polyhedral tool is integrated into a compiler framework that provides non-polyhedral analyses.

3.4.1 Non-Affine Accesses

Prior to our work, LLVM/Polly was able to approximate non-affine or dynamic memory accesses. In the polyhedral representation that was built, these accesses *may* read or write the entire array. While all approximations can induce spurious dependences [Ben+10; CBF95], it is less problematic for non-affine read accesses. Especially if the array is read-only, approximations will not prohibit optimizations. Thus, the loop in Figure 3.28 can be optimized, e.g, parallelized, despite the non-affine accesses.

```
Out = malloc(...);
for (i = 0; i < N; i++)
  Out[i] = In[i*i-1] + In[i*i+1];
```

Figure 3.28: Non-affine accesses to a read-only array that do not prevent optimizations.

This section describes approximations that allow polyhedral representation of partially polyhedral code. Some approximations were part of our work on *Input Space Splitting for OpenCL* [MDH16], while others were developed independently. Note that not all approximations are part of LLVM/Polly but some are only available in the `approximation` branch of our research prototype.

```
    for (i = 0; i < N; i++)
S:    A[i*N] = ..
    for (j = 0; j < N; j++)
P:    A[(j+N)*N] = ..
```

Figure 3.29: Non-affine write accesses that do not prevent loop fusion if a symbolic range analysis is employed.

To statically tighten approximative values, we utilize constant range information computed by the SCALAR EVOLUTION analysis [BWZ94; PCS05] (ref. Section 6.4). Since many expressions cannot be tightly bounded by constant ranges, we additionally implemented a simple, symbolic interval analysis that is also based on SCALAR EVOLUTION. While the constant range approximation is integrated into LLVM/POLLY, the symbolic ranges are only available in the `approximation` branch of our research prototype. Note that these efforts are less sophisticated than other symbolic interval analyses [BE95; RR00]. We do especially not check for dependences explicitly, but only limit the possible effects of a non-affine access with the range information. For the example in Figure 3.29, our symbolic range analysis generates the access approximations shown in Formula 3. Note that the multiplication $N * N$ is considered to be a parameter, and consequently affine, as it only depends on other parameters.

$$\begin{aligned} f_{w_\mathsf{S}} &= \{\, i \rightarrow \mathtt{A}(o) \mid 0 \leq o < (N * N) - N \,\} \\ f_{w_\mathsf{P}} &= \{\, j \rightarrow \mathtt{A}(o) \mid (N * N) \leq o < 2 * (N * N) - N \,\} \end{aligned} \qquad \textbf{(3)}$$

The polyhedral dependence analysis [Fea91] implemented in ISL [Ver10] is able to utilize these approximated access functions and potentially derive less spurious dependences. However, we still require the array base pointers to be fixed and statically known.

3.4.2 Non-Affine Control Regions

Non-affine control regions [MDH16] allow to built a polyhedral representation in the presence of non-affine or dynamic control flow conditions, later often denoted as non-affine branches. The regions approximate the effects of the non-representable condition by enclosing the subgraph of the control flow graph (CFG) that is control dependent on it. Since control flow information inside the region might be dependent on the non-affine or dynamic condition, we cannot express it in terms of one execution of the static control part (SCoP). Thus, in the polyhedral representation, a non-affine control region is treated as a single atomic statement, regardless of the contained control flow paths. Since the execution of accesses inside the region can also be control dependent on the non-affine or dynamic branch, they are generally assumed to be potentially happening. Hence, contained accesses are, similar to approximated non-affine accesses, modeled as *may* accesses. Only if an access is executed regardless of the non-expressible condition, it can be represented as definitively happening, thus as a *must* access. The distinction is important to determine the last write of a memory location and thereby to distinguish between *value-based dependences* and *memory-based dependences* [Fea91; MAL93; PW93].

```
    for (i = 0; i < N; i++)
      if (B[i] > 0.0)
S:      A[i] = A[i] + B[i];
```

(a) Original code version featuring a dynamic condition to guard a memory update.

```
    #pragma parallel
    for (i = 0; i < N; i++)
      if (B[i] > 0.0)
S:      A[i] = A[i] + B[i];
```

(b) Symbolically optimized code version with a parallelized loop.

Figure 3.30: Example program (left) featuring a non-affine conditional in the innermost loop. A non-affine control region will encapsulate the conditional and the statement S. Affine memory accesses allow to analyze and parallelize the loop (right).

Figure 3.30a shows a simplified version of an innermost loop that can be found in various real-world code. The condition B[i] > 0.0 is dynamic as it depends on the content of an iteration-dependent memory cell. However, the access in the condition as well as the guarded accesses are all affine and can be precisely represented. Only the control flow is not statically know, thus it is impossible to predict for which loop iterations statement S is executed. To create a polyhedral representation of this loop, a non-affine control region is used to encapsulate the conditional. We depict these regions as dashed rectangles. The associated polyhedral statement will include all four memory accesses, hence three reads and one write. Note that the accesses to the A array and the second read access to the B array have to be represented as may accesses while the first read of B is known to be executed in every iteration. This distinction is important for dependence calculation but also for transformations like invariant load hoisting (ref. Section 3.6) and expression propagation (ref. Section 5.3). Since the access functions in our example are precise and only control flow information is approximated, we can prove the absence of dependences. Consequently, transformations, e.g., the parallelization shown in Figure 3.30b, can be performed.

```
     #pragma parallel
     for (i = 0; i < N; i++)
       #pragma parallel
       for (j = 0; j < M; j++)
         for (k = 0; k < i * j; k++)
           if (B[j][i] > 0.0)
S:           A[k] = A[k] + B[j][i];
```

(a) Original code version featuring strided accesses to the B array.

```
     #pragma parallel
     for (j = 0; j < M; j++)
       #pragma parallel
       for (i = 0; i < N; i++)
         for (k = 0; k < i * j; k++)
           if (B[j][i] > 0.0)
S:           A[k] = A[k] + B[j][i];
```

(b) Optimized code version after interchange of the two outermost loops.

Figure 3.31: Example program (left) featuring a non-affine loop bound in the innermost loop. A non-affine control region will encapsulate that loop including the conditional and the statement S. The domain knowledge (parallel annotations) contained in the program does however allow to perform a beneficial loop interchange of the two represented (outermost) loops (right).

In Figure 3.31 a more elaborate use case for non-affine control regions is shown. In this example the entire innermost loop is encapsulated as its upper bound is a non-affine combination of surrounding iteration variables. The iterations of the innermost loop will not be distinguished in the polyhedral representation and all dependent accesses, here `A[k]`, have to be approximated as well. Similar to other non-affine accesses, the ones dependent on a loop that is enclosed in a non-affine control region might access any element of the array. While we can again employ non-polyhedral analyses to improve this worst case assumption, the given example also allows to leverage domain knowledge present in the form of parallel loop annotations. While a detailed discussion on the effect of such parallel annotations as well as other user assumptions will follow in Section 5.1.2, it currently suffices to know that parallel loops cannot carry dependences. It is consequently valid to perform the locality improving loop interchange shown in Figure 3.31b.

3.4.2.1 Limitations & Extensions

```
 for (i = 0; i < N; i++)
  if (B[i] != 0.0)
   for (j = 0; j < M; j++)
S:  A[i][j] /= B[i];
```

Figure 3.32: Non-affine conditional that prevents a precise polyhedral representation of the affine inner loop.

The implementation of non-affine control regions, both in LLVM/Polly and our `approximation` branch, comes with two major limitations. First, we always choose the smallest single-entry single-exit (SESE) region surrounding a problematic branch as non-affine control region. Note that this implementation artifact allows for an easy to test, sufficient condition for definitively executed accesses, hence *must* accesses. If an access dominates the exit block of the SESE region, it is definitively executed. In general however, any multi-entry multi-exit region (ref. Section 3.3.3) could be used as non-affine control region as long as it encloses all blocks control dependent on the non-affine or dynamic branch. The second limitations is the use of statically affine control flow constraints inside the non-affine control region. Since they can be represented with regards to one execution of the SCoP, they could be used to improve the description of contained accesses. For the loop nest in Figure 3.32 we currently loose information on the multi-dimensional memory interpretation and have to rely on the non-polyhedral analysis to determine access bounds. Instead, we could derive the access function shown in Formula 4.

$$f_{w_{\mathrm{S}}} = \{ (i,j) \to (i,o) \mid 0 \leq o < M \} \tag{4}$$

Alternatively, we would like to model the iterations of the inner loop as if the conditional was only guarding the statement `S`. Thus, all loop iterations as well as the access functions could then be represented as shown in Formula 5 even though the write would still be a *may* access.

$$\begin{aligned} \mathcal{D}_{\mathrm{S}} &= \{ (i,j) \mid 0 \leq i < N \wedge 0 \leq j < M \} \\ f_{w_{\mathrm{S}}} &= \{ (i,j) \to (i,j) \} \end{aligned} \tag{5}$$

3.4.3 Function Call Approximation

In Section 3.2 we have identified function calls as a major obstacle for polyhedral tools. To improve the applicability, inlining and precise inter-procedural representation (ref. Section 6.1) can be used. However, especially external function calls but also complicated internal functions are not amenable to these techniques. Instead, we rely on static code analysis and programmer annotations to represent a call as a summary of all possible side-effects. The different summary effects known to LLVM are shown in Table 3.33. The last two columns indicate if it is possible to approximate these effects in LLVM/Polly and our research prototype (RP).

Summary Effect[a]	**Description**	**Polly**	**RP**
no memory access	No memory accesses, e.g., "const" in C/C++.	✓✓[b]	✓✓
read only	Only memory loads, e.g., "pure" in C/C++.	✓	✓
write only	Only memory writes.	✗	✓
argument read only	Only loads of argument pointers.	✓	✓
argument read/write	Only accesses to argument pointers.	✓	✓
inaccessible read/write	Only accesses to memory otherwise not accessible, e.g., internal state of an allocator.	✗	✗
arg. & inac. read/write	Only accesses to argument pointers or memory otherwise not accessible.	✗	✗
unknown	No restrictions on the accesses.	✗	✓

Table 3.33: The different summary effects known to LLVM, their support in the Polly, and in the `approximation` branch of our research prototype (RP). A ✓ indicates approximation support and ✓✓ is used if it is enabled by default.

[a] Summary effects, in LLVM called "mod/ref" behavior, summarize all possible side-effects of a function. It can be queried through the alias analysis interface or expressed as a function attribute.

[b] Already available in Polly prior to this work but incorrect as calls might be dropped.

Function summary effects are used to determine a sound polyhedral approximation for function calls contained in a SCoP. The different summaries can consequently be used to estimate the SCoP's optimization potential ahead of time. Function calls with unknown side-effects or arbitrary write effects (write only) behave like optimization barriers because they can write all arrays that are accessible to the callee. Additionally, the original order between calls with arbitrary read or write effects has to be kept, even if all arrays accessed explicitly in the SCoP are inaccessible to these functions. To this end, we augment their representation with an respective access, thus read or write, to a virtual location. Calls that deal with otherwise inaccessible memory, i.a., the internal state of a custom memory allocator, have to be executed in their original order as well. Though, they do not induce other dependences. Similar to the representation of callees with arbitrary effects, we would need to add accesses to the virtual location to ensure their order. Function calls that read or write argument pointers are represented as non-affine read or write accesses to the argument pointers. Similarly, read only calls are modeled as non-affine reads to all arrays accessible to the callee. Finally there are side-effect free calls which are historically ignored and potentially dropped by POLLY. However, it is important to keep them if their impact on the program termination is not known.

In addition to function summary effects, LLVM allows to distinguish between ordinary, read-only, and write-only pointer arguments. Provided an aliasing free environment (ref. Section 4.1), this information allows to reduce the necessary approximations. In contrast to LLVMPOLLY, the `approximation` branch of our research prototype supports pointer-wise annotations. It also features a simple, intra-procedural analysis to identify function internal base pointers that are consequently not accessible to callees. The analysis is a slight derivation of the live-out access analysis discussed in Section 5.3.6.1.

3.4.3.1 Builtin Function Representation

Programming languages and standard libraries commonly provide a variety of builtin functions such as `memcpy`, `printf` and `strlen` in C/C++. Compilers recognize some of these in order to employ special declarations, also called intrinsics, that provide additional information to analysis passes. Since the side-effects of intrinsics are often limited and known, the call approximation described so far is able to represent them. However, for the most common memory intrinsics, namely `memset`, `memcpy` and `memmove`, we use specialized, semantic-aware modeling. There are two benefits to the default call approximation. First, we obtain a precise representation in the presence of affine arguments, thus there is no approximation for the accessed memory. In addition, we can derive constraints on the parameter if a pointer passed to a memory intrinsic is `NULL`. For such instances the length argument has to be zero whenever the call is executed. Given a call, i.a., `memset(NULL, '\0', e)`, in a statement `S`, we derive parameter constraints to ensure that for all iterations of `S` the polyhedral representation e will be equal to zero:

$$\pi_\rho\big(\mathcal{D}_\mathtt{S} \cap \{(\mathbf{i}) : e = 0\}\big)$$

However, the result is only valid under the assumptions that were needed to create the polyhedral representation e and the domain $\mathcal{D}_\mathtt{S}$. In order to derive independent constraints we have to take these assumptions, in the following denoted as Λ_e and $\Lambda_\mathtt{S}$, into account:

$$\pi_\rho\big(\mathcal{D}_\mathtt{S} \cap \Lambda_\mathsf{e} \cap \Lambda_\mathtt{S} \cap \{(\mathbf{i}) : e = 0\}\big)$$

Similar to constraints derived from value ranges (ref. Section 3.5.3) and user provided assumptions (ref. Section 5.1.2), we can use the above information to simplify and specialize our polyhedral representation. This includes for example profitability functions (ref. Section 3.1.5), iteration domains, and dependence relations.

3.4.4 Related Work

There are several techniques to allow approximative polyhedral optimization in the presence of non-affine or dynamic control flow. Pop and Cohen [PC10] proposed to use a single, atomic approximative statement for OpenMP tasks that contain non-affine control flow or access functions. While the idea is the same as for our non-affine control regions, they treat the approximated region as an actual black box and use only the OpenMP annotations provided by the user to model the side-effects. For arbitrary while loops, Griebl and Collard [GC95] overestimate the potentially executed iteration instances statically and generate dynamic predicates that will guard their execution at runtime. In addition, they describe a predicate to determine the last write to a memory location if the program was transformed into dynamic single assignment form (DSA) [Fea88a]. Similar approaches target distributed memory machines [GL94] and imperfectly nested loop nests [GGL99]. Benabderrahmane et al. [Ben+10] describe the necessary steps to deal with arbitrary loops (and conditionals) throughout the whole polyhedral optimization pipeline. Their approach introduces an explicitly modeled, symbolic upper bound expression (or conditional predicate) for loops (or conditionals) that lack a statically affine trip count (or condition). The loop body (or the conditionally executed code) and the synthetic upper bound (or condition predicate) itself are however (transitively) dependent on all preceding instances of the syntactic upper bound (or condition) expression. Hence, there are dependences that induce a total order on the iterations of the approximated loop (or conditionals). Zhao, Kruse, and Cohen [ZKC18] handle the special case of counted loops with dynamic or non-affine bounds which induce less ordering dependences. Our non-affine control regions are less expressive than these techniques, especially in the case loops are contained in the non-affine control regions. However, our technique is applicable without modifications to the polyhedral pipeline. Thus, the original dependence analysis, scheduling and code generation algorithms can be used.

Speculative execution is an alternative to static approximations [Col95; Pra11]. However, specialized code generation, including a runtime system, is required to detect and recover from misspeculations dynamically. Though, the Apollo framework [Caa+17; Jim+13a; Suk+14; SC16] shows that speculative polyhedral optimization, if combined with appropriate profiling, can be successfully used to optimize partially affine code regions speculatively at runtime.

To improve dependence accuracy for partially polyhedral code, Collard, Barthou, and Feautrier [CBF95] introduce a fuzzy extension to the classical dependence analysis [Fea91] employed by most polyhedral optimizers. Similarly, Baghdadi et al. [Bag+11] propose to combine runtime dependence tests [DYR02; RP95; RRH02] with optimistic polyhedral optimization at compile time. These proposals are alternatives to our approximation schemes and can also be used in combination. The code specialization framework presented Section 3.5 is actually a first step in the direction of a hybrid representation. The statically derived model is an optimistic representation of the input, together with assumptions that have to be verified at runtime.

Polyhedral optimizations often require function calls to be pure [GGL12] or inlined prior to the optimization [Gir+06]. Related inter-procedural approaches to optimize (mainly parallelize) code in the presence of function call do however exist. Triolet, Irigoin, and Feautrier [TIF86] approximate the memory effects of function calls with affine ranges in order to find parallelizable loops or asynchronous executable calls. Their technique was later improved by Creusillet and Irigoin [CI95] in order to determine exact memory effects of functions and also to take "array kills" or overwrites, thus privatization opportunities, into account. These approaches are part of the inter-procedural, source-to-source parallelizer PIPS [IJT91] and consequently in the Par4All [Ami+12] framework that is built on top. Note that our function call approximation is not inter-procedural on its own. Their inter-procedural analyses are more closely related to the inter-procedural SCoP descriptions we present in Section 6.1.

The Pencil intermediate language [Bag+15] allows to provide function effect summaries as affine descriptions of potentially and definitively read and written memory locations. Similarly, Bastoul et al. [Bas+03] rely on expert user to improve conservative analysis results in the presence of function calls. Both can be seen as user-driven specializations of the function summary effects (ref. Table 3.33) used by our function call approximation scheme.

3.4.5 Evaluation

To evaluate the applicability impact of non-affine control regions without enclosed loops, we run two experiments. The results of the first one are shown in Table 3.34a as percentages of the baseline applicability provided in Figure 3.3 and Table 3.4 on Page 26 and 27. In this experiment we allowed SCoPs to contain non-affine control regions but we ignored those that did when we evaluated the applicability metrics (ref. Section 3.1). In the second experiment non-affine control regions were disallowed already during SCoP detection. The results are shown in Table 3.34b. We conducted both studies as the greedy and cost unaware SCoP detection might detect alternative SCoPs if an extension technique is completely disabled. Since the numbers are basically identical here, we know that the use of non-affine control regions was the only suitable choice.

The results of both experiments show that the number of valid SCoPs drops by 19.9% - 35.5%, depending on the benchmark suite. Thus, at least one in five feasible SCoPs contains non-affine control regions. Since the monotone applicability score for $\alpha = 0$ decreased by the same amount, we can conclude that the SCoPs with non-affine control regions were of average size, thus contained the average number of affine loops. The applicability score for $\alpha = 1$ did decrease significantly less for SPEC2000 and significantly more for SPEC2006, compared to the percentage of SCoPs that was still detected. This indicates that the former benchmark suite contains non-affine control regions mostly in smaller SCoPs while the latter features them often in larger SCoPs.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	64.5%	79.7%	65.2%	80.1%
# depth 1 SCoPs	62.2%	84.3%	65.5%	77.4%
# depth 2 SCoPs	77.8%	51.5%	60.6%	81.9%
# depth 3 SCoPs	100.0%	100.0%	71.4%	96.2%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	67.2%	76.3%	64.9%	82.0%
C_1 score	86.7%	54.3%	62.0%	86.0%

(a) Relative results when non-affine control regions without enclosed loops were allowed but SCoPs that contained them are excluded when the metric was evaluated.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	65.5%	80.1%	67.4%	82.3%
# depth 1 SCoPs	63.3%	85.8%	67.9%	79.0%
# depth 2 SCoPs	77.8%	51.5%	61.7%	85.5%
# depth 3 SCoPs	100.0%	100.0%	71.4%	96.2%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	68.0%	77.4%	67.0%	84.3%
C_1 score	86.7%	54.3%	63.0%	88.6%

(b) Relative results when non-affine control regions were disallowed.

Table 3.34: Applicability scores in various metrics (ref. Section 3.1) relative to the baseline numbers presented in Figure 3.3 and Table 3.4 on Page 26 and 27. In the top part, SCoPs with non-affine control regions were recognized but excluded from the metric count. The lower part shows the results when SCoPs were not allowed to contain non-affine control regions.

3.5 Code Specialization

Programs do often not completely comply with the requirements of the polyhedral model. While approximations are a suitable alternative for some cases, an accurate model is often crucial to avoid significant precision loss. Manually operated polyhedral tools [AB15; Bas+03; Ver+13] are meant to be used by experts. They require source code annotations and impose strict syntactic limitations. While they might provide feedback, it is up to the user to comprehend the output and to alter or annotate the code if a problem was encountered. Fully automatic approaches integrated in the compilation chain [GGL12; Pop+06] cannot rely on the user. Without iterative code refinement, the input program has to fully adhere to the syntactic and semantic requirements of the polyhedral model or no transformations are applied. Especially if a user expects polyhedral optimizations, as the code is supposed to be amenable, it is unsatisfactory if they are not performed. However, any semantic mismatch between the input language and the polyhedral model, as well as prior canonicalizations which are oblivious to the user, can prevent optimizations. While it is already hard to comprehend the compiler's choices for classical optimizations, it is even harder to understand drastic differences in the performance if high-level loop optimizations are prevented by subtle semantic differences or seemingly benign changes to the program.

```
double rhs[JMAX][IMAX][5];

for (j = 0; j < grid[0] + 1; j++) {
  for (i = 0; i < grid[1] + 1; i++) {
    for (m = 0; m < 5; m++) {
      rhs[j][i][m] = /* ... */;
      /* ... */
```

Figure 3.35: Simplified excerpt[a] of the `compute_rhs` function in the BT benchmark as provided in the C implementation of the NAS Parallel Benchmarks (NPB) [SJL11].

[a] This Figure was first presented by Doerfert, Grosser, and Hack [DGH17].

To automatically bridge the gap between the requirements of the polyhedral model and programs that almost fulfill them we developed a code specialization framework [Alv+15; Doe+13; DGH17]. It ensures robust applicability of polyhedral optimizations through program versioning with runtime checks synthesized from automatically derived preconditions. The overall design of the system is shown in Figure 3.36. If a semantic mismatch between the input program and the polyhedral model is detected or some (corner case) input would prevent effective and efficient optimization, we derive preconditions that exclude these problematic inputs. We refer to these preconditions as assumptions and denote them as Λ. To represent the code shown in Figure 3.35 in the polyhedral model we will require four distinct assumptions as shown in Figure 3.37a. Note that the

Static code specialization is the basis of our *Optimistic Loop Optimization* work [DGH17]. This section describes the overall framework, assumption simplification and runtime check generation while the actual assumptions are discussed in detail later on. In addition, we contrast static and dynamic specialization. We explored the latter with the SPolly tool [Doe+13] that was integrated in the Sambamba framework [Str+12; Str+15].

constructed polyhedral model is only accurate with regards to these assumptions. Thus, if the assumptions do not hold at runtime, all information provided by the polyhedral representation, which was derived under these assumptions, is potentially void. After all assumptions have been collected, the framework will generalize them to refer only to parameters, or inputs, of the analyzed code region. For our example this would correspond to the code shown in Figure 3.37b. We choose to represent assumptions as symbolic Presburger formulae to allow a native embedding within the polyhedral toolchain. Assumption generalization can therefore be achieved by eliminating all loop iteration counters. Hence, we project the assumptions onto the parameter space, thereby creating generalized versions that hold for *all* loop iterations. This projection is later denoted as $\pi_\rho(\circ)$. Generalization is important as we want to synthesize a single runtime check (RTC) that can be verified prior to the entire analyzed region (or static control part (SCoP)). Before the RTC is created, a simplification step prunes redundant assumptions and simplifies the rest. This will decrease the complexity of the assumptions and the RTC which will consequently reduce the required verification time. A detailed description of assumption simplification is provided in Section 3.5.3. In the final step, the RTC is synthesized using common polyhedral code generation techniques [GVC15] in combination with explicit tracking of integer overflows (ref. Section 3.5.4). The RTC is then used to version the program. It acts as a guard that dynamically chooses between the optimistically optimized and the original program version based on the input values at runtime. This construction is shown in the right most part of Figure 3.36.

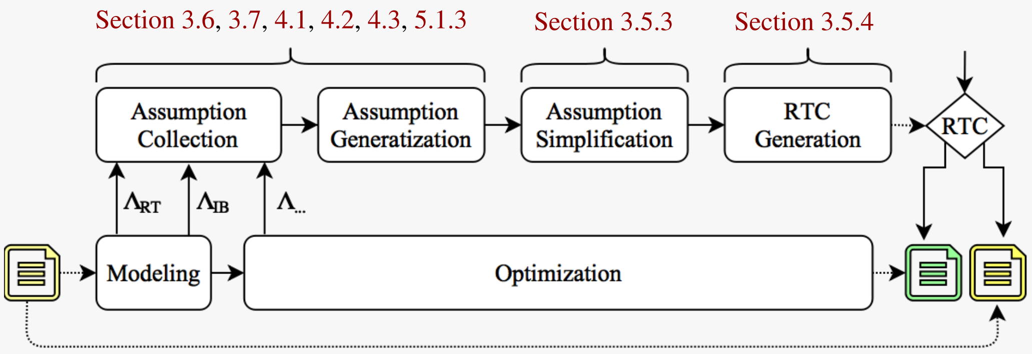


Figure 3.36: Architecture overview[a] of the code specialization framework. Preconditions, denoted as assumptions (Λ), are generated during the modeling and optimization steps. After generalization and simplification they are used to synthesize a runtime check (RTC) that will choose between the original and optimized program version dynamically, based on the inputs to the program.

[a] This Figure was first presented by Doerfert, Grosser, and Hack [DGH17].

The symbolic Presburger formulae that encode optimistically taken assumptions will evaluate to *false* for a given combination of program input values, if it was assumed to not occur. The assumptions can be seen as a sufficient precondition for the generated polyhedral model to be a sound representation of the program. Initially, all inputs are valid and our assumption is therefore the trivial formula *true*. New assumptions, which are added during the SCoP construction, will constrain the set of valid inputs. If the assumptions become too complex or statically evaluate to *false*, the optimization of the SCoP is aborted. Note that assumptions are already collected while the polyhedral representation is still under construction in order to minimize the compile time investment for non-compliant code regions. If the assumption set becomes statically infeasible, or if it causes parts of the input to be excluded and thereby make the SCoP unprofitable (ref. Section 3.1.5), costly steps, like dependence calculation and scheduling, will be skipped.

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;   // no integer overflow (ref. Section 4.2)
for (j = 0; j < grid[0] + 1; j++) {
  assume grid[1] != MAX_VALUE: // no integer overflow (ref. Section 4.2)
  for (i = 0; i < grid[1] + 1; i++) {
    for (m = 0; m < 5; m++) {
      assume j < JMAX && i < IMAX; // no out-of-bounds [Gro+15][DGH17]
      assume &rhs[j][i][m] >= &grid[2] || // no aliasing (ref. Section 4.1)
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
      /* ... */
```

(a) Code shown in Figure 3.35 with explicit assumptions about the program behavior as represented in the polyhedral model.

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE && grid[1] != MAX_VALUE &&
       grid[0] + 1 <= JMAX && grid[1] + 1 <= IMAX &&
       (&rhs[0][0][0] >= &grid[2] ||
        &rhs[grid[0]][grid[1]][5] <= &grid[0]);

for (j = 0; j < grid[0] + 1; j++) {
  for (i = 0; i < grid[1] + 1; i++) {
    for (m = 0; m < 5; m++) {
      rhs[j][i][m] = /* ... */;
      /* ... */
```

(b) Example from Figure 3.37a after generalization of the assumptions to the whole region. This eliminated all occurrences of the loop counters `i`, `j`, and `m`.

Figure 3.37: Code shown in Figure 3.35 with explicit assumptions[a]. The first two assumptions prevent integer overflow (ref. Section 4.2) in the loop bounds, the third one out-of-bound accesses [DGH17; Gro+15], and the last one ensures the absence of overlapping arrays (ref. Section 4.1) and static control (ref. Section 3.6).

[a] This Figure was first presented by Doerfert, Grosser, and Hack [DGH17].

3.5.1 Static Specialization

Static (compile time) specialization allows to model programs that are generally applicable to polyhedral-model-based techniques but might exhibit problematic behavior for corner case inputs. Since all specialization techniques increase the code size, compile time, and execution time on a misspeculation, they have to be used with caution. As static specialization is often performed without knowledge of the (expected) program inputs, it is imperative to be conservative in order to avoid costly misspeculations. Thus, assumptions taken statically should only restrict inputs which trigger behavior that happens rarely, if at all, in average program runs. Promising candidates for static assumptions are binary decisions that depend on the program inputs, e.g., the pointers are aliasing or not, and of which one outcome is far more likely than the other.

In our optimistic loop optimization work [DGH17], we present several important use cases for static specialization. These, and others we added later on, are listed in Table 3.38. Note that a detailed discussion on the first six assumptions is provided in their respective sections while the last one, *always in-bound accesses*, is explained in detail elsewhere [DGH17; Gro+15].

C	**LLVM-IR**	**Polyhedral Model**
Referentially Transparent *Expressions* (Section 3.6)		
not-given	not-given	required
Complex or Unknown Side Effects[a] (Section 6.3)		
possible	possible	not desirable[b]
Aliasing Arrays (Section 4.1)		
possible	possible	impossible
Expression Evaluation Semantics (Section 4.2)		
type-dependent	computation-dependent	evaluation in $\mathbb{Z}$
Signed & Unsigned Values[a] (Section 4.3)		
yes	yes	no
Always Bounded Loops (Section 5.1.3)		
no	no	preferable[c]
Always In-bound Accesses [DGH17; Gro+15]		
sometimes[d]	no	yes

Table 3.38: Semantic differences between C, LLVM-IR and the polyhedral model[e].

[a] Not part of the optimistic loop optimization paper [DGH17].

[b] Unknown or complex side effects can be approximated (ref. Section 3.4) but might prevent optimizations.

[c] Unbounded loops, and therefore unbounded polyhedra, are not necessarily supported by the employed algorithms. If they are, they can cause compile time hazards without real-world benefit.

[d] Out-of-Bound accesses to constant sized multi-dimensional arrays are undefined [C11, Section 6.5.6]. However, symbolically sized multi-dimensional arrays do not have a defined bound that could be violated.

[e] Part of this Table was first presented by Doerfert, Grosser, and Hack [DGH17].

3.5.2 Dynamic Specialization

Dynamic (runtime) specialization is based on information, e.g., actual input values, observed during the program run. Profiling can be seen as the simplest form of dynamic specialization, even if the profile information is only used in subsequent compilations. Dynamic information can prevent the use of static assumptions that are likely to fail at runtime as well as costly polyhedral optimization on never executed program parts. Additionally, dynamic information allows to take assumptions for situations which seem plausible at compile time but are not observed in program runs. If, for example, a parameter value can be predicted fairly accurate, specialization based on the (few) most likely outcome(s) can enable and improve polyhedral optimization.

```
for (i = 0; i < N; i++)
  for (j = 0; j < i * U; j++)
    A[i] = A[i] + B[j];
```

(a) Non-affine loop bound due to the multiplication with the parameter `U`. Specialization with a constant will allow polyhedral optimization.

```
for (i = 0; i < N; i++)
  A[i] = A[i] + B[i];
for (i = 0; i < M; i++)
  B[i] = B[i] * A[j];
```

(b) Affine loop nests that can be fused without remainder loops only if `N` and `M` have the same value at runtime.

Figure 3.39: Examples for which parameter specialization can enable or improve polyhedral optimization.

With SPolly [Doe+13] we introduced an extension of LLVM/Polly in the Sambamba framework [Str+12; Str+15] to allow dynamic code specialization. SPolly employs a profiling version of the loop nest to identify reoccurring parameter values at runtime. For those values, specialized versions are created where (problematic) parameters are replaced by constant values. A dispatcher function checks at runtime whether a specialized version for the current parameter values exists. If so, it is executed, otherwise the original code version is used. For the example shown in Figure 3.39a, specialization of the parameter `U` with a commonly occurring constant makes the multiplication in the loop bound affine. It is therefore representable in the polyhedral model, and thus amenable for all optimizations implemented in LLVM/Polly.

Replacing parameters with constants can additionally lead to superior code as more information is available. Thus, even in the case that the code is already amenable to polyhedral optimization, dynamic specialization can be beneficial if certain parameter valuations are most likely to occur at runtime. For the example in Figure 3.39b, specialization of `N` and `M` allows to perform loop fusion without the need for two remainder loops. While SPolly is able to specialize for specific constant values, it is not able to generate generalized versions, e.g., for the case of equal parameter values. It is important to note that a polyhedral scheduler is capable of generating such specialized code on its own but not necessarily eager to do so. Without knowledge of the expected parameter values specializations have little chance of improving performance while they inevitably increase code size and thereby also compile time.

3.5.3 Assumption Simplification

The kind and number of assumptions necessary for a sound and concise polyhedral representation heavily depends on the programming style and the source language. However, their cost is also determined by their representation and the framework's capabilities to simplify them. Note that the cost is not only limited to the size and complexity of the generated runtime check but also the compile time spend handling the assumptions. To this end, it is crucial to ensure a consistently small and concise representation.

To incorporate assumptions natively into the polyhedral tool we represent them as symbolic Presburger formulae. This representation allows to handle assumptions and generate runtime checks with the algorithms and techniques already available in the polyhedral toolchain. We can especially identify redundant assumptions and find concise, potentially conservatively approximative, formulations for the remaining ones [Ver10; Ver15a]. However, the complexity of the employed techniques is most often exponential in various input characteristics. Additionally, the representation of Presburger formulae itself can become rather complex. To tackle these problems we employ five explicit simplification strategies described in the following.

Constraint Simplification

There are various established simplification techniques for Presburger formulae that can be used to simplify assumptions. These techniques include redundant constraint removal, equality detection [Ver10] and coalescing, thus the combination of multiple constraints into an equivalent set of smaller size [Ver15a]. As a result, redundancies in large assumption sets are reliably eliminated. As an example, consider the constraints $N \geq 0 \land N - 2 > 0 \land (N > 1 \lor N < 0)$ which can be simplified to $N > 2$ without loss of precision.

Program Execution Context

Assumptions are used to constrain the set of valid parameter valuations in order to guarantee a sound polyhedral representation of the input program. Though, if neither the original nor the optimized version will cause observable side effects for some input, there is no need to explicitly choose a version. We will use such configurations in order to prune and simplify assumptions. This is especially useful if all assumptions can be dropped as it will eliminate the need for code versioning. Note that this simplification comes with two requirements. First, it has to take place after the polyhedral representation was completely built, and second, it is only applicable if the entire program is represented in the polyhedral model (ref. Section 3.7 and 6.3). The first requirement is necessary as we might otherwise prune constraints early while later parts of the code perform interesting computations under the pruned conditions. The second one is required because non-represented parts of the input can cause side-effects that need to be preserved.

Impossible or Undefined Behavior

Parameter configurations that are known to not occur or that inevitably trigger undefined behavior define implicit preconditions for the input values. To make these explicit we derive Presburger formulae that describe the possible value ranges of parameters as well as conditions under which undefined behavior would occur. The former are used to prune assumptions that would require parameters to assume values outside their value range. As an example, consider the addition of two boolean parameters that were first promoted to n-bit integers. Since the value range of an (unsigned) boolean is $\{0, 1\}$ we know that the addition cannot cause an integer overflow for any $n > 1$ (ref. Section 4.2). Conditions that would inevitably trigger undefined behavior are exploited similarly.

In addition we allow to enrich the source code with information to prevent assumptions. This is especially useful to avoid out-of-bounds accesses to constant-sized arrays, which are undefined in C/C++ but, due to the relaxed type and memory model, not (generally) in LLVM-IR. We altered the front-end to automatically emit explicit annotations which, similar to domain knowledge (ref. Section 5.1.2), restrict the set of valid inputs. Our annotations will ensure that offsets into constant-sized arrays are in-bounds, thereby eliminating the need to generate in-bounds assumptions [DGH17; Gro+15].

Since this kind of assumption simplification is based on the elimination of constraints that are already implied by know facts, other sources of information can be easily added. An example is given in Section 3.4.3.1 where we describe how calls to known library functions allow to derive constraints on parameters necessary to prevent undefined behavior.

Assumptions vs. Restrictions

For the sake of simplicity we generally describe code specialization with regards to assumptions, thus constraints on the set of valid parameter configurations. However, their inverse, namely restrictions or invalid parameter configurations, can be equivalently used. While this choice does not impact correctness, it can, depending on the implementation of Presburger formulae, significantly impact the representation efficiency. LLVM/POLLY employs ISL [Ver10] for the representation of Presburger sets and ISL uses disjunctive normal form (DNF) as canonical representation. Assumptions are collected by *intersection* (conjuction) while new restrictions are added by computing the *union* (disjunction). For new assumptions that form a single convex polyhedron, the intersection corresponds to a concatenation of all constraints. However, if the assumption is a union of convex polyhedra, distributivity can cause the DNF representation to grow exponentially. In contrast, the representation size for restrictions always grow linearly. To this end, our implementation uses assumptions only to ensure in-bounds memory accesses and restrictions otherwise.

Conservative Over-Approximation

Our Presburger based assumptions are always as precise as the underlying polyhedral representation. If no approximations were employed (ref. Section 3.4), the assumptions will define the weakest preconditions that exclude the unwanted behavior. However, it can be beneficial to strengthen the preconditions through conservative approximations, in order to reduce the complexity of the constraints and consequently of the runtime check that verifies them. As an example, consider the loop in Figure 3.40a. It contains memory accesses to two potentially aliasing arrays (ref. Section 4.1). Due to the non-unit stride, the precise aliasing runtime checks shown in Figure 3.40b are rather complicated. To get to the conservatively simplified version shown in Figure 3.40c, we eliminate all existentially quantified variables from the precise assumption set. Such variables commonly arise in the presence of non-unit strides and modulo expressions and they have shown to complicate assumptions without providing a real world benefit. In this case, the simplification of the runtime check might cause a false positive result if there are less than `(N-1) % 5` elements between the last accessed element of one array and the first accessed element of the other. Note that during our evaluation (ref. Section 3.5.5.1) we have not observed any runtime check failure caused by the elimination of existentially quantified dimensions.

```
for (i = 0; i < N; i += 5) {
  A[i+0] += B[i+0];
  A[i+1] -= B[i+1];
  A[i+2] += B[i+2];
  A[i+3] -= B[i+3];
  A[i+4] += B[i+4];
}
```

(a) Loop with potentially aliasing accesses and a non-unit stride.

```
&B[N+4 - ((N-1) % 5)] <= &A[0] ||
&A[N+4 - ((N-1) % 5)] <= &B[0]
```

(b) Precise but complex alias check that prevent overlapping accesses to `A` and `B`.

```
&B[N+4] <= &A[0] || &A[N+4] <= &B[0]
```

(c) Simplified alias check as a sufficient condition for the absence of overlapping accesses.

Figure 3.40: Precise but complicated runtime alias checks (part 3.40b) and a conservatively simplified version (part 3.40c) for the loop shown in part 3.40a.[a]

[a] This Figure was first presented by Doerfert, Grosser, and Hack [DGH17].

3.5.4 Runtime Check Generation

To verify assumptions dynamically, a runtime check (RTC) is synthesized and used as a guard for the optimized code version. Since the runtime check is an implementation of the assumptions which in turn are required for the polyhedral model to be sound, it is utterly important that all unwanted parameter configurations are filtered out. Thus, if the runtime check succeeds for a given parameter combination, the underlying assumptions have to evaluate to *true* as well. Similar to the assumptions, runtime checks do not need to be complete. Instead, they can conservatively fail if a precise evaluation would be significantly more expensive. This is especially important since assumptions and runtime checks form a circular problem. Assumptions bridge the gap between the semantics of the input language and the polyhedral model. At the same time, runtime checks are supposed to encode the semantics of the model effectively, thus natively, in the target language which can be the same or similar to the input language. In order to resolve this circular dependence, we strengthened the requirements for the runtime check, hence we make it more restrictive with regards to valid inputs, hence we disallow more parameter valuations.

For efficient and sound evaluation of runtime checks we have to deal with two challenges. First, the difference in the expression evaluation semantics used in the polyhedral representation and in the target language (ref. Section 4.2). Second, the observable side-effects that occur during the evaluation. Currently, all but invariant load assumptions (ref. Section 3.6) are conceptually pure[11], thus they do not cause observable side-effects. For invariant load assumptions we first handle the non-pure part explicitly and prior to the runtime check generation as explained in Section 3.6.1.2. The difference in the expression evaluation semantics is resolved in the following.

Assumptions, as the rest of the polyhedral model, are based on Presburger arithmetic and therefore evaluated with *precise* semantics, hence in $\mathbb{Z}$. Programming languages commonly use machine arithmetic, thus *wrapping* semantics evaluated in $\mathbb{Z}/n\mathbb{Z}$, or *error* semantics, which results in undefined behavior if the result of a computation is not represented precisely. Note that neither *wrapping* nor *error* semantics can be used to implement *precise* semantics (ref. Section 4.2). However, if no integer overflow occurs, all three will yield the same results. Since integer overflows are consequently a required condition for a semantic difference, their absence is a sufficient condition for a precise runtime check. To this end, we track overflows during the evaluation of the runtime check in order to conservatively abort if one occurred. Since most modern machines come with built-in overflow detection for arithmetic operations, e.g., an overflow flag that is automatically set by the hardware, overflow tracking is quite efficient (ref. Section 3.5.5.1).

Overflow tracking can be implemented in different ways, as illustrated by the three examples shown in Figure 3.41. In our framework we use the first approach, which sets a boolean variable (here `ov`) to *true* if an overflow occurred. The final guard is the runtime check result and the state of the overflow tracking variable. The second scheme will abort the runtime check evaluation as soon as an overflow occurs. However, this would require various conditional branches which can easily degrade performance. The last alternative uses multiple tracking variables in order to increase the instruction level parallelism (ILP) in exchange for a higher register pressure. Since our evaluation showed that runtime checks are often not too costly (ref. Section 3.5.5.1), we leave it up to further investigation to choose the best runtime check overflow tracking implementation.

[11] This is not true for divisions and modulo operations which can become part of the assumptions if they behave like parameters. In this case, the implementation needs to ensure that the divisor is either non-zero or that the operation would have been executed anyway. Currently, we always generate code to ensure the former.

```
ov = false;
t0 = n + 2;
ov |= checkOverflow();
t1 = m - 1;
ov |= checkOverflow();
rtc = !ov && t1 < t0;
```

(a) Implementation of overflow tracking for runtime checks we integrated into LLVM/POLLY.

```
t0 = n + 2;
if (checkOverflow())
  goto original;
t1 = m - 1;
if (checkOverflow())
  goto original;
rtc = t1 < t0;
```

(b) Tracking implementation for runtime checks with early exits on integer overflows.

```
t0 = n + 2;
ov0 = checkOverflow();
t1 = m - 1;
ov1 = checkOverflow();
ov = ov0 || ov1;
rtc = !ov && t1 < t0;
```

(c) Instruction level parallelism (ILP) concious runtime check overflow tracking with higher register pressure.

Figure 3.41: Different possible schemes to track overflows in the runtime check for the contrived assumption $m - 1 < n + 2$.

3.5.5 Evaluation

To evaluate our code specialization framework we performed two studies that measure the effect on polyhedral applicability. The results are presented in the following but also in parts in the evaluation sections of the extensions that employ assumptions (ref. Table 3.38).

3.5.5.1 Compile Time and Runtime Effect

In our work on *Optimistic Loop Optimization* [DGH17], we determined the efficiency of the generated runtime checks as well as the effect of assumption handling and simplification on the compile time[12]. The execution time cost of the generated runtime checks was always less than 4% of the overall execution time and often vanishingly small. Simplification and modeling choices (ref. Section 4.2) mostly eliminated compile time hazards. The well known LINPACK [Don87] benchmark finished in less than three seconds with simplifications enabled but did not terminate for 500 seconds without. Our evaluation also showed that the misspeculation rates were relatively low: 10.7% of all SCoP executions in SPEC2000, 1.7% in the LLVM Test Suite (LLVM-TS) and 0.3% in the SPEC2006 benchmark suite violated one of our statically derived assumptions at runtime. Interestingly, more than 99.96% of the violations in SPEC2006 were caused by a single SCoP in the 403.gcc benchmark. The remaining ≈ 0.04% are in fact only six misspeculations in ≈ 5.2 million runtime check evaluations. In the case of 403.gcc, as for most other violations, the alias check (ref. Section 4.1) failed. For this particular case, two function arguments with pointer type were identical and therefore aliasing. Coincidentally, the identical pointer arguments did not induce new dependences but due to the conservative nature of the runtime check a fallback onto the original code version was required.

[12] The numbers presented in this paragraph are taken from the evaluation of our work on optimistic loop optimization [DGH17]. In that work we only considered five out of the seven assumptions shown in Table 3.38 and did not evaluate SPEC2017 but the NAS Parallel Benchmarks (NPB) [SJL11] instead.

3.5.5.2 Effective Assumptions

The extensions to classical polyhedral optimization presented in this thesis can take assumptions to ensure statically unknown properties at runtime. While the applicability effect is discussed together with the respective extension, we are also interested in the assumptions they take. To this end we captured the number of SCoPs that were affected by the different kinds of assumptions. It is important to note that, due to our simplification techniques described in Section 3.5.3, not all assumptions must have an effect. They can be statically fulfilled for all defined and meaningful program inputs or already implied by combinations of other assumptions. The number and percentage of feasible and profitable SCoPs for which assumptions were necessary is shown in Table 3.42. The need for assumptions to guard static unknown properties ranges from 1.3% (invariant load assumptions Λ_{RT} in SPEC2006) to 78.8% (no-aliasing assumptions in SPEC2006) of all SCoPs. The differences between the benchmark suites are quite high and reach up to 37.7% for the no-overflow assumptions (Λ_{EE}) and 35.9% for the bounded loop assumptions. Note that the impact of invariant load hoisting (ref. Section 3.6) is far greater than the number of times assumptions were employed by it. This is due to the fact that invariant load assumptions are often either trivially fulfilled or, as discussed in Section 3.5.5.3, cause the SCoP to be infeasible.

Assumption	Λ	Sec.	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
invariant load	Λ_{RT}	3.6	6 (5.5%)	3 (1.3%)	35 (2.9%)	14 (3.3%)
error block	Λ_{EB}	6.3	14 (12.7%)	15 (6.5%)	44 (3.7%)	6 (1.4%)
no-aliasing	Λ_{AA}	4.1	70 (63.6%)	182 (78.8%)	671 (55.9%)	246 (58.9%)
no-overflow	Λ_{EE}	4.2	37 (33.6%)	143 (61.9%)	856 (71.3%)	155 (37.1%)
signedness	Λ_{UR}	4.3	37 (33.6%)	54 (23.4%)	221 (18.4%)	97 (23.2%)
bounded loop	Λ_{BL}	5.1.3	2 (1.8%)	87 (37.7%)	117 (9.8%)	35 (8.4%)
in-bounds	Λ_{IB}	n/a[a]	12 (10.9%)	8 (3.5%)	271 (22.6%)	60 (14.4%)

Table 3.42: Number (and percentage) of *feasible* and *profitable* SCoPs for which assumptions (first two columns) have been taken. Detailed information on the assumption is provided in the Section shown in the third column.

[a] In-bounds assumptions are explained elsewhere [DGH17; Gro+15].

3.5.5.3 Assumption Feasibility

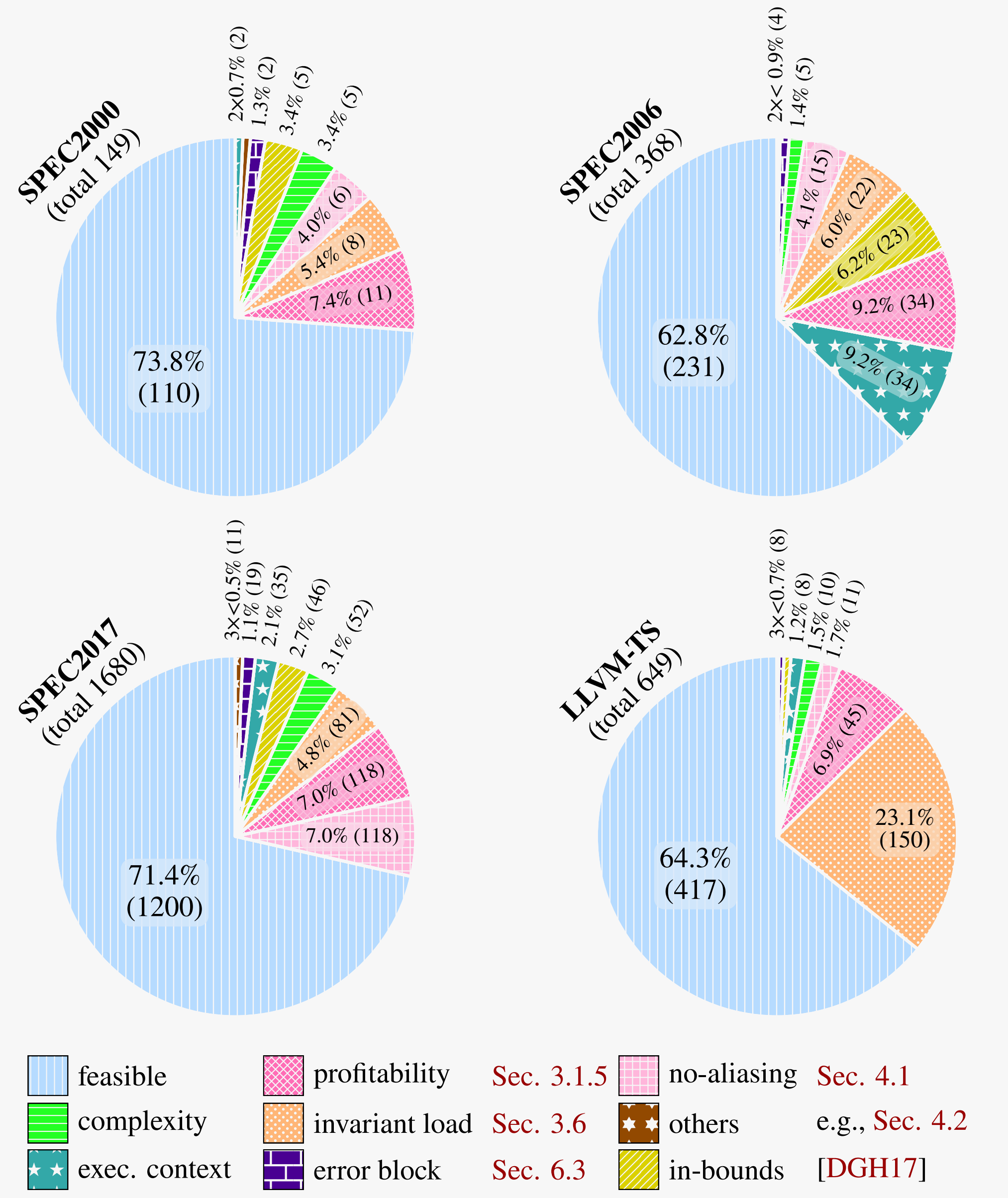


Figure 3.43: Statistics on the feasibility of SCoPs including the reason for infeasibility. If a fatal assumption was taken, thus one that caused the set of valid inputs to become empty, it was counted under the associated assumption name (ref. Table 3.42). If valid inputs existed, but the SCoP's execution context did not contain them, we counted it as "exec. context" ★. We also counted the number of SCoPs that were unprofitable ▩ (ref. Section 3.1.4) and the ones that violated the (arbitrary defined) complexity bounds 🟩.

Effective assumptions reduce the set of valid inputs. It can consequently happen that a SCoP will not perform any meaningful work for all remaining valid inputs. In this case the SCoP becomes infeasible and it is dropped. In Figure 3.43 we show the distribution of reasons that cause SCoPs to become infeasible. While most valid and profitable SESE regions (ref. Section 3.1.4) are also feasible SCoPs ▥ (between 62.8% and 73.8%), more than one quarter is not. The invariant load assumptions ▤ are most often the reason for infeasibility (up to 23.1%). Since only a few feasible SCoPs (up to 5.5%, ref. Table 3.42) actually feature non-trivial invariant load assumptions, it is a strong indicator that the current implementation is too aggressive when it comes to assuming invariance (ref. Section 3.6). No-alias assumptions ▦ eliminate up to 7% of the SCoPs because[13] the alias runtime checks would become too complex (ref. Section 4.1.2). In-bounds assumptions ▨ invalidate up to 6.2% of the SCoPs because the presumed dimension sizes [Gro+15] turn out to be always violated. Due to the late profitability check ▩ (ref. Section 3.1.4) another 6.9% to 9.2% of all otherwise valid SCoPs are dropped. The remaining assumptions, partially summarized as others ★, always invalidate less than 4% of all SCoPs.

3.5.6 Related Work

We discuss works related to the actual assumptions (ref. Table 3.38) in their respective sections.

In contrast to most precondition based program versioning schemes we provide a general framework to take, simplify, and synthesize checks for assumptions expressed as Presburger formulae. We use these assumptions to generate a correct polyhedral program abstraction (ref. Chapter 4), extend the applicability of polyhedral tools (ref. Section 3.6 and 6.3), and to simplify our representation (ref. Section 5.1.3 and [DGH17, Section 4.4]).

Our assumptions are taken to prevent unwanted events at runtime. They can be seen as preconditions, a topic well studied over the years [CH78; Cou+13; Hoe+09]. The use of preconditions to generate runtime checks is common practice, especially in the context of memory bound check elimination for safe languages [BGS00; Gam+08; Nie+09; QHV02; WWM07]. Approaches use various methods to generate preconditions that exclude out-of-bound array accesses, similar to the in-bounds assumptions Λ_{IB} derived within our framework [DGH17; Gro+15].

Several techniques combine polyhedral optimization with runtime version selection, thus the dynamic choice between multiple generated or provided program versions. Approaches find the best candidate with statically generated symbolic selector functions [MDH16; PCL11], or employ dynamic tests that determine dependence [Caa+17; Jim+13a] or timings [BB14; Jim+12]. In contrast to these schemes we only generate one optimistically optimized program version that is executed whenever possible. Though, we have shown that adding more program versions is orthogonal to the assumption-based versioning [MDH16]. One future use for our framework would be to collect optimization specific assumptions as proposed by Baghdadi et al. [Bag+11]. They describe how assumptions could exclude statically derived conditional dependences in order to increase the scheduling freedom. This is similar to the runtime check hoisting discussed

[13] Alias runtime checks have other limitations, e.g., they require affine memory accesses (ref. Section 4.1.3). However, in our test setting (ref. Section 2.3) these limitations cannot occur.

in Section 3.7. Alternatively, we can employ assumptions to allow other optimizations such as vectorization. In the `minimal_dependence_distance` branch of our research prototype we use assumptions to ensure that symbolic dependence distances of innermost loops will have a certain length in order to facilitate vectorized execution (ref. Section 5.1.1).

LLVM [LA04] allows the user and compiler passes to insert flow sensitive boolean facts[14] into the intermediate language. Our framework will automatically use (statically affine) facts to simplify the assumption set (ref. Section 3.5.3 and 5.1.2).

Hoenicke et al. [Hoe+09] use static analysis to identify statements for which there is no valid execution. Similarly, Cousot et al. [Cou+13] compute "necessary preconditions" that, if violated, always break a predefined contract. While we did not try to expose programming errors we could issue warnings, potentially even errors, if certain assumptions, e.g., no-overflow assumptions, are known to be infeasible (ref. Section 3.5.5.3).

[14] In LLVM these facts are actually called "assumptions" and expressed with the `llvm.assume(i1 %c)` intrinsic.

3.6 Invariant Load Hoisting

Statically affine expressions are the corner stone of the polyhedral model. These expressions are pure as they depend only on unknown but fixed parameter values and loop iteration counters. Polyhedral scheduling optimizations can therefore evaluate expressions at a different program point, or loop iteration, with the appropriate modifications to the iteration counters. Statically affine expressions can also be duplicated or they can be computed once and used repeatedly instead.

Expressions that cannot be natively translated to the polyhedral model are either dynamic or non-affine. Each evaluation of a dynamic expressions, e.g., a memory load or function invocation, could result in a different value, regardless of the loop iteration. Additionally, the evaluation can cause side-effects ranging from memory modifications to exceptions. However, not all dynamic expressions show such behavior. There is a certain class of memory loads that, under some conditions, will evaluate to a single unique value during the execution of a code region. Such expressions behave like parameters and can therefore be represented as such.

In this section we discuss *invariant load hoisting*, a technique to represent memory loads as parameters to allow polyhedral representation and optimization of a code region. We show how to derive necessary conditions to guarantee correctness, thus invariance of a load, during the execution of a SCoP. We also discuss the complications that arise from conditionally executed loads of potentially invalid memory locations as well as the impact of approximated, e.g., assumption based, modeling. The described technique is similar to loop-invariant code motion (LICM) [ASU86; Cli95] and partial redundancy elimination (PRE) [BGS99; Cho+98; MR79] as discussed in more detail in Section 3.6.4. Dynamic expressions involving function invocations are handled as part of the inter-procedural SCoP representation presented in Section 6.1.

Our motivating example is shown in Figure 3.44. It is centered around the use of the compound “vector view” data type illustrated in part 3.44a. This vector view provides access to an underlying data array and, similar to Java or Rust array types, also includes the size of the array. The indirection through the data pointer allows for fast shallow copies and for multiple instances to work on a single, shared underlying array. Depending on the pointer and length value, multiple instances might be used to access overlapping or disjoint parts of the data. For now we will assume the latter and postpone the discussion of the overlapping case until Section 4.1.

> The technique described in this section is called *invariant load hoisting* or referential transparent expression generation [DGH17]. It is fully integrated in LLVM/POLLY and enabled in our baseline evaluation configuration (ref. Section 2.3). Though, it was disabled in recent upstream versions.

```
struct VectorView {
  unsigned Length;
  float *Data;
}
```

(a) Compound data type, similar to Java or Rust array types, including an array pointer and size.

```
VectorView &Vec = ...;


for (i = 0; i < Vec.Length; i++)
  Vec.Data[i] = ...;
```

(b) Simple use case for a vector view object. While `Vec.Length` and `Vec.Data` look like unknown but fixed constants they are memory loads that potentially evaluate to different values in each loop iteration.

```
VectorView &Vec = ...;
auto Length = Vec.Length;
auto Data = Vec.Data;
for (i = 0; i < Length; i++)          SCoP
  Data[i] = ...;
```

(c) Example code that illustrates the polyhedral representation which is built under the assumption that the memory loads `Vec.Length` and `Vec.Data` can be hoisted, thus behave like parameters.

Figure 3.44: A simplified example (left) to show how subtle memory accesses are introduced in control flow or array access expressions when using global variables, arrays or compound objects (3.44a). A polyhedral representation can be built if the code can be interpreted like the one shown on the right.

In Figure 3.44b, a simple use case for a vector view is shown. While the code looks amenable to polyhedral techniques it is important to note that both `Vec.Length` as well as `Vec.Data` are actually memory loads of the respective member field, hence dynamic expressions. Situations like this commonly arise if global variables, arrays or indirections are used since they will always be (first) translated to memory loads. Polyhedral techniques can only create a precise representation of such inputs if each memory load actually behaves like a parameter. Thus, loads have to evaluate to the same, statically unknown but fixed constant value throughout the entire execution of the SCoP. If an assumption can be found under which this is the case, hence the memory loads are invariant and evaluate to the same value at any position in the SCoP, the code can be represented as if the memory loads are actually executed prior to the SCoP. For our simple example the assumption is statically *true* as there are no writes that might alter the values of the vector view members inside the loop. Consequently, the polyhedral representation can be built as if the input looked like the one presented in Figure 3.44c. While such an assumption-based representation is generally possible for any (non-volatile, non-atomic) memory load, it is only practical for loads of fixed, not loop iteration dependent locations that are *potentially invariant*. The reason is twofold: First, it is cheaper to verify invariance for accesses to fixed locations (ref. Section 3.6.1). Second, it is generally unlikely for loads of varying locations to have the same value and it is unpractical to introduce a parameter for each one.

Memory loads are called *required invariant* if they are part of a control condition or array access expression, including the array base pointer expression. While any invariant load can be hoisted as an optimization, we focus our discussion on required ones since they prohibit precise polyhedral representation. As mentioned above, only loads of fixed, not loop iteration dependent locations[15] should be considered invariant. Additionally, we exclude loads that are trivially overwritten inside the SCoP, e.g., based on their dominance relation to a write access with the same syntactic pointer expression. If a memory load inside a control condition or array access expression is variant or if the loaded location is iteration dependent, the load will not be represented as a parameter. Instead, the surrounding expression is either approximated (ref. Section 3.4), excluded from the SCoP (ref. Section 6.3) or the reason for the SCoP to be dropped altogether.

3.6.1 Correctness

```
for (i = 0; i < (UseUB ? *UB : 64); i++) {
  A[i] = ...;
```

(a) Guarded invariant memory load (`*UB`) that cannot be accessed unconditionally. If the guard, here `UseUB`, evaluates to *false*, the pointer `UB` might be invalid, thus not dereferenceable.

```
if (2 * P > P)
  A[Offset[P] + i] = ...;
```

(b) Invariant memory load guarded by a condition that might not be modeled accurately due to the potential integer overflow (ref. Section 4.2).

```
for (i = 0; i < *UB; i++) {
  if (ShrinkUB) *UB = *UB - 1;
```

(c) A memory load (`*UB`) in the loop bound that is not necessarily invariant during the execution of the loop.

Figure 3.45: Example programs to show possible correctness issues arising from invariant load hoisting. In the first example (top) the dynamic loop bound is only conditionally accessed and should not be pre-loaded unconditionally. The second example (left) is similar, though the condition might not be modeled accurately due to a potential integer overflow. The third example (right) shows a conditionally invariant access in loop upper bound that cannot be unconditionally assumed to behave like a parameter.

Figure 3.45 illustrates three main correctness issues for invariant load hoisting, conditionally executed loads, approximations in the modeling and potentially varying values. In the first example, 3.45a, the value of `*UB` is always invariant but the access is guarded. Pre-loading it unconditionally in order to make the value available prior to the loop (ref. Figure 3.44c) is therefore not necessarily sound. If the guard does never evaluate to *true*, the pointer might not be dereferenceable and accessing it would cause undefined behavior that will most likely manifest in a program crash. To ensure correctness we therefore have to determine the conditions under which a memory load behaves like a parameter and, in addition, when it is safe to access it, e.g., pre-load its value. Approximations in the modeling can cause accesses to look invariant or unconditionally executed. An example for the latter is shown in Figure 3.45b. If the potential integer overflow in the multiplication is ignored, the access could falsely be assumed to happen for all positive values of the parameter `P`. Instead, the access will also not be executed for large values of `P` as

[15] A memory location is also considered fixed if it depends on code or loop iterations which are not part of the analyzed code region (or SCoP) as their value is fixed for each execution of the SCoP.

discussed in more detail in Section 4.2. Note that other approximations can have a similar effect and need to be taken into account as well. The last correctness issue is shown in Figure 3.45c. The load of `UB` is only conditionally invariant. Only if `ShrinkUB` is *false*, the loaded value will not change during the execution of the loop and therefore be represented as a parameter.

Note that we will discuss the effects of potentially aliasing pointers in Section 4.1. Their interaction with invariant load hoisting is additionally detailed in Section 4.1.4.

3.6.1.1 Invariance Test

To identify invariant loads we will perform a simple *invariance test*. It is based on the polyhedral representation which is built under the assumption that all *required* invariant loads are in fact invariant. The test is not a boolean predicate by itself. It will instead determine the parameter conditions, or invariant load assumptions Λ_{IL}, under which the loads behave like parameters.

A read access r in statement `S` with the access function f_r is invariant if there is no write access to the read location $l_r := f_r(\mathcal{D}_{\mathtt{S}})$. The set of all locations written in a SCoP is denoted as $\mathcal{W}$ and defined in Formula 6.

$$\mathcal{W} := \bigcup_{\mathtt{S} \in SCoP} \; \bigcup_{w \in \mathtt{S}} f_w(\mathcal{D}_{\mathtt{S}}) \tag{6}$$

A read access r is *always* invariant if l_r is never written, thus $\mathcal{W} \cap l_r = \{\}$. The read is *conditionally* invariant if there are parameter constraints under which l_r is not written. Hence, the invariant load assumption $\Lambda_{\text{IL}}(r)$ for a single read access is the parameter context of the negated intersection shown in Formula 7. To obtain this context the projection onto the parameter space $\pi_\rho(\circ)$ is used.

$$\Lambda_{\text{IL}}(r) := \pi_\rho(\neg(l_r \cap \mathcal{W})). \tag{7}$$

Formula 8 shows the invariant load assumptions Λ_{IL} for the whole SCoP. It is the intersection, thus the common parameter constraints, of the invariant load assumptions for all *required* invariant loads $\mathcal{R}_{ril}$.

$$\Lambda_{\text{IL}} := \bigcap_{r \in \mathcal{R}_{ril}} \Lambda_{\text{IL}}(r) \tag{8}$$

As mentioned before, there can be potentially invariant loads that are not used in control flow or access expressions. While we will not require them to be invariant, we will detect and hoist them if they are *always* invariant. This optimization does not introduce new assumptions, thus the inputs for which the polyhedral optimization is valid remains the same.

3.6.1.2 Execution Context

The *execution context* Δ of an invariant load contains all parameter combinations under which the load can be executed safely, hence without introducing undefined behavior that was not present in the input program. Given a precise polyhedral representation, an invariant access r is executed under the parameter constraints that will result in a non-empty domain $\mathcal{D}_r$ of the surrounding statement, thus $\Delta_r := \pi_\rho(\mathcal{D}_r)$. The execution context is not needed, or alternatively equivalent to the static condition *true*, if we can prove that an access is always safe. This is especially important as it allows unconditional pre-loading of most stack and global variables.

An optimistic polyhedral optimizer [DGH17] might employ assumptions to represent iteration domains and access functions (ref. Section 3.5). If these assumptions are violated, the constraints under which a load is assumed to be executed, as well as the location that is assumed to be accessed, could be different. It is consequently not sound to pre-load an access r if the assumptions Λ_r that justify the representation of the domain $\mathcal{D}_r$, as well as the access function f_r, do not hold.

```
val_r1 = *r1;
val_r2 = getZeroForType(val_r2);
if (notEmpty($\pi_\rho(\Lambda_{r_2} \cap \Delta_{r_2})$))
  val_r2 = *r2;
// RTC and optimized SCoP
// which use to the pre-loaded
// values val_r1 and val_r2.
```

Figure 3.46: Two pre-loaded invariant accesses r_1 and r_2. The former is loaded unconditionally while the latter is guarded by a check of the execution context Δ_{r_2} and the required assumptions Λ_{r_2}.

To make the values of invariant loads available at runtime, they are either loaded on-demand or pre-loaded once. We implemented the latter option as illustrated in Figure 3.46. Before the runtime check (RTC), the values of the potentially invariant accesses r_1 and r_2 are determined. For this example we assumed that the access to r_1 was statically proven safe and that there were no assumptions needed to represent the access function f_{r_1}. For the access to r_2 we assume the opposite. The hoisted load of r_2 is consequently guarded by a check of the execution context Δ_{r_2} and the required assumptions Λ_{r_2}. If, at runtime, the condition is *false*, either the execution context Δ_{r_2} was empty or the required assumptions Λ_{r_2} were not fulfilled. The actual value of r_2 is in either case not required. If the execution context Δ_{r_2} was empty, the value is not used in the SCoP, otherwise the RTC will fail and the original, unoptimized code version is executed. Note that the code for the initialization and control condition (both in italic) is optimized and generated statically (ref. Section 3.5.3).

Pre-loading invariant loads prior to the SCoP can be seen as an optimization, especially in combination with load coalescing (ref. Section 3.6.2). Additionally, it allows to generate and check the execution context only once, which decreases the verification overhead at runtime. However, if invariant loads can be accessed unconditionally, and especially if they are not required by the RTC, it might be beneficial to pre-load them late in order to shorten their lifetime.

3.6.2 Load Coalescing

Our approach coalesces invariant loads of the same location. This allows us to explicitly represent their equivalence with a single parameter in the polyhedral representation. In addition, the preloading and consequently the execution context check is only done once. There are several syntactic [ASU86; Cho+98] and semantic ways [Cli95; CH78] to determine if access locations are equal. However, since the access function can be specialized by control constraints (ref. Figure 3.47) we have to use a flow-sensitive comparison to distinguish syntactically equal accesses. In addition we require coalesced accesses to have compatible types, e.g., `int64_t` and `uint64_t` are fine, but `int8_t` and `float` are not. The execution context for a set of coalesced invariant loads is the union of the individual execution contexts as any access is sufficient to justify dereferenceability for the entire SCoP. However, the assumptions that justify the correct representation of each access have to be checked regardless.

```
... = ... A[c] ...;
if (c == 0)
  ... = ... A[c] ...;
```

Figure 3.47: Simplified situation in which a control constraint, here `(c == 0)`, specializes the polyhedral access function for one of two invariant loads with syntactically equal location.

3.6.3 Evaluation

To evaluate the effects of our invariant load hoisting technique we performed several experiments and determined various metrics described in Section 3.1. First, we allowed SCoP detection to employ invariant load hoisting but then excluded SCoPs that did contain them when we evaluated the metrics. The results are shown in Table 3.49a as relative percentages of the results achieved for all SCoPs shown in Figure 3.3 and Table 3.4 on Page 26 and 27. As these numbers only express how invariant load hoisting was used when it was enabled, we re-run the experiments with invariant load hoisting completely disabled[16]. The results are shown in Table 3.49b, again as relative percentages of the results achieved for all SCoPs. The difference between the two experiments stems from the cost-oblivious SCoP detection implementation in LLVM/POLLY. The example code in Figure 3.48

```
N = Sizes[0];
for (i = 0; i < N; i++)
  S(i);
                                SCoP 0
M = Sizes[1];
for (j = 0; j < M; j++)
  P(j);
Sizes[0] = Sizes[1] = 0;        SCoP 1
for (k = 0; k < N + M; k++)
  Q(k);
```

Figure 3.48: Example code with two mutually exclusive valid SCoP regions if invariant load hoisting is enabled. If not, only the region denoted as *SCoP 1* remains valid.

[16] Our base version of LLVM/POLLY provides a switch to disable invariant load hoisting but it does so only partially. While it prevents the actual hoisting, it does not always force the SCoP detection to fail when loads are required to be invariant. Instead it collects these required invariant loads as if they would be hoisted later. Since the hoisting is actually disabled, the SCoP will be built but then classified as statically infeasible (ref. Figure 3.43). In our evaluation this behavior is corrected and invariant load hoisting is either completely enabled or disabled.

illustrates this behavior. If invariant load hoisting is enabled, either of the two SCoPs could be detected. Though, because they overlap and any expansion would statically violate our invariance assumption, only one of the two is recognized. Since there is no cost heuristic involved it could be either of them, depending on the order LLVM will present the regions to POLLY's SCoP detection. If however invariant load hoisting was disabled, *SCoP 1* is still recognized as a valid SCoP while *SCoP 0* is invalid because it then contains dynamic loop bounds.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	38.2%	43.3%	32.3%	67.5%
# depth 1 SCoPs	33.7%	38.6%	28.6%	51.2%
# depth 2 SCoPs	77.8%	69.7%	72.3%	92.8%
# depth 3 SCoPs	66.7%	100.0%	85.7%	92.3%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	42.4%	47.0%	35.8%	75.6%
C_1 score	73.3%	71.4%	74.1%	92.7%

(a) Relative results if required invariant loads were allowed during SCoP detection but SCoPs that contained them are excluded when the metric was evaluated.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	81.8%	55.4%	70.5%	76.7%
# depth 1 SCoPs	82.7%	48.7%	67.0%	63.1%
# depth 2 SCoPs	77.8%	93.9%	108.5%	97.8%
# depth 3 SCoPs	66.7%	100.0%	114.3%	96.2%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	80.8%	60.5%	73.7%	83.3%
C_1 score	73.3%	94.3%	109.3%	97.4%

(b) Relative results if invariant load hoisting was disabled completely.

Table 3.49: Relative scores in various metrics (ref. Section 3.1) compared to the baseline results shown in Figure 3.3 and Table 3.4 on Page 26 and 27. In the top part, SCoPs with required invariant loads were recognized but excluded from the metric count. The bottom part shows the results if invariant load hoisting was completely disabled, thus SCoP detection was not allowed to require loads to be invariant.

When we compare the results shown in the two parts of Table 3.49 we can see that the applicability effect of invariant load hoisting varies significantly between the different benchmark suites. Though, if disabled, all suites show a decrease in the number of SCoPs, reaching from 18.2% to 44.6%. Similarly, the monotone applicability scores (ref. Section 3.1.1) indicate a decrease in the SCoPs optimization potential of up to 39.5%. Note that the increase in the number of depth two and three SCoPs, as well as the applicability score improvement for the SPEC2017 benchmark suite, is caused by 9 additionally *feasible* SCoPs. More feasible SCoPs are a consequence of the greedy SCoP detection which, if invariant load hoisting was enabled, expanded valid SESE regions to a point where the assumptions required to ensure a sound polyhedral representation could not be fulfilled anymore. This is a general implementation problem of LLVM/POLLY as it forces the maximal valid regions, for which a polyhedral representation should be built, to be determined prior to the modeling. Since an unfeasible combination of assumptions will only be detected after the polyhedral representation was built, and there is no (simple) way to shrink the valid region afterwards, subregions with feasible assumptions will not be considered anymore.

		SPEC2000	**SPEC2006**	**SPEC2017**	**LLVM-TS**
SCoP Modeling	inv. loads	320 (66.4%)	661 (62.3%)	3631 (72.5%)	962 (43.5%)
	req. inv. loads	226 (61.8%)	441 (56.7%)	2084 (67.7%)	618 (32.5%)
	coal. inv. loads	47 (13.6%)	120 (14.7%)	1180 (13.3%)	157 (13.9%)
	feasible Λ_{IL}	6 (5.5%)	4 (1.3%)	47 (2.9%)	27 (3.3%)
	infeasible Λ_{IL}	8	22	81	150
Code Gen.	no preload	27.3%	29.4%	20.8%	40.9%
	succ. preload	57.3%	57.6%	63.2%	37.1%
	unsucc. preload	3.6%	1.3%	1.8%	3.8%

Table 3.50: Statistics for various invariant load hoisting related events. The percentages indicate in how many *feasible* SCoPs the event occured in while the number provides the total occurrence count.

In addition to the impact invariant load hoisting has on the different applicability metrics, we provide information on other related events in Table 3.50. Each row provides up to two results per benchmark suite: First, the number of occurrences of the specific event and second, the percentage of *feasible* SCoPs that were affected by the event. The second number is always relative to the number of feasible SCoPs as reported in Table 3.4 on Page 27. The first three rows show the number of invariant loads, required invariant loads and coalesced invariant loads as well as the percentage of SCoPs that contained them. For the SPEC benchmark suites $\approx 5\%$ of all SCoPs contain invariant loads that were not required while the same holds true for 11% of the SCoPs in the LLVM test suite. The absolute number of invariant loads is between 41.6% and 74.2% higher than the number of required invariant loads. Load coalescing is performed in 13-15% of all SCoPs, consistent across the benchmark suites. Between 14.7% (SPEC2000) and 32.5% (SPEC2017) of all invariant loads were coalesced. The two center rows report the number of taken invariant load assumptions Λ_{IL} that were statically feasible/infeasible (ref. Section 3.6.1.1). These assumptions are only necessary if a non-read only array contains elements that were *required invariant* (ref. Figure 3.45c). This situation occurred in 2.3% (SPEC2006) to 10.3% (LLVM-TS) of all feasible SCoPs that also contained required invariant loads. The number of infeasible SCoPs is high because the greedy SCoP detection uses the rather simple heuristic described in Section 3.6 to determine if required invariant locations might be overwritten. Only when the polyhedral representation of a maximal and valid SESE region is built, we perform the accurate invariance test described in Section 3.6.1.1 to determine if and when required invariant locations are actually overwritten. However, as already described above, the implementation of LLVM/POLLY does not (easily) allow to shrink a SCoP with statically infeasible assumptions. The three bottom rows show for how many SCoPs no invariant loads were generated or the invariant loads were successfully/unsuccessfully pre-loaded. The sum of these categories is not 100% since not all feasible SCoPs actually reached the code generation phase[17].

The runtime cost and general misspeculation rate of assumptions are provided in Section 3.5.5.1.

3.6.4 Related Work

Several optimizations, including loop-invariant code motion [ASU86; Cli95] and partial redundancy elimination [BGS99; Cho+98; MR79], hoist invariant loads out of loops. Such low-level transformations commonly try to move loads to predecessor blocks or loop pre-headers if they can be unconditionally loaded there. In contrast to these techniques we use a high-level perspective to solve the problem, the polyhedral representation of the program that was built under the assumption that some loads are invariant. We can utilize it not only to determine the conditions under which the loads are invariant, e.g., with regards to stores to the same array, but also to derive constraints that prevent aliasing or overlapping pointers (ref. Section 4.1). Since aliasing is generally a major limitation whenever memory instructions are moved, it is common for alias analysis improvements [CH00; DMW98; DMM98] to evaluate their impact on loop-invariant code motion or (partially) redundant load elimination. In contrast to other approaches [BA98; KRS98; KRS99] we did not investigate the placement of invariant loads further. Instead, we rely on LLVM to move hoisted loads to the most suitable location. However, it could be beneficial to place them directly inside, not in front, of the optimized SCoP. Even if preloaded values are needed for the runtime check, reloading them later could shorten lifetimes and improve the result of certain register allocation schemes.

Fitzgerald et al. [Fit+00] extend known invariant load hoisting techniques to Java programs that can exhibit implicit writes due to synchronization events. Their technique is limited to loads of addresses that are known to be non-null, thus always accessible. This is similar to universal execution contexts that do not require runtime checks. Odaira and Hiraki [OH05] hoist instructions regardless of potential exceptions, e.g. due to null pointers. To ensure correctness if exceptions occur, they employ special exception handlers that rectify the situation at runtime. Xu, Yan, and Rountev [XYR12] proposed to go beyond expression or loads and hoist the allocation of complex but loop-invariant data structures in Java programs.

[17] SCoPs are discarded due to complexity or profitability issues (ref. Section 3.1.4) but also implementation artifacts.

3.7 Runtime Check Hoisting

Runtime checks are used for program versioning by optimistic optimizations (ref. Section 3.5), but they are also native to programs or implicitly present to guarantee the language semantics. Developers commonly utilize them to explicitly guard debug or timing statements, e.g., as shown in Figure 3.51, and to identify error states, as done by the uBLAS linear algebra implementation in the C++ boost project [WK00]. Additionally, compilers manifest checks implicitly present in the program, e.g., to guard memory accesses in safe languages like Java, Julia, or Rust. Even for unsafe languages, such as C/C++, there exists compiler support to emulate this memory safety for debugging and testing purposes [Ser+12]. To keep the program, as well as such compiler extensions maintainable, runtime checks are usually placed directly at the potentially offending operation. This way, a basic front-end pass can introduce out-of-bound checks at each memory access in the program. Later optimizations are then tasked to simplify, move, or even eliminate such checks [BGS00; Gam+08; Nie+09; QHV02; WWM07]. However, not all runtime checks can be eliminated and, if they are placed within loops, they will not only hurt performance but also induce loop carried dependences that prevent most loop transformations. While languages like Julia and libraries like uBLAS offer the programmer to omit checks, the guarantees that come with them are consequently lost, too.

```
for (j = 1; j <= grid_points[1]-2; j++)
  for (i = 1; i <= grid_points[0]-2; i++)
    for (m = 0; m < 5; m++)
      [...]
if (timeron) timer_stop(t_rhsz);
for (k = 1; k <= grid_points[2]-2; k++)
  for (j = 1; j <= grid_points[1]-2; j++)
    for (i = 1; i <= grid_points[0]-2; i++)
      [...]
```

Figure 3.51: Excerpt from BT benchmark in the NAS parallel benchmark suite [SJL11].

The runtime check hoisting we integrated into LLVM/Polly uses assumptions and the code specialization framework described in Section 3.5 to ensures that "error states" are never reached. In this context, we treat basic blocks as error states if they are: (1) not dominating all in-SCoP predecessors of the SCoP exit block, (2) directly dominated by a conditional which is not a loop header, (3) not loop headers themselves, and (4) containing a call to an unknown, non-pure function. We use these easy to test conditions as a preliminary heuristic to limit the error state candidates. Condition (1) will ensure that no single error state is unconditionally executed in the SCoP. The next two conditions try to prevent assumptions that

Runtime check hoisting is a control flow graph (CFG) specialization technique we integrated into LLVM/Polly. While it was initially part of our work on optimistic loop optimizations [DGH17], it was not described in the paper. In contrast to the more general *polyhedral program slicing* we present in Section 6.3, runtime check hoisting is fully integrated in LLVM/Polly and also enabled by default.

would cause loops to be considered error states. Finally, the last condition ensures that error states would act as an optimization barrier.

To build the error block assumptions Λ_{EB} we first determine the set of error statements, denoted as ErrorStmts, that contain a basic block which is considered an error state. For each such statement $\mathtt{S}$, we derive the statement error block assumption $\Lambda_{\text{EB}}(\mathtt{S})$, a parameter context that ensures the statement is never executed. To this end, we project the complement of the iteration domain $\mathcal{D}_{\mathtt{S}}$ onto the parameter space ρ (ref. Section 2.1.2). Note that the resulting set of conditions is only dependent on parameters of the SCoP and it evaluates to *true* only if statement $\mathtt{S}$ is will not be executed for the parameter valuation at runtime. Formula 9 and 10 show the definition of statement- and SCoP-wide error block assumptions. The latter Λ_{EB} is the intersection of all statement error block assumptions for statements that contain an error block.

$$\Lambda_{\text{EB}}(\mathtt{S}) := \pi_{\rho}\left(\neg(\mathcal{D}_{\mathtt{S}})\right) \quad \textbf{(9)}$$

$$\Lambda_{\text{EB}} := \bigcap_{\mathtt{S}\,\in\,\text{ErrorStmts}} \Lambda_{\text{EB}}(\mathtt{S}) \quad \textbf{(10)}$$

For the excerpt of the BT benchmark shown in Figure 3.51, the error block assumptions would be used to ensure the function call `timer_stop` (and others) are never reached. Thus, at runtime, the value of `timeron` has to be *false*.

3.7.1 Evaluation

To determine the effects of runtime check hoisting on the applicability and profitability of SCoPs, we performed two separate experiments which are summarized in Table 3.52, 3.53, and 3.54.

SPEC2000		SPEC2006		SPEC2017		LLVM-TS	
w/ Λ_{EB}	w/o Λ_{EB}	w/ Λ_{EB}	w/o Λ_{EB}	w/ Λ_{EB}	w/o Λ_{EB}	w/ Λ_{EB}	w/o Λ_{EB}
149	148	368	327	1680	1617	649	645

Table 3.52: Number of maximal and valid regions found during SCoP detection with (w/ Λ_{EB}) and without (w/o Λ_{EB}) runtime check hoisting. Note that this is not the number of (valid) SCoPs because it also includes regions for which the polyhedral representation was deemed unprofitable or the required assumptions were statically infeasible.

In the first experiment, we ran our baseline version of LLVM/Polly but evaluated the metrics (ref. Section 3.1) only for SCoPs that did not contain error states. The numbers in Table 3.54a show that the applicability with regards to the number of SCoPs and also the monotone applicability scores C_{α} drops for each benchmark suite. Especially the number of depth 2 SCoPs in SPEC2006 and SPEC2017 was reduced significantly by up to 48.5%. A similar decrease can be observed for the C_1 metric (ref. Section 3.1.1). It determines the optimization potential of SCoPs with at least two loops, nested or consecutive.

In the second experiment, runtime check hoisting was disabled, thus SCoPs could not contain error states. The results, shown in in Table 3.54b, indicate that this extension often has a negative impact on the overall applicability. For SPEC2000 and the LLVM Test Suite, all metrics report higher values compared to our baseline. This effect can be explained with the numbers presented in Table 3.53 which show how many polyhedral representations were built but afterwards ruled unprofitable (ref. Section 3.1.4) or dropped due to statically infeasible assumptions (ref. Section 3.5.5.3). Without runtime check hoisting there is a significant decrease in unprofitable SCoPs and infeasible assumptions. The impact on the metrics is comparably small since the number of maximal and valid regions found was increased by a similar amount. Since these additional regions, quantified in Table 3.52, could not improve our metrics, we believe error states are identified too aggressively in order to expand already valid SCoPs.

	w/ Λ_{EB}	w/o Λ_{EB}
SPEC2000	39	31
SPEC2006	137	96
SPEC2017	480	420
LLVM-TS	232	220

Table 3.53: Number of unprofitable polyhedral representations and statically infeasible assumptions with and without runtime check hoisting.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	64.5%	79.7%	65.2%	80.1%
# depth 1 SCoPs	62.2%	84.3%	65.5%	77.4%
# depth 2 SCoPs	77.8%	51.5%	60.6%	81.9%
# depth 3 SCoPs	100.0%	100.0%	71.4%	96.2%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	67.2%	76.3%	64.9%	82.0%
C_1 score	86.7%	54.3%	62.0%	86.0%

(a) Relative results if SCoPs containing error states were excluded from the metrics.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	106.3%	100.0%	99.8%	101.9%
# depth 1 SCoPs	107.1%	99.5%	99.5%	101.6%
# depth 2 SCoPs	100.0%	103.0%	103.2%	102.9%
# depth 3 SCoPs	100.0%	100.0%	100.0%	100.0%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	105.6%	100.4%	100.0%	102.0%
C_1 score	100.0%	102.9%	102.8%	102.1%

(b) Relative results if runtime check hoisting was disabled completely.

Table 3.54: Relative impact of runtime check hoisting in various metrics (ref. Section 3.1) compared to the baseline results in Figure 3.3 and Table 3.4 on Page 26 and 27. In the top part, SCoPs that contain error states were recognized but excluded from the metrics. The bottom part shows the results if runtime check hoisting was disabled.

3.7.2 Related Work

The only conceptual difference between the runtime check hoisting and the general polyhedral program slicing described in Section 6.3 is the eagerness to identify error states. Our implementation of the former in LLVM/POLLY already restricts potential error state candidates to prevent aggressive SCoP expansion that would employ error block assumptions to generate large, only sparsely populated SCoPs. Even though this heuristic is far from optimal, it allowed us to enable this technique by default while keeping the optimization potential consistent.

The memory in-bounds assumptions Λ_{IB}, introduced by Grosser et al. [Gro+15] and extended later on [DGH17], ensure that the polyhedral representation of multidimensional memory accesses adhere to the array dimension sizes. However, safe languages and security oriented libraries can require explicit runtime checks for all memory accesses. In this case, our runtime check hoisting can be used to built a polyhedral representation of the input, assuming the inbound checks only depend on SCoP parameters. Since the explicit manifestation of such inbounds checks blurs the line between in-bounds assumptions Λ_{IB} and error block assumptions Λ_{EB}, works akin to the former are also related to the latter. However, approaches that deal with potential out-of-bound memory accesses [BGS00; Gam+08; Nie+09; QHV02; WWM07], are less general than the presented approach (ref. Figure 3.51).

There are several approaches that perform speculative, polyhedral-model-based parallelization in the presence of unknown side-effects [Jim+12; Jim+13b; Pra11; Suk+14]. These schemes ignore dependences emanating from conditionally executed code which was dead during an initial profiling period. Due to the speculative execution, a runtime verification system has to be in place that monitors the execution. In case the speculation was successful, the computed results are finalized. However, misspeculation require a rollback of all effects caused by eagerly executed statement instances. To reduce the tracking and rollback overhead, these schemes often employ chunking, a one-dimensional loop tiling around the speculatively executed conditionals. While systems like these are more powerful than our runtime check hoisting, they require far more machinery which can easily introduce a non-trivial overhead [Ham17, Chapter 4]. We believe that for cases where our static code versioning is applicable, it is be the preferable choice.

There are various non-polyhedral-model-based code specialization techniques that are similar to our approach as they also determine predicates under which certain code parts are not executed [KCB07; MWD00; Oh+13]. However, these schemes perform specialization as an end in itself and additionally require actual profiling information to identify promising candidates. As our technique was developed to enable polyhedral schedule optimizations, we have not looked into its application as a stand-alone optimization. We are also not directly concerned with problems like code size increase or the benefit of runtime check hoisting. Instead, we expect the polyhedral optimizer to determine the profitability of the applied transformations (ref. Section 3.1.5).

Chapter 4

Correctness

> *The most important property of a program is whether it accomplishes the intentions of its user.*
>
> C.A.R. Hoare, *An Axiomatic Basis for Computer Programming*, 1969

General purpose programs do often not completely fulfill the requirements of the polyhedral model [Doe+13; DGH17; Sim+13]. While improvements in the modeling, e.g., of function calls (ref. Section 6.1), lower the requirements of polyhedral techniques, and approximations are appropriate for some cases [Ben+10; MDH16; ZKC18], the polyhedral model is still restrictive. This is especially true for programs that exhibit different behaviors for corner case inputs. Enumerative modeling of these corner case behaviors increases complexity and thereby cause scalability issues. Approximations alternatively introduce spurious behaviors that prevent meaningful transformations. However, any semantic mismatch between the program and the polyhedral representation can cause silent miscompilations. While manual or semi-automatic optimization tools shift some responsibly to the user [AB15; Bag+15; Bas+03; Bon+08; Ver+13], fully-automatic approaches have to guarantee correctness on their own, optimally without sacrificing applicability.

Chapter 3 dealt with applicability limitations, while very similar to correctness issues, their impact on the polyhedral representation is different. For applicability enhancements, specialized modeling techniques alter the polyhedral representation to encode the program semantics. For correctness issues, the default representation is the desired one and already sound for most, but not all, inputs. While an additional, explicit representation of the corner case semantics is often possible, it is generally not beneficial. Nevertheless, the model has to be correct for all allowed inputs, especially if the optimizer is used in an unsupervised fashion on general purpose code.

In this Chapter we describe how a sound polyhedral representation can be built for low-level input programs. We focus on efficient modeling and optimization of common inputs rather than a complete representation of all possible corner cases that are covered by the program semantics.

Nevertheless, we provide sound solutions utilizing the assumption and code specialization framework introduced by Doerfert, Grosser, and Hack [DGH17] and already discussed in Section 3.5.

The example in Figure 4.1a illustrates common correctness issues that occur in general purpose code. The loop nest averages `2 * NB + 1` neighboring values from the array `A` and store the results in the array `B`. Though, the average is only computed for indices between `NB` and `SIZE - NB`, for all other indices the input is just copied. While the code seems intuitively amenable to various loop optimizations, the actual semantics depends on the context in which the code is placed. The programming language and the actual types involved are crucial for an example like this.

While the example may seem contrived, it is based on a bug reported to the LLVM/POLLY project[1]. Similarly, all correctness issues described in the following have been observed in general purpose code that was seemingly amenable to polyhedral optimizations. These situations illustrate how subtle changes to the input, or its context, can cause vastly different semantics that have to be accounted for in the polyhedral program representation.

```
for (i = 0; i < SIZE; i++) {
  if (NB <= i && i <= SIZE - NB) {
    result = 0;
    for (n = -NB; n <= NB; n++)
      result = result + A[i+n];
    B[i] = result / (NB*2+1);
  } else {
    B[i] = A[i];
  }
}
```

(a) Original version with two comparisons and a boolean operator in the if condition.

```
for (i = 0; i < SIZE; i++) {
  if (i - NB <= SIZE - 2 * NB) {
    result = 0;
    for (n = -NB; n <= NB; n++)
      result = result + A[i+n];
    B[i] = result / (NB*2+1);
  } else {
    B[i] = A[i];
  }
}
```

(b) Optimized version of part 4.1a with only one comparison in the if condition.

Figure 4.1: Two versions, original[a] and optimized, of a loop nest to average neighbouring values in the center of the array. The optimized version (right) intentionally uses an underflow in the computation `i - NB` to reduce the complexity of the if condition. Note that we do assume the underflow to be well defined here, thus the use of *wrapping* semantics as introduced in Section 4.2 and not C/C++ semantics for signed values.

[a] The example is taken from a *Bones* tool test cases by Nugteren and Corporaal [NC12].

If we assume a memory-safe language such as Java, Julia or Rust, we have to consider the implicit index out-of-bounds check at each array access (ref. Section 3.7). Such error handling not only makes static dependence analysis less precise, but, as part of the language semantics, also prevents transformation. In the example, the iteration order of the loops would be fixed as exceptions could reveal an intermediate memory state. A similar situation arises if a memory sanitizer for unsafe languages such as C/C++ is used.

[1] Bug report by Roel Jordan, see `https:goo.gl/Gppgyd`.

Another well-known correctness issue are integer overflows, hence computations that require more bits than available to represent the result exactly. The overflow semantics of C/C++ depends on the types involved. Operations on n-bit unsigned types use machine arithmetic in which an overflows will “wrap around”. Thus, only the n least significant bits of the result are considered and the rest is ignored. A programmer or a compiler could exploit this behavior to simplify the if condition shown in Figure 4.1a to the one in 4.1b, effectively reducing the number of comparisons by one. However, this is only sound if the expression `i - NB` will actually wrap around for `i < NB`. Hence, the unsigned comparison `i - NB <= Size - 2 * NB` is not equivalent to `i + NB <= SIZE`, since the wrapping behavior was not taken into account. In case the simplified if condition is interpret without wrapping semantics, it evaluates to true for all values of `i` smaller than `SIZE - NB`, while the original did so only between `NB` and `SIZE - NB`.

Similarly, the programmer could have expressed the loop exit condition as `i != SIZE`, however that would not necessarily guarantee `SIZE` iterations in all languages, i.a., Java. A negative `SIZE` would cause a wrap-around and for an n-bit type, therefore $2^n - |\texttt{SIZE}|$ iterations.

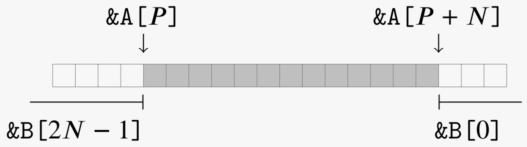
Lastly, we have to distinguish between array identifiers and the associated memory. In most low-level languages the two arrays `A` and `B` could alias, thus the memory accessed through `A` might overlap with the memory accessed through `B`. If the offset between pointers is unknown and aliasing might or might not happen at runtime, there is little chance for any meaningful loop transformation to take place.

4.1 Aliasing & Overlapping Arrays

In the context of polyhedral optimizations we call two arrays *aliasing*, or *overlapping*, if they are used to access the same memory cell in one execution of a Static Control Part (SCoP). Aliasing is one of the main applicability issues of polyhedral optimizers on low-level code [Doe+13]. Since polyhedral tools often require source code annotations, they simply assume an aliasing free environment, effectively shifting the burden of an alias analysis to the user [Bag+15; Bon+08]. Similarly, (semi-)automatic tools, i.a., PLUTO+ or LLVM/POLLY, come with the possibility to ignore potential aliasing issues. Even though this is useful to evaluate new approaches in a controlled environment [AB15], it is inappropriate for unsupervised, fully automatic usage.

```
for (i = 0; i < N; i++)
  A[i + P] += B[2 * i]
```

(a) Example featuring the two possibly aliasing pointers/arrays A and B with equal element size.



(b) Two possible memory layouts that will prevent accesses via A and B to overlap/alias during the execution of the code shown in Figure 4.2a.

Figure 4.2: Accesses to possibly aliasing arrays in part 4.2a and the memory layouts in part 4.2b for which every memory cell is only accessed through a single base pointer.

While we almost never observe aliasing at runtime [DGH17; Guo+06], potential aliasing will nevertheless cause arbitrary dependences between different arrays at compile time. To deal with cases of potential aliasing that were not precluded statically, we employ runtime alias checks and code versioning (ref. Section 3.5). The goal is to determine if any memory cell might be accessed through different base pointers during *one* execution of the analyzed code region *prior* to the execution. If this might be the case, we will execute the original code, otherwise the optimized version in which all arrays are known to be alias free. To simplify the runtime checks, hence to decrease verification time, all accesses to one base pointer are abstracted with ranges [Alv+15; Doe+13]. While more precise checks are possible, e.g., for strided accesses, range based alias checks have shown to be sufficient in practice [DGH17; Guo+06]. For the example in Figure 4.2a, the minimal/maximal accessed locations are &A[P]/&A[$P + N - 1$] and &B[0]/&B[$2N - 2$] respectively. To determine if all locations accessed through different pointers are disjoint it suffices to show that the maximally accessed location via one pointer is smaller than the minimally

> The extent possible aliasing inhibits polyhedral optimizations has been exposed in our first applicability study [Doe+13] where we also introduced runtime checks to prevent aliasing. In later works, we extended the applicability, simplified the checks, and evaluated them more thoroughly [Alv+15; DGH17]. Aliasing checks are enabled in LLVM/POLLY.

accessed location via the other pointer. In our example[2] this is true if $\texttt{\&B[}2N-1\texttt{]} \texttt{ <= } \texttt{\&A[}P\texttt{]}$ or $\texttt{\&A[}P+N\texttt{]} \texttt{ <= } \texttt{\&B[0]}$, thus if the memory layout fulfills the conditions illustrated in Figure 4.2b.

4.1.1 Runtime Alias Check Generation

```
for (i = 0; i < N; i++)
  for (j = i; j < 2 * i; j++)
    A[i][j] += B[j][i] + B[i][j];
```

Figure 4.3: Loop nest with four potentially aliasing multi-dimensional accesses.

Our runtime alias checks are used when the alias analyses performed by LLVM[3] fail to prove the absence of aliasing accesses statically. In the first step, LLVM's analyses are used to collect sets of possible aliasing accesses inside the SCoP. These analyses guarantee the absence of overlapping accesses between different sets, thus we can handle each one in isolation. If a set contains only accesses to read-only arrays it is dropped. This is sound because the potential read-after-read (*RAR*) dependences between these accesses do not need to be preserved. Similarly, we ignore sets that only contain accesses to a single array as the dependences between them will be explicitly computed during the polyhedral dependence analysis as described by Feautrier [Fea91]. For the example shown in Figure 4.3, there is one set containing all four multi-dimensional memory accesses as illustrated in Formula 1.

$$\begin{aligned}\big\{\{\texttt{A[}i\texttt{][}j\texttt{]} \mid 0 \le i < N \wedge i \le j < 2*i\},\ \{\texttt{A[}i\texttt{][}j\texttt{]} \mid 0 \le i < N \wedge i \le j < 2*i\},\\ \{\texttt{B[}j\texttt{][}i\texttt{]} \mid 0 \le i < N \wedge i \le j < 2*i\},\ \{\texttt{B[}i\texttt{][}j\texttt{]} \mid 0 \le i < N \wedge i \le j < 2*i\}\big\}\end{aligned} \qquad \textbf{(1)}$$

Accesses to an array are summarized as the range between the lexicographically smallest and largest accessed array element. To this end, the lexicographic minimum and maximum of each memory access function over its respective iteration domain is computed. In contrast to the original accesses, these ranges do not depend on the loop iterations but only involve constants and parameters. Consequently, they can be compared prior to the SCoP. For our example loop nest in Figure 4.3, the pairs of minimal and maximal accessed array elements are shown in Formula 2.

$$\big\{(\texttt{A[0][0]},\ \texttt{A[}N-1\texttt{][}2N-2\texttt{]}),\ (\texttt{B[0][0]},\ \texttt{B[}2N-2\texttt{][}2N-2\texttt{]})\big\} \qquad \textbf{(2)}$$

The above range describes accessed array elements but overlapping might occur on single bytes in memory. To obtain a range that includes all accessed bytes in memory, the extent of the smallest or, depending on the memory layout, largest accessed element needs to be taken into account. Since all bytes till the beginning of the next smaller/larger one are accessed, it is sufficient to decrement/increment the innermost dimension of the smallest/largest accessed array element by

[2] Note that we use C/C++ syntax here but our runtime checks are generated in LLVM-IR. The latter has a defined semantics for certain features the former prohibits, e.g., relative comparison between pointers to different objects.

[3] LLVM provides various alias analyses but not all are enabled by default. Available analyses include the one by Diwan, McKinley, and Moss [DMM98], Lattner, Lenharth, and Adve [LLA07], and Steensgaard [Ste96].

one. Assuming the memory addresses grow with the array indices, the largest element needs to be adjusted and the accessed memory ranges for our example are described by Formula 3.

$$\{(\mathtt{A[0][0]}, \mathtt{A}[N-1][2N-1]), (\mathtt{B[0][0]}, \mathtt{B}[2N-2][2N-1])\} \qquad \textbf{(3)}$$

The actual checks are generated using common polyhedral code generation techniques [GVC15] and the dimension sizes derived for the multi-dimensional accesses [Gro+15]. The final runtime check for the example in Figure 4.3 is sketched as high-level C code in Figure 4.4.

```
bool AliasFree = &B[2*N-2][2*N-1] <= &A[0][0] || &A[N-1][2*N-1] <= &B[0][0];
```

Figure 4.4: Runtime alias check for the loop nest shown in Figure 4.3.

4.1.2 Complexity of Runtime Alias Checks

The examples in Figure 4.5 illustrate complexity challenges associated with runtime alias checks. We first describe the problems and then present practical solutions in Section 4.1.2.1 and 4.1.2.2.

```
for (i = 0; i < N; i++)
  A[i] += B[i + p0] + B[i + p1]
        + B[i + p2] + B[i + p3]
        + B[i + p4] + B[i + p5]
        + B[i + p6] + B[i + p7];
```

(a) Parametric accesses to the B array cause a quadratic growth in the representation of the minimal/maximal access to that array.

```
for (i = 0; i < N; i++) {
  A[i] += B[i] * C[i];
  D[i] += E[i] * F[i];
  G[i] += H[i] * I[i];
}
```

(b) Multiple possibly aliasing arrays cause an exponential number of comparisons needed to exclude aliasing accesses.

Figure 4.5: Examples to illustrate the two scalability issues for runtime alias checks. The minimal (or maximal) access to the B array in the left example is a piecewise affine function with 8 pieces and $8 * (8 - 1) = 56$ inequalities. The right code features nine different and potentially aliasing arrays. To exclude all possible combinations of aliasing we require $9 * (9 - 1) = 72$ comparisons in total.

LLVM/Polly relies on the representation and transformation capabilities of isl [Ver10]. The isl representation of the minimal and maximal access descriptions grows quadratically with the number of parameters involved. For the example in Figure 4.5a, this leads to $8 * (8 - 1) = 56$ inequalities. The representation of the minimal offset for only four parametric accesses is shown in Formula 4. It already requires $4 * (4 - 1) = 12$ inequalities.

$$\begin{aligned}&\{\, p0 \mid p0 \le p1 \land p0 \le p2 \land p0 \le p3 \,\} \cup \{\, p1 \mid p1 \le p0 \land p1 \le p2 \land p1 \le p3 \,\} \cup \\ &\{\, p2 \mid p2 \le p0 \land p2 \le p1 \land p2 \le p3 \,\} \cup \{\, p3 \mid p3 \le p0 \land p3 \le p1 \land p3 \le p2 \,\}\end{aligned} \qquad \textbf{(4)}$$

The time needed to compute the lexicographic minimum/maximum for a loop nest like the one in Figure 4.5a is as shown in Table 4.6. It grows super exponentially in the number of involved parameters and is therefore not feasible in the presence of many accesses parametrized differently.

#P	3	4	5	6	7	8	9	10
sec.	0.002	0.005	0.012	0.034	0.149	0.749	5.181	54.155

Table 4.6: Time in seconds to compute the lexicographic minimum of a Presburger set depending on the number of involved parameters (#P). Measurements were taken with islpy version 2016.2.1 on an Intel(R) Core(TM) i7-4800MQ CPU.

The example in Figure 4.5b involves nine potentially aliasing arrays. As indicated in Figure 4.2b, we need two comparisons per overlap check and one overlap check per pair of pointers. Thus, given n potentially aliasing pointers, we need $n * (n - 1)$ comparisons to exclude overlapping accesses to different arrays. For the example this would result in $9 * (9 - 1) = 72$ comparisons.

4.1.2.1 Parametric Memory Accesses

Our runtime alias check implementation in LLVM/POLLY [DGH17] avoids the complexity increase due to parametric accesses (ref. Figure 4.5a) with an arbitrary threshold on their number. Thus, if more than a fixed number of parameters are involved in the access functions of a single array, we abort the optimization of a SCoP. By default this will happen if more than 8 parameters are involved. While this solution is often sufficient in practice, it can easily lead to lost optimization opportunities for large SCoPs. As an alternative, we designed, implemented and evaluated a new approach that is described in Section 4.1.3.

4.1.2.2 Alias Groups

For a single set of potentially aliasing pointers, the number of comparisons required to exclude aliasing grows quadratically with the number of contained arrays. However, for multiple sets it is only the sum of comparisons required for each individual one. Thus, it is generally beneficial to split a set of potentially aliasing accesses into multiple smaller sets, whenever it is sound. To this end, we use precise control flow information to create *alias groups* that contain only potentially aliasing accesses that might occur under the same parameter conditions. In addition, alias groups are split to ensure that at most one read-only array is contained.

```
    Out = malloc(...);
    for (i = 0; i < N; i++) {
S0:   Out[i] = R0[i];
      if (c)
S1:     A[i] = Out[i] + R1[i];
      else
S2:     B[i] = Out[i] + R2[i];
    }
```

(a) Example featuring six different arrays. Three are written but one (`Out`) cannot alias with the others. The remaining arrays (`R0`, `R1` and `R2`) are read-only.

$$
\begin{aligned}
ag_1 &:= \{(\texttt{A[0]},\texttt{A[N-1]},\{c \neq 0\},\\
&\qquad (\texttt{R0[0]},\texttt{R0[N-1]},\{\mathit{true}\})\}\\
ag_2 &:= \{(\texttt{A[0]},\texttt{A[N-1]},\{c \neq 0\},\\
&\qquad (\texttt{R1[0]},\texttt{R1[N-1]},\{c \neq 0\})\}\\
ag_3 &:= \{(\texttt{B[0]},\texttt{B[N-1]},\{c = 0\},\\
&\qquad (\texttt{R0[0]},\texttt{R0[N-1]},\{\mathit{true}\})\}\\
ag_4 &:= \{(\texttt{B[0]},\texttt{B[N-1]},\{c = 0\},\\
&\qquad (\texttt{R2[0]},\texttt{R2[N-1]},\{c = 0\})\}
\end{aligned}
$$

(b) The four alias groups of size two generated for the example shown in part 4.7a.

Figure 4.7: Example to show how alias groups are generated depending on the kinds and execution contexts of memory accesses. The four alias groups of size two shown in part 4.7b require only $2 * 4 = 8$ comparisons to exclude aliasing. A single alias group with all five pointers would require $5 * 5 - 5 = 20$ comparisons instead.

The example in Figure 4.7a features eight memory accesses to five different pointers spread over three statements. LLVM's alias analyses can determine that the `Out` pointer is alias free because of the allocation just prior to the loop. Since the default analyses are flow insensitive, they will however put the five remaining accesses into a single set. We utilize the control flow information available through the iteration domains of the statements surrounding the accesses to split sets which contain accesses executed under disjoint parameter contexts. For the example, the initial set is divided into two, each with one write and two read accesses, because the parameter contexts of `S0` and `S1` are distinct: $\pi_\rho(\mathcal{D}_{\texttt{S0}}) \cap \pi_\rho(\mathcal{D}_{\texttt{S1}}) = \{\}$. Conceptually, this split prevents alias checks between accesses that are never executed together in one instantiation of the SCoP.

Potential aliases between read-only arrays can only introduce spurious read-after-read (*RAR*) dependences. Since they do not impact correctness, we do not need to exclude them with runtime checks. To this end, we split alias groups that contain accesses to multiple read-only arrays into multiple alias groups with one read-only array each. An alias group $ag := \{w_1, \ldots, w_n, r_1, \ldots, r_k\}$ with accesses to arrays that are not read-only ($w_1, \ldots, w_n$, with $n \geq 1$), and multiple accesses to read-only arrays ($r_1, \ldots, r_k$, with $k > 1$) is split into k alias groups as shown in Formula 5.

$$ag_1 := \{w_1, \ldots, w_n, r_1\}, \;\ldots,\; ag_k := \{w_1, \ldots, w_n, r_k\} \qquad \textbf{(5)}$$

The runtime check to exclude aliasing for the initial alias group ag required $(n + k)^2 - (n + k)$ comparisons. Hence, it was quadratic in both l and k. The checks for the alias groups ag_1 to ag_k can be performed with $k * ((n + 1)^2 - (n + 1))$ comparisons, thus k is now a linear factor.

4.1.3 Limitations & Extensions

```
// f is a pure function
for (i=0; i < N; i++)
  A[f(i)] = ...;
```

(a) Non-affine access caused by an unknown index expression.

```
for (i=0; i < N; i++)
  for (j=0; j < i; j++)
    A[B[j]] = ...;
```

(b) Access with a data-dependent index expression.

```
for (i=0; i < N; i++)
  for (j=0; j < M; j++)
    (Ptr[i])[j] = ...;
```

(c) Access with a data-dependent base pointer.

Figure 4.8: Examples to showcase the limits of the polyhedral runtime alias checks described here and employed by LLVM/Polly.

Our runtime alias checks are derived from the polyhedral representation of a program. They are consequently only as precise as the representation of potentially aliasing accesses is in the model. It is especially required that potentially aliasing accesses have a fixed, statically known base pointer, e.g., not as the access to `Ptr[i]` in Figure 4.8c, and reasonably tight bounds on the access range. While non-affine accesses generally comply with the first requirement, dynamic accesses might not. Additionally, it is not always possible to determine bounds on the index range of non-affine or dynamic accesses (ref. Section 3.4.1). The examples in Figure 4.8 show different situations for which our technique cannot create a runtime alias check. If such a situation arises, there are generally three options: Stop the optimization of the SCoP, model the dependences caused by any potential interleaving of the arrays, or employ a more powerful runtime check, e.g., an inspector loop [RP94] or runtime allocation tracking [Alv+15]. Our current implementation in LLVM/Polly will conservatively choose to reject a single-entry single-exit (SESE) region for which alias checks are required but cannot be build with the presented technique.

To limit the compile time and runtime spent for accesses involving multiple parameters, a threshold is used (ref. Section 4.1.2.1). Especially for larger, more profitable SCoPs this can easily cause missed optimization opportunities. Since the complexity arises from the symbolic representation of the lexicographic minima/maxima using piecewise defined functions, we can employ a different representation instead. The idea is that the minimal and maximal accesses do not need to be determined statically but can also be computed dynamically. Instead of the lexicographic minimal and maximal accesses, we only need the maximal loop bounds[4] to eliminate the induction variables from the access ranges (ref. Formula 1), and thereby allow a single check prior to the SCoP. The actual minimal/maximal offset value is determined at runtime using an n-ary min/max computation as shown in Formula 6 for the example loop in Figure 4.5a.

$$\begin{gathered} \{\, 0 + \min(p0, p1, p2, p3, p4, p5, p6, p7)\,\} \\ \{\, N - 1 + \max(p0, p1, p2, p3, p4, p5, p6, p7)\,\} \end{gathered} \tag{6}$$

[4] The lower bound is trivial since the loops in our representation are normalized to start with zero.

4.1.4 Aliasing Checks & Invariant Load Hoisting

```
for (i = 0; i < *UB; i++)
  (*Ptr)[i] = ...;
```

(a) Simple loop featuring a dynamically loaded upper bound and base pointer. The values of `*UB` and `*Ptr` are needed to perform runtime alias checks, though, at the same time aliasing has to be excluded to justify invariant load hoisting.

```
UBVal = *UB;
if (UBVal > 0)
  PtrVal = *Ptr;
if (UBVal > 0 && (&PtrVal[0] >= &Ptr[1]
    || &PtrVal[UBVal] <= &Ptr[0]) &&
    (&PtrVal[0] >= &UB[1] ||
     &PtrVal[UBVal] <= &UB[0]))
```

(b) Hoisted invariant loads and runtime alias check for the example loop shown in part 4.9a.

Figure 4.9: Example to illustrate the interplay between runtime alias checks and invariant load hoisting (ref. Section 3.6). Without a runtime alias check we cannot prove the loads loop invariant and without invariant loads we cannot create a runtime alias check.

Invariant load hoisting, as described in Section 3.6, assumes an alias free environment. This improves the applicability and may also enable the generation of runtime alias checks which ensure the alias free environment that was assumed. Loads in control flow conditions and access functions need to be assumed invariant to create precise access ranges and consequently alias checks. At the same time, their invariance is conditioned on not being overwritten by aliasing accesses. Figure 4.9 illustrates this situation. A precise polyhedral representation is only possible if both loads are invariant and (conceptually) hoisted. However, that is only sound if the write does not alias with either of them. To this end, we optimistically assume this to be true, hoist the loads, and check for invariance and aliasing, using the initially loaded values. Only if the runtime check rules aliasing out and determines the loads to be invariant, the representation was actually sound and the optimistically optimized code can be executed.

4.1.5 Related Work

Ramalingam [Ram94] showed that pointer aliasing is an undecidable problem. While different static alias analysis implementations exist that offer a subset of context-, flow-, type- and field-sensitive information [DMM98; EGH94; Jeo+17; LLA07; SH97; Ste96], any potential may-alias still introduces spurious dependences that might prohibit transformations. Since aliasing in programs, especially SCoPs, almost never manifests at runtime [ADT13; Alv+15; DGH17; FE02; Guo+06], it is reasonable to guard optimistically optimized code regions with a check that rules aliasing out dynamically (ref. Section 3.5). Different implementations fur such runtime alias checks exist. In LLVM, simple innermost loops with static control flow and affine access function can be guarded to allow vectorization. Similarly, Bernstein, Cohen, and Maydan [BCM94] proposed dynamic dependence checks for arrays accesses but they restrict themselves to innermost loops. Alves et al. [Alv+15] use a variety of techniques, including the one described here. In addition, they propose a runtime environment that tracks memory allocations to allow dynamic aliasing queries. While accesses with data-dependent base pointers (ref. Figure 4.8c) are still out of reach, their system could allow polyhedral optimizations in the presence of non-affine and data-dependent access offsets (ref. Figure 4.8a and 4.8b).

Guo et al. [Guo+06] describe how to reduce the number of required comparisons (ref. Figure 4.2b) when the base pointer addresses are known, e.g., constant addresses in binaries.

Runtime alias checks can be used for a variety of reasons. Liu, Gorbovitski, and Stoller [LGS09] employ them to determine if cached results might be invalidated by potentially aliasing accesses. Wu et al. [Wu+13] describe how to use runtime checks to discover incorrect alias analysis results and Guo et al. [Guo+06] introduce them to allow instruction rescheduling in dynamic binary translation. Additionally, there are various approaches that utilize speculative execution as an alternative to classical runtime checks. Huang, Slavenburg, and Shen [HSS94] use code duplication to break dependence chains caused by insufficient alias information. Performance is gained if instruction level parallelism in combination with speculative or conditional execution can execute the code versions simultaneously. Ahn, Duan, and Torrellas [ADT13] propose an extension to the transactional memory modern hardware features to allow low-cost alias detection at runtime. Similarly, Gallagher et al. [Gal+94] introduce a new hardware memory conflict buffer to detect misspeculation at runtime. Fernández and Espasa [FE02] describe several heuristics to speculatively answer alias queries, however they do not provide a check and recovery system to ensure correctness. In contrast to all these local optimization and verification schemes, we need to exclude aliasing for a sequence of loop nests prior to the execution. If we would recognize aliasing accesses only when they happen, schedule changes could have caused memory effect that are invalid in the presence of aliasing. Nevertheless, under the assumption that aliasing will almost never manifest, a low-cost dynamic check accompanied by a powerful recovery system could be an alternative to our runtime checks. Hardware and software transactional memory are a promising candidate for such a recovery system but they often lack either performance or the necessary granularity for programs with lots of array accesses [Ham+16].

4.1.6 Evaluation

To evaluate the applicability impact of our runtime alias checks we performed two experiments and show the results in Table 4.10. In both experiments we determined the effect of missing runtime alias checks compared to the baseline presented in Figure 3.3 and Table 3.4 on Page 26 and 27. For the first experiment (ref. Table 4.10a) we allowed potentially alias pointers to be contained in a SCoP but then ignored such SCoPs for the metrics. In the second (ref. Table 4.10b) we disallowed potentially aliasing pointers all together. The differences stem from the greedy and cost oblivious SCoP detection. For a more detailed explanation see Section 3.6.3 and for assumption overhead and misspeculation rates see Section 3.5.5.

The results of both experiments indicate that runtime alias checks provide a significant applicability improvement. The number of valid SCoPs that do not contain potentially aliasing array accesses is below 50% in all four benchmark suites and it is less than a quarter in SPEC2006. Even though our implementation comes with certain limitations, the rejection reason statistics collected with our baseline version (ref. Table 3.19) indicate that the severeness of aliasing decrease drastically. Before the runtime alias checks were available, 68.9% of all rejected regions contained potentially aliasing pointers (ref. Table 3.16). Afterwards, only 18.7% of all rejected regions that did include a loop were rejected because aliasing could not be ruled out.

The monotone applicability scores (ref. Section 3.1.1) decrease roughly the same as the number of SCoPs, though the latter percentage is often slightly higher. This indicates that SCoPs suffering from aliasing are on average smaller than the ones that do not. Nevertheless, the score reduction of 33.2% - 60.0% is a strong argument for the use of runtime alias checks when polyhedral optimizations are applied to general purpose code.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	36.4%	21.2%	44.1%	41.2%
# depth 1 SCoPs	34.7%	14.7%	44.1%	30.1%
# depth 2 SCoPs	66.7%	57.6%	40.4%	62.3%
# depth 3 SCoPs	0%	100.0%	85.7%	38.5%
# depth 4 SCoPs	n/a	n/a	n/a	0%
C_0 score	36.8%	26.3%	44.3%	45.6%
C_1 score	40.0%	60.0%	46.3%	54.9%

(a) Relative results when potentially aliasing pointers were allowed but we only considered SCoPs that did not contain them for the metrics.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	44.5%	23.4%	41.9%	45.6%
# depth 1 SCoPs	44.9%	16.8%	41.9%	31.0%
# depth 2 SCoPs	44.4%	60.6%	39.4%	68.8%
# depth 3 SCoPs	33.3%	100.0%	85.7%	65.4%
# depth 4 SCoPs	n/a	n/a	n/a	0%
C_0 score	44.0%	28.6%	42.2%	52.3%
C_1 score	40.0%	62.9%	45.4%	66.8%

(b) Relative results when potentially aliasing pointers were disallowed during SCoP detection.

Table 4.10: Results to show the applicability impact of missing runtime alias checks. The numbers are percentages of the baseline results shown in Figure 3.3 and Table 3.4 on Page 26 and 27.

4.2 Representation of Integer Arithmetic

Because the polyhedral model is based on on Presburger arithmetic over the space $\mathbb{Z}$ of integers [Fea91; Len93], all computations are conceptually evaluated *precise*. Hence, expressions in control flow conditions and access relations are evaluated with arbitrary precision over the entire range of $\mathbb{Z}$. However, programming languages have to consider machine constraints to allow for performant programs. *Precise* expression evaluation is too resource intensive and slow for everyday use in general purpose programs. Programming languages do consequently impose evaluation semantics that can be implemented efficiently on machines while *precise* semantics is only available through specialized libraries [Fea88b; Pug91a; Ver10]. The most often employed alternative is *wrapping* semantics which computes the result modulo the maximal value representable with the desired bit width. Due to the two's complement representation of signed integers in modern machines, *wrapping* semantics can be implemented by simply discarding all but the least significant bits. This defined integer wrapping can often hinder optimizations as their implementation relies on basic arithmetic properties, e.g., monotonicity of addition or multiplication with a constant. To overcome this issue, low-level languages also use *error* semantics which causes undefined behavior if the result of a computation does not fit in the number of associated bits. *Error* semantics is consequently only defined if *precise* and *wrapping* evaluation compute the same result. Hence, *error* semantics can be implemented with both *precise* and *wrapping* semantics but it also allows for more optimizations.

4.2.1 Related Work

Integer wrapping is a reoccurring source of security and reliability problems [Ahm03; Die+12; Wan+09]. To this end, static analyses are used to identify potentially or definitively wrapping expressions [CH13; Cou+05; RCP13; Wan+09; Zha+15]. Analyses are often augmented with instrumentation, i.e., dynamic checks, to ensure complete results [Bru+07; Che+09; Die+12]. Long et al. [Lon+14] derive input filters to prevent integer overflows which are similar to our expression evaluation assumptions described in Section 4.2.4. While they completely give up on control flow constraints in favor of performance, we use iteration domains to tighten and simplify the assumptions that check for potentially wrapping expressions at runtime.

Based on the seminal work by Cousot and Halbwachs [CH78], Simon and King [SK07] and later Bygde, Lisper, and Holsti [BLH12] describe how integer overflows impact abstract values in a

> The representation of integer arithmetic was part of our efforts to make polyhedral optimization automatically applicable and sound on general purpose programs [DGH17]. Partially precise expression evaluation (ref. Section 4.2.6), as well as expression evaluation assumptions (ref. Section 4.2.4), are both integrated and enabled in LLVM/POLLY.

polyhedral domain without quasi affine functions. In addition, there are multiple precise relational program analyses that take the bit width into account [Eld+14; KS10; MS04]. These analysis derive context-insensitive affine equalities that hold in the presence of machine arithmetic. Due to the inevitable complexity increase to add inequalities, and given the almost non-existence of integer overflows in real-world loop nests [DGH17], we believe an optimistic representation, safeguarded by assumptions, is the most practical option to deal with potentially wrapping expressions in low-level programs.

The polyhedral extraction tool (PET) [VG12] precisely models integer wrapping of unsigned C expressions. However, our experiments show that wrapping almost never occurs in practice and is consequently not worth the increased cost that comes with an exact representation [DGH17]. Alternatively, we optimistically assume all expressions are evaluated with *error* semantics (ref. Table 4.11). To ensure correctness, we restrict the possible inputs to parameter valuations that do not cause control flow conditions or access relations to wrap. This optimistic, assumption-based modeling allows for a concise representation, without spurious dependences, that is very unlikely to cause misspeculation at runtime.

4.2.2 Integer Evaluation Semantics

Three integer evaluation semantics relevant for polyhedral optimization of low-level programs are illustrated in Table 4.11. The first column shows the name of a semantics and the second defines the result if the semantics is used to evaluate an expression `e`. In the last column common usages are listed. Note that programming languages employ different semantics based on the signedness of types, e.g., in C/C++, or computation attributes, i.e., **`nsw/nuw`** in LLVM-IR[5].

Semantics	Value of `e`	Usage
precise	$\llbracket \mathtt{e} \rrbracket_p := \llbracket \mathtt{e} \rrbracket_{\mathbb{Z}}$	PipLib [Fea88b], Omega [Kel+95; Pug91a], isl [Ver10]
wrapping	$\llbracket \mathtt{e} \rrbracket_w := \llbracket \mathtt{e} \rrbracket_{\mathbb{Z}/n\mathbb{Z}}$	Java, Julia, C/C++ (unsigned types), LLVM-IR (without **`nsw/nuw`**)
error	$\llbracket \mathtt{e} \rrbracket_e :=$ if $\llbracket \mathtt{e} \rrbracket_p = \llbracket \mathtt{e} \rrbracket_w$ then $\llbracket \mathtt{e} \rrbracket_p$ else *undef*	C/C++ (signed types), LLVM-IR (with **`nsw/nuw`**)

Table 4.11: Different expression evaluation semantics and examples for their usage. The value of n is the bit width of the expression `e`. Note that the range of *wrapping* semantics is shifted by 2^{n-1} for signed numbers.

[5] The LLVM-IR attributes **`nsw/nuw`** are short forms of "no signed/unsigned wrap" and can be attached to operations that might cause integer overflows, e.g., additions and multiplications, to indicate hat they will not.

```
for (i = 0; i <= N; i += 1)
  A[2 * i] = A[2 * i + 1];
```

Figure 4.12: Example loop[a] with dependences only if *wrapping* semantics is used. If i is an 8-bit unsigned value, a loop-carried dependence is present for N = 2^7 = 128.

[a] This Figure was taken from Doerfert, Grosser, and Hack [DGH17].

To obtain a sound polyhedral representation of low-level inputs it is crucial to consider the expression evaluation semantics. While it is possible to express *wrapping* semantics in Presburger arithmetic [VG12], our experience shows that it has a negative effect on compile time as well as runtime of the generated code [DGH17]. Compile time is increased due to the additional existentially quantified dimensions that modulo expressions generally introduce. The generated code is more complex due to the additional dependences that are only present in case of wrapping but almost never actually appear in practice. An example for a loop that is constrained by loop carried dependences if and only if *wrapping* semantics is used is shown in Figure 4.12. In general, programmers do not employ integer wrapping intentionally to implement part of the program semantics. Instead, potential wrapping and the dependences that come with it are most often only a byproduct of the poor flexibility programming languages offer when it comes to integer arithmetic or potentially unawareness on the programmers side [Ahm03]. That said, it is still crucial to generate correct code for corner case inputs that trigger integer under- or overflows, especially in an automatic approach used on general purpose code. If the polyhedral representation does not take these inputs into account, silently miscompilations are inevitable and likely followed by data corruption or a program crash.

4.2.3 Potentially Wrapping Expressions

```
uint64_t a, b, c, d;
uint8_t s;
a = b = ...;
c = a + b;
d = c * c;
s = (uint8_t) d;
```

Figure 4.13: Example computations that could cause an integer under- or overflows in each of the last three lines.

There are four conceptual types of potentially wrapping expressions: addition, multiplication, truncation, and division by -1. A minimal example featuring three of them is shown in Figure 4.13. The addition of two n-bit values can create an $n + 1$-bit wide result. Similarly, a multiplication of two n-bit values can create a $2n$-bit result. In either case the outcome is truncated to fit into the resulting type which is generally required to be the same as the operand type. Since all three operations will consider only the lower bits of a value, the most significant bit, thus the sign in the two's complement representation, can potentially change.

In LLVM, the result of an operation that can potentially wrap might be well-defined if it does. To determine if that is the case, thus to distinguish between *wrapping* and *error* semantics, we check for the "no-signed-wrap (`nsw`)" flag[6]. Operations that carry this flag are, similar to computations

[6] As explained in Section 4.3, LLVM/POLLY interprets all values as signed integers, thus `nsw` flags are considered.

on signed values in C/C++, are evaluated with *error* semantics. They are consequently not allowed to cause a signed overflow. Thus, we can assume that the truncation at the end of the operation will not change the signed bit as it would otherwise cause undefined behavior that allows us to model the result as any value. Due to the indirection LLVM/POLLY uses to simplify the generation of a polyhedral representation (ref. Section 6.4), the `nsw` flags present in the source code might be hidden. In this case, more computations might be considered potentially wrapping and needless expression evaluation assumptions will be generated.

4.2.4 Expression Evaluation Assumptions

The expression evaluation assumptions Λ_{EE} constrain the allowed parameter valuations in order to prevent wrapping control flow conditions and access expressions. To identify them, each expression that influences the polyhedral representation and that could potentially wrap is translated twice, once with *precise* and once with *wrapping* semantics. Given such an expression $\texttt{e}$, we use $[\![\texttt{e}]\!]_p$ to denote the former translation and $[\![\texttt{e}]\!]_w$ for the latter. Both translate $\texttt{e}$ to a piecewise defined affine function over the surrounding iteration variables and program parameters. Assuming $\texttt{e}$ is contained in a statement $\texttt{S}$, we can compute the set of all iterations of $\texttt{S}$ for which $\texttt{e}$ would wrap as shown in Formula 7.

$$\{\mathbf{i} \in \mathcal{D}_{\texttt{S}} \mid [\![\texttt{e}]\!]_p(\mathbf{i}) \neq [\![\texttt{e}]\!]_w(\mathbf{i})\} \tag{7}$$

A different but equivalent solution to detect iterations that do not cause an n-bit expression $\texttt{e}$ to wrap is shown in Formula 8. It is implemented in PET to avoid the representation of iterations that will cause undefined behavior due to wrapping. Here, the precise representation is compared against the maximal values of the result explicitly. However, a comparison of the two implementations in LLVM/POLLY showed a slightly better compile time for the version shown in Formula 7 than for the one used by PET.

$$\{\mathbf{i} \in \mathcal{D}_{\texttt{S}} \mid -2^{n-1} \leq [\![\texttt{e}]\!]_p(\mathbf{i}) < 2^{n-1}\} \tag{8}$$

To obtain the evaluation assumptions $\Lambda_{\text{EE}}(\texttt{e})$ for the expression $\texttt{e}$ we have to isolate parameter valuations that do not cause wrapping. To this end, we first project all wrapping iterations computed in Formula 7 onto the parameter space ρ as shown in Formula 9.

$$\pi_\rho(\ \{\mathbf{i} \in \mathcal{D}_{\texttt{S}} \mid [\![\texttt{e}]\!]_p(\mathbf{i}) \neq [\![\texttt{e}]\!]_w(\mathbf{i})\}) \tag{9}$$

The result does not depend on the surrounding loop variables but only on parameters of the SCoP. For each contained parameter valuation there is at least one iteration of $\texttt{S}$ for which $\texttt{e}$ does wrap. Thus, it is empty if and only if there is no iteration of $\texttt{S}$ in which $\texttt{e}$ will under-or overflow. The complement of the set shown in Formula 9 describes the expression evaluation assumptions

$\Lambda_{\mathrm{EE}}(\mathtt{e})$. The absence of integer wrapping and thereby a correct polyhedral representation is ensured under the expression evaluation assumptions of all control flow conditions and access expressions. If we denote all such expressions contained in the SCoP as $\langle exp \rangle$, we can compute Λ_{EE} as shown in Formula 10.

$$\Lambda_{\mathrm{EE}} \coloneqq \bigcap_{\mathtt{e}\in\langle exp\rangle} \Lambda_{\mathrm{EE}}(\mathtt{e}) = \bigcap_{\mathtt{e}\in\langle exp\rangle} \neg\,\pi_p(\,\{\mathbf{i}\in\mathcal{D}_{\mathtt{S}} \mid [\![\mathtt{e}]\!]_p(\mathbf{i}) \neq [\![\mathtt{e}]\!]_w(\mathbf{i})\}) \qquad \textbf{(10)}$$

4.2.5 Textual vs. Polyhedral Location of Potentially Wrapping Expressions

It is important to note that the optimistically generated polyhedral representation of the input program is not necessarily sufficient to compute the expression evaluation assumptions Λ_{EE}. Due to referential transparency of expressions in the polyhedral model, it is possible that values are replaced by their definition, thus altering the domain under which an expression is evaluated. In Figure 4.14 two programs are shown that are equivalent if expressions are evaluated with *precise* semantics but not necessary with *wrapping* semantics. While the polyhedral representation of the loops in Figure 4.14a and 4.14b can be equal (Figure 4.14c) the former might exhibit a wrapping increment while the latter does not. Consequently, it is necessary to utilize the textual expression and the textual location to compute Λ_{EE}, not a polyhedral representation thereof.

```
     for (i = p + 1; i < 10; i += 1)
S:    A[i - 1] = A[i - 1] + 1;
```

(a) Possibly wrapping increment of `p` in the initialization of `i`.

```
     for (i = p; i < 9; i += 1)
S:     A[i] = A[i] + 1;
```

(b) Non-wrapping increment of `i` guarded by the loop bound.

$$\mathcal{D}_{\mathtt{S}} = \{\mathbf{i} \mid 0 \leq \mathbf{i} \leq 8 - p\} \qquad f_{\mathtt{S}} = \{\mathbf{i} \to (\mathtt{A}, (\mathbf{i} + p))\}$$

(c) Possible polyhedral representation of the loops shown in part 4.14a and 4.14b.

Figure 4.14: Two example loops shown[a] in part 4.14a and 4.14b with equal polyhedral representation, shown in part 4.14c, but different wrapping behavior.

[a] This Figure was taken from Doerfert, Grosser, and Hack [DGH17].

4.2.6 Explicit Representation of Integer Arithmetic

While our experience and experimental data [DGH17] indicate that most programmers do not exploit integer wrapping, it is however utilized by compiler passes or programming experts. The loop nest in Figure 4.1 shows how a compiler pass reduced the complexity of the conditional expression using an intentional underflow that needs to be represented precisely in the polyhedral model. If not, the expression evaluation assumptions Λ_{EE}, and thereby the runtime check, will inevitably fail and the original code version has to be executed. While the behavior therefore remains sound, the optimization only wasted compile and execution time.

The challenge is to determine the intention from the code, thus if integer wrapping is the intended semantics or if it is unintentional and will not be triggered by common inputs. As this is generally undecidable, we introduced a heuristic to determine if precise modeling might be beneficial and cause only moderate overhead. The heuristic is derived from programs that caused our assumptions to fail. The examples in Figure 4.15 show such a case where *wrapping* semantics is needed. The SCALAR EVOLUTION analysis [BWZ94; PCS05], which is used to derive polyhedral value representations (ref. Section 6.4), models the if conditions always as an additive recurrence, thus after the behavior of the scalar variable c in Figure 4.15c. This one bit counter overflows intentionally every second iteration to alternate between *false* (0) and *true* (1).

```
for (i = 0; i < N; i++)
  if (i % 2 == 0)
    S(i);
  else
    P(i);
```

(a) Conditional with an explicit modulo computation.

```
for (i = 0; i < N; i++)
  if (i & 1 == 0)
    S(i);
  else
    P(i);
```

(b) Conditional with an implicit modulo computation using a bitwise operation.

```
bool c = 0; // 1-bit
for (i = 0; i < N; i++)
  if (c++ == 0)
    S(i);
  else
    P(i);
```

(c) Conditional with an implicit modulo computation using the wrapping boolean c.

Figure 4.15: Three different but equivalent versions of a loop. In all versions the statements S and P are executed in an alternating fashion.

Since all three examples in Figure 4.15 are equivalent, we also want their polyhedral representation to be. However, the overflow in part Figure 4.15c would cause the conservative expression evaluation assumption $N < 3$. This assumption restricts the optimization to factitious inputs and the alternating behavior of the code is missed. Alternatively, we can precisely represent the computation involving the variable c, thus employ *wrapping* semantics for the increment. This is reasonable as a one-bit variable was very likely intended to wrap. Similarly, we model wrapping on all small integer types with less than 8 bits. We assume that such computations have been placed by the compiler as the smallest type available to programmers is generally 8 bits long.

The final expression evaluation function is shown in Formula 11. It takes an expression e and generates the polyhedral representation as well as a set of assumptions that guarantee correctness.

$$[\![\mathtt{e}]\!] := \begin{cases} \big([\![\mathtt{e}]\!]_w,\ \{\}\big) & \text{if } \operatorname{bits}(\mathtt{e}) < 8 \\ \big([\![\mathtt{e}]\!]_p,\ \Lambda_{\text{EE}}(\mathtt{e})\big) & \text{otherwise} \end{cases} \qquad \textbf{(11)}$$

Our initial implementation did only use explicit wrapping for small bit-width expressions that were zero extended to a larger type (ref. Section 4.3). While the small bit-width constraint is still enforced, a later patch[7] introduced this logic for computations that do not involve type extensions.

[7] Patch by Eli Friedman and review discussion are available online: https://reviews.llvm.org/D25287

4.2.7 Evaluation

To determine the applicability effect of expression evaluation assumptions on the benchmarks suites described in Section 2.3, we performed two experiments. The results, shown in Table 4.16, are relative percentages of the baseline values presented in in Figure 3.3 and Table 3.4 on Page 26 and 27. Note that assumption overhead and misspeculation rates are discussed in Section 3.5.5.

While the number of SCoPs and the monotone applicability scores are lower across all benchmark suites, the effect is most significant for SPEC2006 and SPEC2017. If potentially wrapping expressions are disallowed (ref. Table 4.16b), the number of SCoPs decreased by 88.5% for SPEC2017. The applicability score for $\alpha = 1$, thus the number of affine loops that can be combined with a fixed one (ref. Section 3.1.1), is down by 89.8%. As this score is always decreased further than the number of SCoPs, we know that mostly SCoPs with multiple loops did contain potentially wrapping expressions. Non-nested loops, thus depth 1 SCoPs, are least affected.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	37.3%	20.8%	9.0%	31.7%
# depth 1 SCoPs	37.8%	20.8%	8.6%	29.4%
# depth 2 SCoPs	11.1%	21.2%	13.8%	27.5%
# depth 3 SCoPs	100.0%	0.0%	0.0%	73.1%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	38.4%	20.7%	9.3%	34.6%
C_1 score	46.7%	20.0%	12.0%	40.9%

(a) Relative results when potentially wrapping expressions were allowed during the SCoP detection but SCoPs that contained them were ignored when the metrics were evaluated.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	40.9%	20.8%	11.5%	25.1%
# depth 1 SCoPs	42.9%	21.3%	11.6%	31.7%
# depth 2 SCoPs	11.1%	18.2%	11.7%	15.2%
# depth 3 SCoPs	66.7%	0%	0%	15.4%
# depth 4 SCoPs	n/a	n/a	n/a	0%
C_0 score	40.0%	20.3%	11.3%	22.0%
C_1 score	33.4%	17.1%	10.2%	15.0%

(b) Relative results when potentially wrapping expressions were considered non-affine.

Table 4.16: Impact of potentially wrapping expressions on the applicability or POLLY. The results are relative to the baseline values shown in in Figure 3.3 and Table 3.4 on Page 26 and 27.

4.3 Reconciliation of Signed & Unsigned Values

Programming languages differentiate between signed and unsigned variables to extend the value range and, in case of C/C++, to allow for different integer wrapping semantics. Program representations, like the polyhedral model, consequently have to deal with signdness as well. In the nowadays ubiquitous two's complement binary representation the signedness of a variable will however not influence its bit representation in the shared value range. The example code sequence in Table 4.17 illustrates this for different signed and unsigned variables. In addition to their bit representation, their value in both signed and unsigned interpretation, denoted as $\llbracket \cdot \rrbracket^s$ and $\llbracket \cdot \rrbracket^u$, is given. The increments in line 3 and 4 illustrate that addition does not differentiate between the two interpretations. In fact, this is true for all common bit-wise and arithmetic operations except comparisons, divisions, right shifts, and type extensions. To this end, low-level code representations, e.g., assembly languages like LLVM-IR, do not associate signedness with a bit string but instead with an operation, if the result depends on the operands interpretation.

	Code	Bit Repr.	Signed Interpr.	Unsigned Interpr.
1.	`int8_t  s = 2 - 127;`	`0b1000011`	$\llbracket \mathtt{s} \rrbracket^s = -125$	$\llbracket \mathtt{s} \rrbracket^u = 131$
2.	`uint8_t u = 4 - 129;`	`0b1000011`	$\llbracket \mathtt{u} \rrbracket^s = -125$	$\llbracket \mathtt{u} \rrbracket^u = 131$
3.	`int8_t  t = s + 1;`	`0b1000100`	$\llbracket \mathtt{t} \rrbracket^s = -124$	$\llbracket \mathtt{t} \rrbracket^u = 132$
4.	`uint8_t v = u + 1;`	`0b1000100`	$\llbracket \mathtt{v} \rrbracket^s = -124$	$\llbracket \mathtt{v} \rrbracket^u = 132$

Table 4.17: Signed and unsigned uses of the same bit pattern and the associated values in both representations. Note that both values are equal modulo the maximal representable value, here $2^8 = 256$, as explained in Section 4.3.3.

4.3.1 Related Work

Type conversion that change the signedness and value of a variable are often considered programming errors due to security concerns [Sun+15]. To this end, various approaches have been devised to detect these signedness errors. As other integer arithmetic issues (ref. Section 4.2), signedness errors generally depend on input data. Thus, instrumentation is often needed to detect them at runtime. The first approaches were complete in the sense that they reported all errors at runtime [Bru+07; Dan+10; Die+12]. However, follow up work by Pomonis et al. [Pom+14] and Sun et al. [Sun+15] introduces a differentiation between benign and potentially problematic signedness errors. The former is identified by static analysis and only the latter is instrumented to reduce the overhead and false positive rate. While their definition of benign signedness errors

The reconciliation of signed and unsigned value interpretations is an extension to our work on optimistic loop optimizations [DGH17]. It is integrated in LLVM/POLLY to allow inputs that use unsigned interpretation for control flow conditions or access functions.

is different from ours, we also identify only certain signedness problems, namely in control flow conditions and access functions. In contrast to these instrumentation schemes we generalize the conditions under which signedness errors can occur. This allows to create a single check prior to the SCoP (ref. Section 3.5) instead of a dedicated check for each potential error source. There are also pure static techniques to detect [Cou+05; Kan12] and repair [Che16; Che+17] problematic source code. Similarly, Coker and Hafiz [CH13] use a user-guided C source code analysis to detect potential overflow, truncation and signedness bugs. To remove these security vulnerabilities they introduce overflow tracking arithmetic, change the variable types or use explicit casts.

The sign-agnostic interval analysis presented by Gange et al. [Gan+14] is based on the superposition of signed and unsigned interpretation. This is similarly to the dual representation we propose in Figure 4.3.5 as an alternative for the current polyhedral modeling of low-level inputs.

4.3.2 Signed and Unsigned Interpretation of Values

The three signedness aware operations in LLVM-IR that are important for polyhedral modeling are: comparisons, divisions and type extensions. While our implementation is focused on the correct representation of these three operations, we will use C code examples and type casts to increase readability. Even if not all low-level situations can be accurately translated to such “high-level” code, they are generally sufficient to illustrate the problems. In addition, it shows that correct and efficient reconciliation of signed and unsigned interpretation is not dependent on the association of signedness to either values (C/C++) or operations (LLVM-IR).

```
for ( int8_t i = 0; i < ( int8_t) N; i++)
  S(i);
for (uint8_t i = 0; i < (uint8_t) N; i++)
  P(i);
```

Figure 4.18: Two loops that use N as an upper bound but interpret it once as a signed and once as an unsigned value. If this semantic difference is not taken into account, any value for N that is not equal for both interpretations will cause an incorrect iteration domain for one of the statements.

For a consistent polyhedral program representation with respect to signedness it is crucial to interpret values depending on their use. However, LLVM/Polly interprets all values as signed (ref. Figure 4.3.5). In order to ensure a correct polyhedral model, unsigned value uses have to be considered invalid or reconciled with the otherwise signed interpretation. An explanatory example is given in Figure 4.18. If the two loop bounds are translated without the appropriate adjustment for signed and unsigned interpretation, the iteration domains of statement S and P would be equal. Though, for values of N that differ in the two interpretations, thus if the signed bit is set, only the iteration domain of statement S should be empty. Due to signed interpretation of the second loop bound, the number of iterations in the domain was determined to be $\llbracket \texttt{N} \rrbracket^s$ instead of $\llbracket \texttt{N} \rrbracket^u$, hence $2^8 = 256$ iterations too few. Consistently, an unsigned interpretation of the first bound will increase the iteration count of statement S by 256 if the signed bit of N is set.

4.3.3 Transitioning Between Signed and Unsigned Value Representations

Assuming two's complement representation, the difference of signed or unsigned interpretation is only the sign of the most significant bit. If this bit is not set, the interpretations do not differ. If it is set, the signed and unsigned interpretation of an n-bit string differ by exactly 2^n. To transition between interpretations half of the value space is moved as illustrated in Figure 4.19.

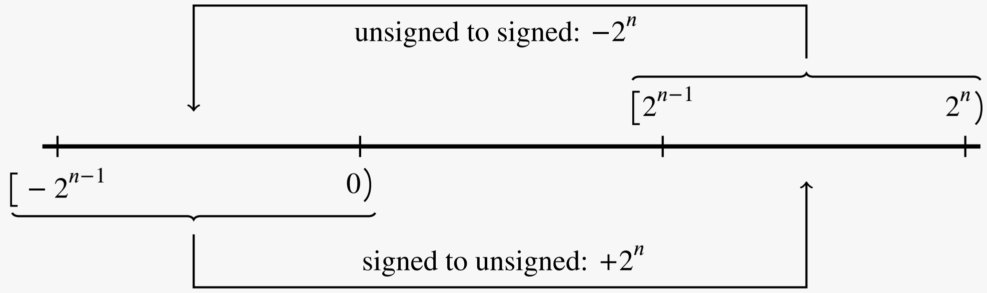


Figure 4.19: Graphical illustration of the transition between a signed and unsigned interpretation of an n-bit value. If the interpretation was signed, each value in the lower half of the value range, thus $[-2^{n-1}, 0)$, is increased by 2^n. If the interpretation was unsigned the values in the upper half, thus $[2^{n-1}, 2^n)$, are decreased by 2^n.

To translate between signed and unsigned interpretation in the polyhedral model we have to determine the affine transition functions s2u and u2s such that

$$\mathrm{s2u}\left(\llbracket x \rrbracket^s\right) := \llbracket x \rrbracket^u$$

and

$$\mathrm{u2s}\left(\llbracket x \rrbracket^u\right) := \llbracket x \rrbracket^s.$$

In Formula 12 and 13 their implementation as piecewise defined affine functions is shown.

$$\mathrm{s2u}(x) = \begin{cases} x & \text{if } x \geq 0 \\ x + 2^n & \text{otherwise} \end{cases} \quad \textbf{(12)}$$

$$\mathrm{u2s}(x) = \begin{cases} x & \text{if } x < 2^{n-1} \\ x - 2^n & \text{otherwise} \end{cases} \quad \textbf{(13)}$$

Alternative, quasi affine versions are provided in Formula 14 and 15.

$$\mathrm{s2u}(x) = \left(x + 2^n\right) \bmod 2^n \quad \textbf{(14)}$$

$$\mathrm{u2s}(x) = \left(\left(x + 2^{n-1}\right) \bmod 2^n\right) - 2^{n-1} \quad \textbf{(15)}$$

The correctness of the piecewise defined versions follows from the graphical illustration in Figure 4.19. To prove the quasi affine functions correct we can distinguish two cases. First we assume the value of x is equal in both interpretations, thus $x \in \left[0, 2^{n-1}\right)$. The s2u function in Formula 14 will first increase the value of x by 2^n such that it is between $\left[2^n, 2^n + 2^{n-1}\right)$. However, the modulo operation will then reverse this increase and yield the original value of x. For the unsigned to signed transition, hence u2s in Formula 15, the value is increased and decreased by 2^{n-1} also yielding the original input. Note that the modulo is not affecting the intermediate result as x is between $\left[2^{n-1}, 2^n\right)$ after the increment. For the second case we have to determine the effects if the most significant bit of x is set, thus if $x < 0$ or $x \geq 2^{n-1}$. Here the signed interpretation of x has to be in $\left[-2^{n-1}, 0\right)$ and the unsigned interpretation in $\left[2^{n-1}, 2^n\right)$. The signed to unsigned conversion of the former will only increase the value to the range $\left[2^{n-1}, 2^n\right)$ as the modulo operation is not affecting it. For the unsigned to signed transition the modulo operation on the value of $x + 2^{n-1}$ will yield a result in $\left[0, 2^{n-1}\right)$. The final subtraction moves this range down to $\left[-2^{n-1}, 0\right)$. Note that for each of the two cases all operations did preserve the ordering of the values for each of the considered intervals. Thus, if $x < y$ and both x and y are in $\left[-2^{n-1}, 0\right)$ or alternatively in $\left[0, 2^{n-1}\right)$, then $\text{s2u}(x) < \text{s2u}(y)$. Similarly, if $x < y$ and both x and y are in $\left[0, 2^{n-1}\right)$ or alternatively in $\left[2^{n-1}, 2^n\right)$, then $\text{u2s}(x) < \text{u2s}(y)$.

	Code	**Signed Interpretation**
1.	`int8_t  N = ...;`	N
2.	`int8_t  s = N - 15;`	$N - 15$
3.	`uint8_t u = s;`	$\begin{cases} N - 15 & 0 \leq N - 15 \\ N + 241 & \text{otherwise} \end{cases}$
4.	`uint8_t v = u + 4;`	$\begin{cases} N - 11 & 0 \leq N - 15 \\ N + 245 & \text{otherwise} \end{cases}$
5.	`int8_t t1 = v;`	$\begin{cases} N - 11 & -117 \leq N \leq 138 \\ N - 276 & 139 \leq N \\ N + 245 & \text{otherwise} \end{cases}$
6.	`int8_t t2 = v;`	$N - 11$ *(simplified)*

Table 4.20: Signed polyhedral representation of different signed and unsigned values. Note that the last representation was simplified using the possible range of `N` and the fact that the signed subtraction is not allowed to underflow.

The example code sequence in Table 4.20 illustrates how the polyhedral representation is changed when values transition between interpretations. The code is shown on the left and the signed interpreted polyhedral value on the right. In line 3, the signedness of `s` is changed. This alters the value depending on the signed bit. If it is not set, hence if $N - 15 \geq 0$, the value of `s` is equal in both interpretations. Otherwise, the value has to be adjusted to account for the different interpretation of the signed bit which increases it by $2^8 = 256$. The addition in line 4 will increment both pieces of the polyhedral representation. With the switch to a signed interpretation in line 5, the polyhedral representation will distinguish three cases in total. However, it converges to the single piece shown in line 6 if we employ context information. First note that the type of N allows to infer its value range, here $-128 \leq N < 128$. This range can be refined due to the signed underflow that is not allowed to happen in line 2, thus we can conclude that $N \geq -113$.

4.3.4 Unsigned Representation Assumptions

```
int32_t N = ...;
for (i = 0; i < (uint32_t) N; i++)
  S(i);
```

Figure 4.21: Example loop that will exhibit more than 2^{31} iterations if the value of N was initially negative.

While the polyhedral model allows precise interpretation of signedness, it is, similar to precise integer arithmetic, often not preferable. For an example consider Figure 4.21. If modeled precisely, the iteration domain for statement S has two pieces. One describes the case where N was negative prior to the cast, the other the case where it was not. For the latter case the loop has N iterations but for the former it iterates at least 2^{31} times. If each iteration touches only one new `float` memory cell, hence 4 distinct bytes, the loop would iterate through more than 8.5 gigabytes of data. As this is unlikely to happen for general purpose code we introduce assumptions that prevent the need for such enormous constants in control flow conditions and access functions. Instead of a precise transition between signed and unsigned interpretation we derive unsigned representation assumptions Λ_{UR} that imply equal interpretations. Thus, n-bit control flow conditions and access functions that are at some point interpreted as unsigned will be required to be in the range $\left[0, 2^{n-1}\right)$. If we denote these expressions as $\langle uexp \rangle$, the unsigned representation assumption ensure that:

$$\forall \mathtt{e} \in \langle uexp \rangle : \Lambda_{\text{UR}}(\mathtt{e}) \implies [\![\mathtt{e}]\!]^{s} = [\![\mathtt{e}]\!]^{u}$$

To prevent conservative assumptions (ref. Section 4.2.6) we can alternatively translate unsigned to signed interpretations instead. This is done for the constant divisor of an unsigned divisions as well as small bit-with type extensions. For all other unsigned uses of values we will determine the parameter valuations that will prevent a difference in the interpretations. To this end, all parameter combinations that result in a negative value for the signed interpretation of a value $\mathtt{e} \in \langle uexp \rangle$ are first isolated and then negated. Note that this is only necessary for the iterations in which e is executed. If e is contained in statement S we check for which iterations $\mathbf{i} \in \mathcal{D}_{\mathtt{S}}$ the signed and unsigned representations would differ, thus the signed interpretation is negative. The definition of unsigned representation assumptions Λ_{UR} is shown in Formula 16.

$$\Lambda_{\text{UR}} := \bigcap_{\mathtt{e} \in \langle uexp \rangle} \Lambda_{\text{UR}}(\mathtt{e}) = \bigcap_{\mathtt{e} \in \langle uexp \rangle} \neg \pi_{\rho}\left(\left\{ \mathbf{i} \in \mathcal{D}_{\mathtt{S}} \mid [\![\mathtt{e}]\!]^{s}(\mathbf{i}) < 0 \right\} \right) \qquad \textbf{(16)}$$

4.3.5 Limitations & Extensions

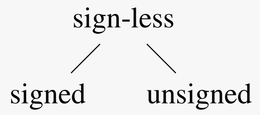


Figure 4.22: Weak ordering defined over the three signedness tags: sign-less, signed and unsigned.

To avoid complexity issues and assumptions, it would be ideal if signedness would influence the polyhedral representation only if absolutely necessary. Thus, only if a value is interpreted as both signed and unsigned in the analyzed code region or, to be more precise, if a value is interpreted as both signed and unsigned in one execution of the analyzed code region. However, to simplify the implementation, polyhedral optimizers like LLVM/POLLY do assume one representation as the default and require the other to be reconciled with this choice. An alternative, and arguably better, implementation would use a sign-agnostic interpretation by default and switch to an explicitly signed representation only if required. This implementation choice would be especially suited for low-level languages that feature only a few operations which are influenced by the signedness of the operands. Hence, the polyhedral representation of a value should be considered "signed-less" as long as it is not determined by an operation. As soon as it is, the value assumes the signedness of the operation without introducing any additional overhead, thus without complexity increase or assumptions[8]. Only if a value with an already specified signedness is used in an operation that requires a different interpretation, a conversion is performed. The conversion can then be done precisely, thus with a piecewise definition, or optimistically, hence with an assumption about the value. Conceptually, this can be seen as an additional tag attached to the polyhedral value representation such that the possible tags (sign-less, signed and unsigned) form the weak ordering shown in Figure 4.22.

A use case for the described principles is showcased by the templated function in Figure 4.23. An ideal implementation of a polyhedral tool would derive the same concise and assumption free representation regardless of the signedness of `DataTy` and `CtrTy`. However, further studies are needed to determine the actual benefit of such a more evolved representation scheme.

```
template<typename DataTy, typename CtrTy>
DataTy sum(DataTy *A, CtrTy N) {
  DataTy res = 0;
  for (CtrTy i = 0; i < N; i++)
    res += A[i];
  return res;
}
```

Figure 4.23: Example to showcase the usefulness of a sign-less polyhedral representation. If the signedness of a polyhedral value representation is determined on demand and not specified by default, the polyhedral representation for the shown function is not influenced by the instantiation of `DataTy` and `CtrTy`.

[8] Constants are a special case as they can be adjusted without assumptions or a piecewise representation.

4.3.6 Evaluation

To evaluate the reconciliation of signed and unsigned values we follow the same scheme before, e.g., in Section 3.3.4. The metric results shown in Table 4.24 are percentages of the baseline values presented in in Figure 3.3 and Table 3.4 on Page 26 and 27. For the upper part, SCoPs were allowed to contain unsigned interpreted values but we excluded those when the metrics were evaluated. For the lower part, SCoP detection forbid unsigned interpreted values.

While the number of SCoPs decreases in both experiments and across all benchmark suites, the second experiment shows an increase ($> 100\%$) in the monotone applicability score (ref. Section 3.1.1). This indicates that the representation of unsigned values causes more smaller and less larger SCoPs for the SPEC benchmarks. One reason is the complexity increase that comes with the unsigned representation, and which can cause valid SCoPs to be dropped due to compile time concerns (ref. Section 3.5.5.3). As this happened only for depth 2 SCoPs, we require the SCoP counts per depth, or alternative the monotone applicability score for $\alpha = 1$, to identify the problem. In all other metrics this trend would have gone unnoticed since depth one results overshadow it, e.g., the SCoP count is actually 7.3%–21.4% lower without unsigned value support.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	89.1%	86.1%	90.0%	89.4%
# depth 1 SCoPs	86.3%	86.3%	90.9%	90.1%
# depth 2 SCoPs	84.4%	84.8%	85.1%	88.4%
# depth 3 SCoPs	100.0%	100.0%	71.4%	88.5%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	87.2%	86.1%	89.8%	89.2%
C_1 score	73.3%	85.7%	83.3%	88.6%

(a) Relative results when SCoPs containing unsigned interpretations were omitted for the metrics.

Metric	SPEC2000	SPEC2006	SPEC2017	LLVM-TS
# SCoPs	82.7%	87.0%	90.4%	87.5%
# depth 1 SCoPs	78.6%	82.2%	89.0%	86.1%
# depth 2 SCoPs	122.2%	115.2%	107.4%	89.1%
# depth 3 SCoPs	100.0%	100.0%	85.7%	92.3%
# depth 4 SCoPs	n/a	n/a	n/a	100.0%
C_0 score	86.4%	90.6%	91.6%	88.4%
C_1 score	133.3%	114.3%	104.6%	90.2%

(b) Relative results when SCoPs were not allowed to contain unsigned interpreted values.

Table 4.24: Applicability results for unsigned value support. The numbers are percentages of the baseline results shown in Figure 3.3 and Table 3.4 on Page 26 and 27.

Chapter 5

Optimizations & Applications

> *The increases in demands on hardware and software will continue: human expectation grows even faster than hardware performance.*
>
> Bjarne Stroustrup, *Software Development for Infrastructure*, 2012

Chapter 3 described techniques to improve the applicability and robustness of polyhedral-model-based tools. Chapter 4 introduced ways to ensure correctness in the presence of common semantic mismatches between programming languages and the polyhedral model. This chapter puts these advances to work and describes the application and optimization through polyhedral tools.

One omnipresent complaint about polyhedral techniques is the *complexity*. In this context, complexity refers not only to the exponential compile time cost but also the usability for non-experts. To improve both, we present several enhancements that reduce different kinds of *complexity* in Section 5.1. One goal of these enhancements was to allow common users to comprehend and guide polyhedral optimizations by providing easy to understand feedback and the possibility to augment the information available to the optimizer. Afterwards, we present a polyhedral *reduction detection* scheme in Section 5.2. Our technique is centered around the precise, iteration-wise dependences already computed by a polyhedral optimizer. In contrast to common, mostly syntax based, reduction detection approaches, we leverage the dependence information to recognize complex memory reductions that are interleaved with other computations in affine loop nests. Finally, Section 5.3 presents *polyhedral expression propagation*, an optimization that differs from classical polyhedral techniques as it does not (principally) alter the schedule, thus iteration order of statement instances, or the data layout. Instead, it changes the time and place intermediate results are computed. The goal is to remove dependences and improve hardware utilization. Though, in the best case, we can additionally eliminate the original storage locations of the former intermediate results, thereby reducing the memory requirement and potentially even cache contention. This approach combines the precise dependence information and code transformation capabilities of the polyhedral model with target specific heuristics to generate high-performance code that would be hard to write and maintain by hand.

5.1 Complexity Reduction

One limitation of a polyhedral optimizer embedded in a general purpose compiler, is its *complexity*. This is true for the exponential compile time cost but also for the polyhedral optimizer tool itself. Especially non-expert users that want to perform polyhedral optimizations on their code base often need guidance and support. The goal should be an easy to use, fast, and comprehensible optimizer which allows casual users to leverage the optimization capabilities of the polyhedral model. While robust applicability (ref. Chapter 3) and correctness (ref. Chapter 4) are important steps towards this ambitious goal, we have to ensure that the user is able to understand and, if necessary, guide the transformations.

5.1.1 AST Level Annotations

In order to provide high-level information to the user, LLVM/POLLY offers debug outputs in various stages of the pipeline. The easiest to understand are AST level annotations for the transformed loop nest. Before our extension, LLVM/POLLY was only able to generate annotations for parallel loops. Now, the minimal dependence distance as well as the reduction dependences (ref. Definition 5.13) that limit parallelism can be shown to the user. Two examples that illustrate the minimal dependence distance annotations are given in Figure 5.2. Similarly, if loops only carry reduction dependences, annotations like the one shown in Figure 5.1 are produced.

```
#pragma reduction parallel (+ : sum)
for (i = 0; i < N; i++)
  for (j = 0; j < P; j++)
    #pragma simd
    for (k = 0; k < M; k++)
      sum[j][k] += i + j + k;
```

Figure 5.1: Example where the outermost loop is known to only reduction carry reduction dependences (ref. Definition 5.13) caused by the computation of `sum`.

While we introduced minimal dependence distance annotations to inform the user, they can also justify optimizations. In the `minimal_dependence_distance` branch of our research prototype (ref. Section 2.2.3) we utilize the information to justify vectorized execution of innermost loops. In case the chosen vector with is only conditionally smaller than the minimal dependence distance, we employ assumptions to ensure correctness (ref. Section 3.5). For the example in

> This Section describes automatic and semi-automatic enhancements towards fast and reliable polyhedral optimization that can be understood and guided by non-expert users. The improvements were mostly developed during our studies on polyhedral optimizations for data parallel languages [Kur17; MDH16] and our work on *Optimistic Loop Optimizations* [DGH17]. The described extension are all available in LLVM/POLLY except the minimal dependence distance guided vectorization (ref. Section 5.1.1) and the language specific context knowledge (ref. Section 5.1.2.2).

Figure 5.2a, we derive the assumption $p \leq -4 \vee p = 0 \vee p \geq 4$ which allows vectorization with a width of 4. To generate it, we assume a loop carried dependence distance less than 4, e.g., for the *RAW* dependence $\{\,\mathtt{S}(i) \rightarrow \mathtt{S}(i-p) \mid p < 0 \leq i < N + p \wedge (\mathbf{i} - \mathbf{p}) - \mathbf{i} < \mathbf{4}\,\}$, project the result onto the parameter space, and negate it. Similarly, we derive the assumption $N\%2 = 0 \vee N \leq 1$ to enable vectorized execution of two consecutive iterations for the loop shown in Figure 5.2b.

```
    #pragma min. dep. dis.: max(-p, p)
    for (i = 0; i < N; i++)
S:    A[i] = A[i + p] + 1;
```

(a) Loop with a simple parametric dependence distance. For p=0 the loop is parallel as there are no inter-iteration dependences.

```
#pragma min. dep. dis.: ((N-1)%2)+1
for (i = 0; i < N; i++)
  A[i] = A[N - i] + 1;
```

(b) Loop that carries dependences of various lengths but source and target are always at least `((N - 1) % 2) + 1` iterations apart.

Figure 5.2: Minimal dependence distance annotations we added to LLVM/Polly.

5.1.2 Leveraging Domain And User Knowledge

The importance of domain and user knowledge for high-performance and productivity can be estimated by the enormous success of domain specific languages (DSLs) in both research and industry [Bag+15; Heg+14; Kös+14; MVB15; Rag+12; Rag+13]. Given the complex requirements code has to fulfill for a sound and precise polyhedral representation (ref. Chapter 3 and 4), it is not uncommon that approximations and uncertainty about the expected inputs limit the optimization potential. So far we used optimistic assumptions based on statically predictable choices as well as dynamically collected information to resolve some uncertainties (ref. Section 3.5). Though, these techniques are not free in terms of compile time, code size, misspeculation overhead, and potentially profiling effort. In addition, the runtime verification cost of optimistically assumed knowledge will grow with the complexity of the assumptions.

5.1.2.1 User Provided Context Knowledge

The assumptions we presented so far are mostly speculative predictions of runtime values that have to hold for polyhedral optimization (ref. Section 3.5). Since they also have to be verified, code versioning is used. This will not only increase the code size, and often more importantly the compile time, but also induce a reoccurring verification cost at runtime.

Generally, we can guide the programmer towards providing additional information if static information is insufficient, thus assumption were needed to represent and optimize the code [DGP15]. Any additional information might allow to simplify the assumption set (ref. Section 3.5.3), and thereby also the synthesized runtime check. In the best case, additional information will not only allow us to generate more efficient checks but potentially even to remove the checks completely.

To ask the user for context knowledge, we first issue descriptions of the taken assumption constraints, together with their source code location and cause, e.g., no integer overflow assumption $N < 2^{31}$ in line 5, column 7. The user may now augment the source code with application specific knowledge that was not available in the original program but that was instead assumed during the modeling and optimization steps. In C/C++ programs this is done via calls to the `__builtin_assume` function. A statement such as `__builtin_assume(N < 32768)` would for example assure the compiler that N is always less than $32768 = 2^{15}$ at this program point. In all subsequent compilations of the source code, the user provided assumptions are utilized to simplify the optimistic ones taken automatically (ref. Section 3.5.3). In the best case, application specific knowledge eliminates the need for runtime checks completely. Though, any information may result in a reduction of the runtime checks and a simplification of the generated code.

5.1.2.2 Language Specific Context Knowledge

Polyhedral optimizers started to adopt manually annotated C-like programs [Bag+15] and also specialized DSLs [Bag+18; MVB15; Pra+17; Vas+18] to simplify the extraction of domain information. Though, we also need to improve the handling of general purpose programming languages. In Section 3.5.3 we described how impossible and undefined behavior can be used to simplify the assumption set. As part of this effort we use the annotation capabilities introduced in Section 5.1.2.1 to automatically generate program annotations in LLVM-IR that represent otherwise lost high-level knowledge from the source language. These annotations ensure that accesses to constant-sized arrays stay in-bounds, which is known in C/C++ but not in LLVM-IR. Similarly, other high-level facts could be communicated to low-level optimizers in order to simplify the required assumptions and potentially allow for more input specific optimizations.

```
procedure DEPENDENCEFILTER(S : SCoP, $\mathcal{D}$: IMap, L : Loop)
  First determine the relative loop depth of the parallel loop L inside the SCoP S.
  ld ← S.getRelativeLoopDepth(L)
  Then we iterate over all dependences and look for inter-iteration dependence carried by the parallel
  loop L. Thus, those that do originate and end up in different iterations of L.
  for d = $\{(S(\mathbf{i}), T(\mathbf{j})) \mid pred(\mathbf{i},\mathbf{j})\}$ in $\mathcal{D}$ do
    if S is not in L or T is not in L then continue
    if $\mathbf{i}_{ld} = \mathbf{j}_{ld}$ then continue
    $\mathcal{D} \leftarrow \mathcal{D}$ - d
  end for
  return $\mathcal{D}$
```

Algorithm 5.3: Dependence filter for parallel loops without barriers.

In our effort to optimize programs written in data parallel languages [MDH16], we used the existing parallelism information to eliminate conservative dependences induced by approximations (ref. Section 3.4). This effort led to the dependence filter shown in Algorithm 5.3. The procedure takes the SCoP S, the set of dependences $\mathcal{D}$, and a parallel loop L. It returns the dependence subset that will not violate the parallelism constraints known to hold. Kurtenacker [Kur17] later introduced support for parallel loops which may also contain barriers.

5.1.3 Representation of Partially Infinite Loops

```
uint8_t i, LB = ..., UB = ...;
for (i = LB; i != UB; i += 2)
  S(i);
```

Figure 5.4: Potentially infinite loop with an iteration counter `i` that will wrap around if `UB < LB`. The iteration count is only finite if `|UB - LB| % 2 == 0`.

The loop trip count is of special importance for all loop centric optimizations. Some transformations, e.g., loop unrolling, are commonly applied if the loop trip count is low while others, e.g., vectorization, are more profitability for longer running loops. However, the trip count is often not a compile time constant and could even be unbounded under certain parameter valuations. In the context of polyhedral optimizations, Benabderrahmane et al. [Ben+10] introduce modeling and code generation techniques for unknown loop bounds. Though, their technique introduces spurious dependences that are not well suited to handle a *partially unbounded* loop as shown in Figure 5.4. First note that for the example the loop counter `i` is generally allowed to overflow, thus wrap around (ref. Section 4.2). If now the difference between `LB` and `UB` is not divisible by the stride, here 2, the loop will not terminate. Figure 5.5 shows a similar but more complicated partially unbounded loop. The piecewise defined loop counter (ref. Section 6.4) is not allowed to wrap around but the loop might still not terminate. In practice, as in the shown examples, possibly unbounded loops are often a result of parametric loop bounds with an equality exit condition, thus == or `!=`. Such exit conditions can be introduced by programmers but also by canonicalizing program transformations performed prior to the polyhedral modeling.

```
signed i = LB;
while (i != UB) {
  S(i);
  i += (i > UB) ? -3 : 5;
}
```

Figure 5.5: Partially unbounded loop with an iteration counter that has a piecewise defined but affine evolution (ref. Section 6.4). The loop terminates only for some values of `LB` and `UB`, i.a., 0 and 8.

For polyhedral-model-based techniques a potentially infinite loop results in a partially unbounded iteration domain. While this is a precise representation of the input, it is also a cause for increased compile time and not necessarily supported by all algorithms [Ben+10]. Given these disadvantages and no immediate benefit for polyhedral optimizations, we believe it is reasonable to prevent the representation of unbounded iteration domains, especially if loops are only partially infinite. To determine the iteration domain, including the trip counts, many high-level polyhedral tools use the polyhedral extraction tool PET [VG12], while low-level compiler optimizations usually rely on a recurrence based approach. Even though both work reasonably well in their targeted areas, their advantages are orthogonal, thus leave room for improvement. PET models the domain for loop nests with affine conditions accurately, even though **`break`**, **`continue`**, equality exit conditions, and non-unit strides can easily lead to complex, piecewise defined, and partially unbounded trip counts. Recurrence based approaches, like the SCALAR EVOLUTION analysis [BWZ94; PCS05] in GCC and LLVM, can only handle bounded loops that have a closed form trip count expression (ref. Section 6.4), thus not the loop shown in Figure 5.4. In LLVM/POLLY we replaced the use of explicit trip counts with the precise iteration domain generation presented in Section 3.3. Now we combine the results of that domain generation with *bounded loop assumptions* Λ_{BL} to ensure finite and concise iteration domains.

Without bounded loop assumptions, the iteration domain of statement `S` in Figure 5.4 is shown in Formula 1. It was generated by the algorithm described in Section 3.3 and expresses the normalized loop iterations for which `S` is executed. Given the simplicity of the input loop, the domain is quite complex. It consists of the three disjuncts shown in separate lines in Formula 1.

$$\begin{aligned}\mathcal{D}_{\mathtt{S}} = \{\mathbf{i} \mid & \big(0 \le \mathbf{i} \wedge 2\mathbf{i} < UB - LB\big)\\ & \vee \big(0 \le \mathbf{i} \wedge (LB + UB)\%2 == 1 \wedge 2\mathbf{i} > UB - LB\big)\\ & \vee \big(0 \le \mathbf{i} \wedge (LB + UB)\%2 == 0 \wedge LB \ge UB + 2\big)\}\end{aligned} \qquad \mathbf{(1)}$$

To keep the iteration domains bounded and concise in the presence of partially infinite loops, we generate preconditions that exclude all unbounded parts. In our example, all three disjuncts provide a lower bound for the loop dimension **i**, but only the first one also bounds it from above. Thus, a bounded iteration domain is only guaranteed if the two last disjuncts both evaluate to *false*. To this end, we first denote the set of unbounded disjuncts in the domain $\mathcal{D}_{\mathtt{S}}$ as $\mathcal{D}_{\mathtt{S}}^{\infty}$. The bounded loop assumptions $\Lambda_{\mathrm{BL}}(\mathtt{S})$, for a statement `S`, are then defined as the negated projection of the unbounded part $\mathcal{D}_{\mathtt{S}}^{\infty}$ onto the parameter space ρ. Because assumptions restrict the set of valid inputs we have to intersect the result for all statements in the SCoP as shown in Formula 2.

$$\Lambda_{\mathrm{BL}} := \bigcap_{\mathtt{S} \in SCoP} \Lambda_{\mathrm{BL}}(\mathtt{S}) = \bigcap_{\mathtt{S} \in SCoP} \neg\pi_{\rho}(\mathcal{D}_{\mathtt{S}}^{\infty}) \qquad \mathbf{(2)}$$

Note that disjuncts can also be bounded by prior dimensions, as illustrated in Figure 5.6. Here, the constraints $0 \le \mathbf{j} < \mathbf{i}$ suffice to bound the inner dimension of statement `S`.

```
for (i = 0; ...; ...)
  for (j = 0; j < i; ...)
    S(i, j);
```

Figure 5.6: Generic loop nest with dependent conditionals.

The number of required bounded loop assumptions is shown in Table 3.42 on Page 75. Since unbounded loops cannot be optimized by POLLY, these numbers can be interpreted as the applicability impact. Thus, the number of valid SCoPs would decrease by 1.8%-37.7%, depending on the benchmark suite.

5.2 Reductions in the Polyhedral Model

Reductions are associative and commutative operations that reduce input data into a *reduction location* [Mid12]. A reduction in a loop will consequently induce a loop carried Dependence due to the reoccurring update of the reduction location. While loop carried dependences generally prevent transformations, especially parallel execution, the *reduction dependences* [PW94] inherit associativity and commutativity from the reduction operations that caused them. Consequently, reduction dependences do not prevent us from reordering the participating statement instances as long as we do not introduce non-reduction accesses to the reduction location in-between them.

Since reductions are common to all kinds of programs and their special properties allow parallel and vectorized execution of seemingly sequential loops, there has been a lot of research into their detection and optimization. While reductions were detected mostly statically [GO17; Jou86; JD89; PP91; PE95; RF93; Str17; SKN96; XKH04] there are also dynamic and speculative approaches with runtime verification [DYR02; Joh+12; RAP95; RP95; YR06]. Other works mainly focus on the exploitation of reduction properties [Ble89; FG94; GR06; RKC16; RF94].

```
  for (i = 0; i < M; i++)
S:  s[i] = 0;
  for (j = 0; j < N; j++) {
P:  q[j] = 0;
    for (k = 0; k < M; k++) {
Q:    s[k] = s[k] + r[j] * A[j][k];
R:    q[j] = q[j] + A[j][k] * p[k];
    }
  }
```

Figure 5.7: BiCG Polybench[a] kernel extracted from the BiCGStab linear solver.

[a] Online: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

In the following we introduce (an extended version of) our polyhedral-model-based memory reduction detection [Doe+15]. We also describe several potential ways to exploit reductions in a polyhedral optimizer. To motivate this work we can consider the BiCG kernel shown in Figure 5.7 and its polyhedral representation graphically illustrated in Figure 5.8. The innermost k-loop contains two statements, Q and R, which both perform a reduction computation. Thus, the dependences induced by Q, which are carried by the j-loop, as well as the dependences induced by R, which are carried by the k-loop, are both reduction dependences. We indicate these reduction dependences in Figure 5.8 through dotted arrows. If parallelization and vectorization of the reduction computations are desired, there are multiple possibilities. As vectorization is most beneficial for consecutive memory accesses, we could decide to keep the loops in their original order.

Our earlier approach to detect memory reductions [Doe+15] is available and enabled in LLVM/Polly. In this section we describe a generalized version which can be found in the `reductions` branch of our research prototype. LLVM/Polly does feature reduction-enabled scheduling while our evaluation [Doe+15] also employs reduction enabled code generation (ref. Section 5.2.3). Parts of our research were introduced into the Sambamba framework [Str+15] and are described in detail in the thesis of Streit [Str17].

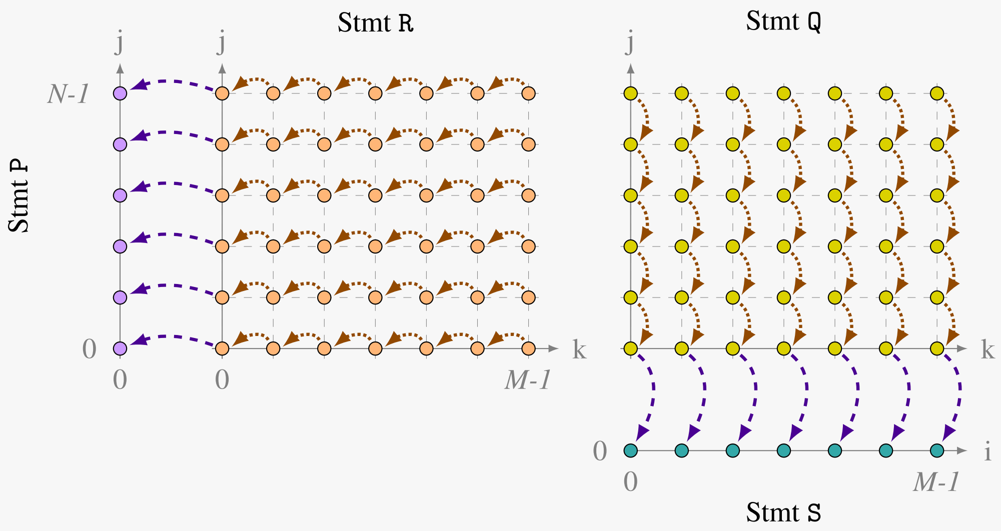


Figure 5.8: Graphical illustration[a] of the polyhedral representation build for the BiCG kernel shown in Figure 5.7. Each coordinate system corresponds to the iteration domain of one source code statement. The dots represent the statement instances and arrows denote dependences between them.

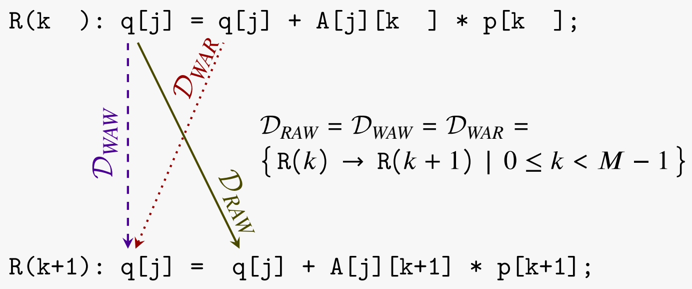
[a] Parts of this figure were first presented by Doerfert et al. [Doe+15].

If we decided to keep the original loop order for the R and Q statements, we would want to employ coarse-grained parallelism in the outer j-dimension and vector instructions in the inner k-dimension. While the associativity and commutativity of reduction dependences allow us to reorder the participating statement instances, we cannot simply execute them in parallel because that would create racing updates of the reduction location. Instead we have to employ either atomic read-modify-write operations or privatize the locations and accumulate the private copies at the end. The pseudo-code shown in Figure 5.9 illustrates both strategies for the reductions in the BiCG kernel. While atomic updates are a viable solution in some contexts [YR06], they are generally expensive. Especially for smaller tasks the overhead of atomic operations might outweigh the actual work [PE95]. Additionally, the vectorization benefit is lost if the hardware does not provide vectorized atomic updates. We therefore focus our discussion (ref. Section 5.2.2) and the evaluation (ref. Section 5.2.4) on privatization as it is generally well-suited for the task at hand.

```
parfor (j = 0; j < N; j++) {
  qpriv[4] = {0, 0, 0, 0}
  for (k = 0; k < M; k += 4) {
    atomic_vec4_add(&s[k], r[j] * A[j][k:3]);
    qpriv[0:3] = qpriv[0:3]+A[j][k:3]*p[k:3];
  }
  q[j] += qpriv[0]+qpriv[1]+qpriv[2]+qpriv[3];
  // Remainder loop
  for (k = M - M % 4; k < M; k++) {
    atomic_add(&s[k], r[j] * A[j][k]);
    q[j] = q[j] + A[j][k] * p[k];
  }
}
```

Figure 5.9: Pseudo-code for a parallelized and vectorized version of the BiCG kernel shown in Figure 5.7.

5.2.1 Reduction Definition



```
R(k  ): q[j] = q[j] + A[j][k  ] * p[k  ];
R(k+1): q[j] =  q[j] + A[j][k+1] * p[k+1];
```

$$\mathcal{D}_{RAW} = \mathcal{D}_{WAW} = \mathcal{D}_{WAR} = \{\mathtt{R}(k) \to \mathtt{R}(k+1) \mid 0 \le k < M - 1\}$$

Figure 5.10: Two consecutive instances of the statement R with regards to the k-loop shown in Figure 5.7.

The reduction definition used often in related works and (high-level) programming, is syntactic and centered around a single statement that reads a value, modifies it with an associative and commutative operation, and stores it back into the originally read location. The dependences (*RAW*, *WAR*, and *WAW*) caused by such a reduction inherit the associativity and commutativity, if, in-between the reduction operations, which are commonly part of a loop, no other operation reads or writes the reduction location. In our initial work on memory reductions [Doe+15], we also subjected ourselves to this general idea. To highlight the dependences caused by a reduction computation we again consider part of the BiCG example shown in the Figure 5.7. For two consecutive but abstract statement instances of statement R in the k-loop, the dependences illustrated in Figure 5.10 manifest. In our earlier work [Doe+15], we first identified *reduction-like computations* in a statement (here `q[j] = q[j] + ...`) and then looked for dependences that matched the shown pattern[1], namely *RAW* and *WAW* (and *WAR*) dependences between two instances of a reduction-like computation. Though, this will fail to detect reductions that span multiple statements. In Figure 5.11 such situations are shown as high-level C/C++ examples. However, in the low-level representation that is the input to LLVM/Polly, all loop carried reductions on a scalar variable will involve multiple "locations" in the form of SSA instructions: The phi node that translates the value from one iteration to the next, and the actual associative and commutative operation. Since these can easily be in different polyhedral statements, thus basic blocks for LLVM/Polly, we have to generalize our reduction definition.

```
    for (k = 0; k < M; k += 4) {
R0:   q[j] += A[j][k+0] * p[k+0];
R1:   q[j] += A[j][k+1] * p[k+1];
R2:   q[j] += A[j][k+2] * p[k+2];
R3:   q[j] += A[j][k+3] * p[k+3];
    }
```

(a) Statement R from the example in Figure 5.7 after the innermost loop body was unrolled four times.

```
    for (i = 0; i <= I; i++) {
      for (j = 0; j <= J; j++)
T:      sum = sum + A[i][j];
U:    result = sum;
      for (k = 0; k <= K; k++)
V:      sum = sum + B[i][k];
    }
```

(b) Two scalar reductions which in SSA form will cause phi nodes in the loop headers.

Figure 5.11: Reductions spanning multiple syntactic statements.

[1] The paper actually states that either *RAW* or *WAW* dependences are sufficient to detect reduction dependences. However, this is *not* the case if we allow partial reductions where the read and write access have different but overlapping access ranges, e.g., Figure 5.15. Our implementation in LLVM/Polly was not affected.

5.2.1.1 Dependence-Centered Reductions

A good definition for reductions in the polyhedral model should be as general as possible while ensuring all properties we are interested in. These properties, thus associativity and commutativity on a dependence level, stem from the binary computations that are performed in the program.

In the following, we will use $\oplus$ to denote a generic associative and commutative binary operator and assume each polyhedral statement consists of a single binary operation. While we want to identify reductions in a general polyhedral program representation, thus with polyhedral compound statements that can contain multiple binary operations as well as write accesses, we will restrict the setting for now to ensure a single write. This is not a conceptual limitation but allows us to base our reduction definition completely on dependences and atomic binary computations. To deal with compound statements we either split them after each write access or employ an intra-statement dependence analysis similar to the one used in our earlier approach [Doe+15].

Reduction Computation **Definition 5.12**

A binary computation $w = r_0 \oplus r_1$, with a binary operator $\oplus$ that is associative and commutative, defines a *reduction computation* c for each operand. Thus, $c_0 := (w, \oplus, r_0)$ is a reduction computation from r_0 to w and $c_1 := (w, \oplus, r_1)$ is a reduction computation from r_1 to w, both with the reduction operator $\oplus$.

A copy assignment $w = r$ defines a *reduction computation* $c := (w, \oplus, r)$ for each associative and commutative binary operator $\oplus$ that has an identity (or neutral) element.

To define a reduction in the polyhedral model, we first define a single *reduction computation* in Definition 5.12. It is a binary computation with an associative and commutative operator or, alternatively, a copy assignment. In contrast to classical definitions, we do not require the write w to be equal (syntactically and semantically) to the reads r_0 and r_1. We also do not specify if these are scalar accesses or memory accesses.

Reduction computations are not useful on their own but they are later needed to identify *reduction dependences*, as defined in Definition 5.13. Reduction dependences are useful as they do not (directly) cause validity constraints [PW94] which have to be fulfilled by schedule optimization.

Reduction Dependence **Definition 5.13**

A *reduction dependence* between two statement instances inherits associativity and commutativity from the underlying computations. Commutativity allows the source and the target to be executed in any order, and associativity allows parallel execution. Reduction dependences do consequently not directly restrict schedule optimization [PW94] but appropriate privatization dependences (ref. Section 5.2.2.1) are necessary for polyhedral optimization.

While Definition 5.13 states the property of a reduction dependence, it is not necessarily helpful to identify them. To this end, we start by defining a simple predicate for reduction dependences in Theorem 5.14. Similar to the definition we used in our earlier work [Doe+15], and which was briefly described in Section 5.2.1, this predicate will identify reductions if they are limited to a single statement. Note that this is already more general than most syntactic definitions because the input and output of the reduction computation are not required to be equal. Thus, we detect partial reductions where the read and write access associated with the input and output of the reduction computation have different but overlapping access ranges, e.g., Figure 5.15

Simple Reduction Predicate **Theorem 5.14**

Given a statement $\mathtt{S}$ with a reduction computation $c_{\mathtt{S}} = (w, \oplus, r)$. All dependent iteration pairs $(\mathtt{S}(\mathbf{i}), \mathtt{S}(\mathbf{j}))$ are part of a *reduction dependence* if,

(a) they are part of all self dependences that start and end in the reduction computation, thus $(\mathtt{S}(\mathbf{i}), \mathtt{S}(\mathbf{j})) \in (w{\rightarrow}r \cap r{\rightarrow}w \cap w{\rightarrow}w)$, and

(b) there is no read-after-write dependence that starts in iteration $\mathtt{S}(\mathbf{i})$ and ends in a read other than r or a statement instance other than $\mathtt{S}(\mathbf{j})$, thus $\forall r' : (\mathtt{S}(\mathbf{i}), \mathtt{T}(\mathbf{k})) \in w{\rightarrow}r' \implies r = r' \wedge \mathtt{S}(\mathbf{j}) = \mathtt{T}(\mathbf{k})$.

Proof.

To proof reduction dependences identified by this predicate adhere to Definition 5.13, hence that they are commutative, we show that we can swap the source and the target of an arbitrary dependent iteration pair $(\mathtt{S}(\mathbf{i}), \mathtt{S}(\mathbf{j}))$ without changing the semantics. This *assumes* there are no constraints induced by the second input of the reduction computation, denoted as v, that prevent this change. Note that associativity can be shown similarly.

The dependence relations required by condition (a) ensure that the read and written location of both reduction computation instances are equal. Thus, the read access r and write access w in iterations $\mathbf{i}$ and $\mathbf{j}$ access a single location *loc*. The presence of a write-after-write relation ensures that there is no intermediate write to *loc*. In addition, condition (b) guarantees that there is no other, e.g., intermediate, read of *loc* dependent on $\mathtt{S}(\mathbf{i})$. The reason is the absence of an intermediate write and the fact that all outgoing read-after-write dependences are known to end in r in iteration $\mathtt{S}(\mathbf{j})$. Thus, there are no accesses to *loc* between $\mathtt{S}(\mathbf{i})$ and $\mathtt{S}(\mathbf{j})$ which could affect, or be affected by, the swap.

It remains to show that the computation is equivalent *after* both statement instances were executed. Initially, the dynamically performed computation (involving *loc*) is $loc = loc \oplus v_{\mathbf{i}}$ in statement instance $\mathtt{S}(\mathbf{i})$ followed by $loc = loc \oplus v_{\mathbf{j}}$ in $\mathtt{S}(\mathbf{j})$. Since *loc* is not accessed in between, we can forward substitute the first result to get $loc = (loc \oplus v_{\mathbf{i}}) \oplus v_{\mathbf{j}}$. Due to the associativity and commutativity of $\oplus$, we can transform the computation to $loc = (loc \oplus v_{\mathbf{j}}) \oplus v_{\mathbf{i}}$, which we can then separate again into the computation $loc = loc \oplus v_{\mathbf{j}}$ in statement instance $\mathtt{S}(\mathbf{i})$ followed by $loc = loc \oplus v_{\mathbf{i}}$ in $\mathtt{S}(\mathbf{j})$. ■

The simple reduction predicate introduced in Theorem 5.14 is able to identify the reduction dependences in our motivating BiCG example (ref. Figure 5.7), the two reductions through `sum`, with regards to the inner j-loop and k-loop, shown in Figure 5.11b, as well as the partial reduction presented in Figure 5.15. The reductions in the unrolled BiCG example (ref. Figure 5.11a) and the reduction in the outer i-loop for Figure 5.11b are missed because they involve multiple statements. At the same time, the "almost reductions" shown in Figure 5.16 are correctly dismissed, thus no reduction dependences are identified even if there are dependences between instances of a reduction computation.

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    A[j + p] = A[j + i] + B[i][j];
```

Figure 5.15: Partial reduction[a] that manifests only in iterations where $p = i = 0$.

[a] This is a slight adaption of a figure presented by Doerfert et al. [Doe+15].

```
   for (i = 0; i < N; i++) {
S:   x = x + 1;
P:   x = i;
   }
```

(a) Two reduction computations which do not cause reduction dependences because neither causes a write-after-write self dependence.

```
for (i = 0; i < N; i++) {
  x = x + x;
}
```

(b) Two reduction computations of which neither causes reduction dependences because there are read-after-write dependences from the write to both reads.

Figure 5.16: Examples for which our reduction predicates, e.g., Theorem 5.14, properly determine that the reduction computations (ref. Definition 5.12) do not cause reduction dependences (ref. Definition 5.13).

5.2.1.2 Multi-Statement Reductions

In Figure 5.11 we introduced reductions that are not isolated computations in a single statement. In order to identify such cases we have to generalize our reduction notation and identification predicate. To this end, we take a closer look at the generalized version of the example presented in Figure 5.11b, that is shown in Figure 5.17a. While both codes are in some sense equivalent, the latter version is closer to static single assignment form (SSA-form) which is commonly used for compiler intermediate languages, e.g., LLVM-IR. Even if it is not completely in SSA-form[2], the code induces similar dependences as Figure 5.11b in SSA-form would. This does not mean the following discussion requires the input to be in any special format, e.g., SSA. On the contrary, we use this generalized example to show how more complicated situations are handled as well.

Brief descriptions of the dependences that arise for Figure 5.17a are provided in Figure 5.17b. These descriptions contain all important constraints without specifying unimportant details, e.g., unnecessary iteration domains information for the involved statement instances. In addition, we visualize the *RAW* / *WAW* / *WAR* dependences in Figure 5.17c / 5.17d / 5.17e.

[2] In actual SSA-form, `t1` and `t2` would not be defined twice. Instead, phi nodes would either select the initial value or the one written in the last loop iteration.

```
     for (i = 0; i <= I; i++) {
S:     t1 = t0;
       for (j = 0; j <= J; j++)
T:       t1 = t1 + A[i][j];
U:     t2 = t1;
       for (k = 0; k <= K; k++)
V:       t2 = t2 + A[i][k];
W:     t0 = t2;
     }
E:   use(t1);
```

(a) Almost SSA form version of Figure 5.11b.

$\gamma_0,\gamma_{12} = \{\mathtt{S}(i) \to \mathtt{T}(i,0)\}$ $\qquad \gamma_{11} = \{\mathtt{S}(i) \to \mathtt{S}(i+1)\}$

$\gamma_1,\gamma_{14},\gamma_{23} = \{\mathtt{T}(i,j) \to \mathtt{T}(i,j+1)\}$

$\gamma_2 = \{\mathtt{T}(i,J) \to \mathtt{U}(i) \mid J \geq 0\}$ $\qquad \gamma_{13} = \{\mathtt{T}(i,J) \to \mathtt{S}(i+1)\}$

$\gamma_3,\gamma_{16} = \{\mathtt{U}(i) \to \mathtt{V}(i,0)\}$ $\qquad \gamma_{15} = \{\mathtt{U}(i) \to \mathtt{U}(i+1)\}$

$\gamma_4,\gamma_{18},\gamma_{24} = \{\mathtt{V}(i,k) \to \mathtt{V}(i,k+1)\}$

$\gamma_5 = \{\mathtt{V}(i,K) \to \mathtt{W}(i) \mid K \geq 0\}$

$\gamma_6 = \{\mathtt{W}(i) \to \mathtt{S}(i+1)\}$ $\qquad \gamma_{17} = \{\mathtt{V}(i,K) \to \mathtt{U}(i+1)\}$

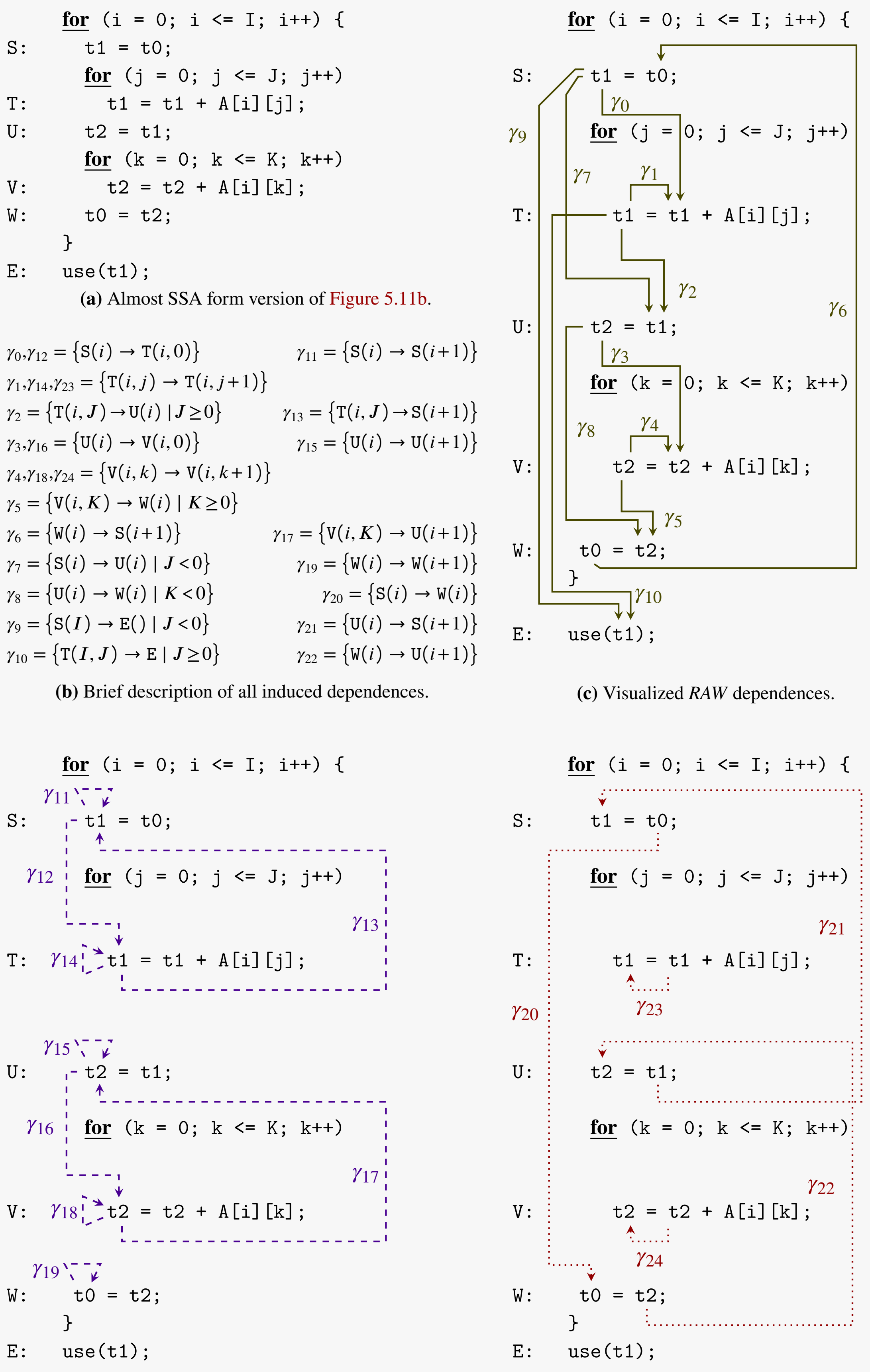
$\gamma_7 = \{\mathtt{S}(i) \to \mathtt{U}(i) \mid J < 0\}$ $\qquad \gamma_{19} = \{\mathtt{W}(i) \to \mathtt{W}(i+1)\}$

$\gamma_8 = \{\mathtt{U}(i) \to \mathtt{W}(i) \mid K < 0\}$ $\qquad \gamma_{20} = \{\mathtt{S}(i) \to \mathtt{W}(i)\}$

$\gamma_9 = \{\mathtt{S}(I) \to \mathtt{E}() \mid J < 0\}$ $\qquad \gamma_{21} = \{\mathtt{U}(i) \to \mathtt{S}(i+1)\}$

$\gamma_{10} = \{\mathtt{T}(I,J) \to \mathtt{E} \mid J \geq 0\}$ $\qquad \gamma_{22} = \{\mathtt{W}(i) \to \mathtt{U}(i+1)\}$

(b) Brief description of all induced dependences.



(c) Visualized *RAW* dependences.

(d) Visualized *WAW* dependences.

(e) Visualized *WAR* dependences.

Figure 5.17: Generalized version, and its dependences, of the code shown in Figure 5.11b.

The simple reduction predicate defined in Theorem 5.14 can again identify two reductions with regards to the innermost loops. Thus, the self dependences γ_1, γ_{14}, and γ_{23} of statement `T`, as well as γ_4, γ_{18}, and γ_{24} of statement `V`, are reduction dependences. To determine that the dependences carried by the i-loop are also associative and commutative we have to define a generalized reduction predicate. As a first step we introduce reduction computation chains in Definition 5.18.

Reduction Computation Chain **Definition 5.18**

A reduction computation $c = (w, \oplus, r)$ in a statement `S` is by itself a singleton *reduction computation chain* $C \coloneqq (w, \oplus, r, \mathbf{i}, \mathbf{i})$ for each statement instance $\mathbf{i}$ in the domain $\mathcal{D}_{\mathtt{S}}$.

A compound *reduction computation chain* $C \coloneqq (w_1, \oplus_0, r_0, \mathbf{i_0}, \mathbf{j_1})$ can be formed from two reduction computation chains $C_0 = (w_0, \oplus_0, r_0, \mathbf{i_0}, \mathbf{j_0})$ and $C_1 = (w_1, \oplus_1, r_1, \mathbf{i_1}, \mathbf{j_1})$ if they are *compatible*. Two chains are *compatible* if the following conditions hold:

(a) their binary operators are equal, thus $\oplus_0 = \oplus_1$, and

(b) there is a read-after-write dependence $w_0{\rightarrow}r_1$ from the output of the first one (w_0) to the input of the second one (r_1) that connects $\mathbf{j_0}$ and $\mathbf{i_1}$, thus $\left(\mathbf{j_0}, \mathbf{i_1}\right) \in w_0{\rightarrow}r_1$.

(c) the only read access dependent on the write w_0 in iteration $\mathbf{j_0}$ is r_1 in iteration $\mathbf{i_1}$, thus $\forall r : \left(\mathbf{j_0}, \mathbf{l}\right) \in w_0{\rightarrow}r \Longrightarrow \mathbf{i_1} = \mathbf{l} \land r_l = r$.

Given a reduction computation chain $C \coloneqq (w, \oplus, r, \mathbf{i}, \mathbf{j})$, we call $\mathbf{i}$ the start iteration of the chain and $\mathbf{j}$ as its end. Accesses that started or ended a sub-chain are part of the chain.

We further denote a reduction computation chain *closed* if the location read in the start iteration is equal to the location written in the end iteration. Using the above notation and denoting the access relation as f, a reduction computation chain is closed if $f_r(\mathbf{i}) = f_w(\mathbf{j})$.

A reduction computation chain $C = (w, \oplus, r, \mathbf{i}, \mathbf{j})$ describes that a value read by the access r in iteration $\mathbf{i}$ is *only* modified through associative and commutative operations according to $\oplus$, until it is written by the access w in iteration $\mathbf{j}$. In addition, intermediate results are neither duplicated nor do they escape the chain.

From the read-after-write dependences for the example in Figure 5.17a, shown in Figure 5.17c, we can identify multiple reduction computation chains. Most notably the closed ones listed in Table 5.19 which start with the read of `t0` in statement `S` and end with the write of `t0` in `W` in the same iteration of the i-loop which is not the last, thus $i < I$.

Reduction Computation Chain	**Traversed** dependences	
$C_0 \coloneqq (w_{\mathtt{W}}, +, r_{\mathtt{S}}, (i), (i))$	$\gamma_0, (\gamma_1)^*, \gamma_2,$	$\gamma_3, (\gamma_4)^*, \gamma_5$
$C_1 \coloneqq (w_{\mathtt{W}}, +, r_{\mathtt{S}}, (i), (i))$	$\gamma_0, (\gamma_1)^*, \gamma_2,$	$\gamma_8,$
$C_2 \coloneqq (w_{\mathtt{W}}, +, r_{\mathtt{S}}, (i), (i))$	$\gamma_7,$	$\gamma_3, (\gamma_4)^*, \gamma_5$
$C_3 \coloneqq (w_{\mathtt{W}}, +, r_{\mathtt{S}}, (i), (i))$	$\gamma_7,$	$\gamma_8,$

Table 5.19: Minimal closed reduction computation chains for Figure 5.17a that start in statement `S` and end in `W`, together with the read-after-write dependences that enabled their construction.

To derive associativity and commutativity for a dependence from associativity and commutativity computations distributed over multiple statements we provide a generalized reduction predicate in Theorem 5.20. While the underlying reasoning is the same as for the simple reduction predicate introduced in Theorem 5.14, the generalized version employs *closed* and *compatible reduction computation chains* to reason about the flow of a value through multiple computations.

Generalized Reduction Predicate **Theorem 5.20**

All dependences between accesses which are part of two *closed* and *compatible reduction computation chains*, $C_0 = (w_0, \oplus_0, r_0, \mathbf{i_0}, \mathbf{j_0})$ and $C_1 = (w_1, \oplus_1, r_1, \mathbf{i_1}, \mathbf{j_1})$, are *reduction* dependences. This includes not only the dependence between the initial read and final write, but also the ones that connect intermediate accesses which are part of the chain.

Proof.

Reduction dependences identified by this predicate adhere to Definition 5.13, hence that they are associative and commutative if appropriate privatization is used (ref. Section 5.2.2). We only show commutativity now by arguing the execution order of the statement instances in the two chains can be swapped without changing the result. This again *assumes* dependences induced by read accesses which are not part of the chains will not prevent this change.

If the statement instances which form the second chain C_1 are executed before the ones that form the first, we could potentially violate *RAW*, *WAR*, and *WAW* dependences. As we do not weaken dependences involving accesses that are not part of the chains, we can restrict the following discussion to those that are part.

Condition (c) in Definition 5.18 ensures that all intermediate writes of the preceding chain C_0, hence all write access instances until w_0 into which the value read by r_0 (in iteration $\mathbf{i_0}$) flows, are *only* read by access instances on that chain. Thus, read access instances *not* on the chain C_0, including ones on the succeeding chain C_1, are not flow-dependent on these writes. Consequently, the *false* dependences (*WAR* and *WAW*) between the chains can be broken as long as the chains are *not interleaved*. Additionally, dependences that end in the first chain C_0, or start in the second chain C_1, might require privatization as discussed in Section 5.2.2.

The above reasoning also guarantees that the only read-after-write dependence between the two chains is $w_0{\to}r_1$. Furthermore, condition (b) in the definition of *compatible* chains ensures that $w_0{\to}r_1$ contains the iteration pair $\left(\mathbf{j_0}, \mathbf{i_1}\right)$ and no other pair that starts in iteration $\mathbf{j_0}$. Since both chains are *closed* and *compatible* we know that both start in a read, and end in a write, of the same location *loc*. Thus, the result of the chain scheduled first and stored in *loc* is only read by the chain scheduled second, and later overwritten by that chain. Since all computations use the same associative and commutative operator, we can employ the same reasoning as in Theorem 5.14 to argue that the final result in *loc* is unchanged. ■

The generalized reduction predicate is able to identify all reduction dependences in the examples we presented in this section. For the unrolled inner loop from the BiCG example shown in Figure 5.11a, we can first determine the four minimal and closed reduction chains presented in Formula 3. Each actually describes a single reduction computation (ref. Definition 5.12) on the location `q[j]` in the respective statement.

$$C_{\mathtt{R}_x} := (w_{\mathtt{q[0]}}, +, r_{\mathtt{q[0]}}, (k), (k)) \qquad \text{for } 0 \leq x \leq 3 \tag{3}$$

The dependences between the statements are reduction dependences because the first three chains ($C_{\mathtt{R}_0}$ to $C_{\mathtt{R}_2}$) are compatible with the succeeding one in the same iteration of the k-loop, and the last chain ($C_{\mathtt{R}_3}$) is compatible with the first chain ($C_{\mathtt{R}_0}$) in the next iteration.

The generalization of the code in Figure 5.11, shown in Figure 5.17a, contains multiple reductions. First, there are the two innermost reductions that induce the self reduction dependences γ_1, γ_{14}, and γ_{23} of statement `T`, as well as γ_4, γ_{18}, and γ_{24} of statement `V`. Additionally, the dependences $\gamma_6, \gamma_{11}, \gamma_{13}, \gamma_{15}, \gamma_{17}, \gamma_{19}, \gamma_{21}$, and γ_{22}, all carried by the i-loop, are reduction dependences due to the compatible chains already shown in Table 5.19. Note that we can identify these reduction dependences in the original version (Figure 5.11) only after all non-escaping writes of the variable `result` have been eliminated. Thus, all but the last instance of statement `U` have to be removed to avoid spurious read-after-write dependences ending in $\mathtt{U}(i)$ for $0 \leq i < I$.

5.2.2 Privatizing Reductions

Privatization in the context of reductions commonly describes how the parallelized evaluation of a reduction is performed on private copies of the reduction location before the results are accumulated at the end. Thus, every parallel context c_i, which might be a thread or a vector lane, depending on the kind of parallelization, gets its own private location loc_i for the reduction location loc. Prior to the region executed by p parallel threads or vector lanes, the private locations $loc_1, \cdots, loc_p$ are allocated and initialized with the identity element of the corresponding reduction operation $\oplus$. The parallel instances are modified to perform cheap, non-atomic updates on their own private location loc_i instead of the original location loc. After the parallelized region, and before the first non-reduction access to loc, accumulation code is placed to join all intermediate results into loc, thus: $loc := loc \oplus loc_1 \oplus \cdots \oplus loc_p$.

Parallel execution of reductions through the use of privatization is legal because the associativity and commutativity of the reduction dependences (ref. Definition 5.13) guarantees that the final accumulation generates the same result as sequential execution of the region would have. Note that the final accumulation can be performed in logarithmic time by parallelizing the accumulation correspondingly [FG94].

```
// (A) initialization
for (i = 0; i < N; i++)
  // (B) initialization
  for (j = 0; j < M; j++)
    // (C) initialization
    for (k = 0; k < L; k++)
      A[j] += Q[i][j] * R[j][k];
    // (C) aggregation
  // (B) aggregation
// (A) aggregation
```

(a) Reduction enclosed in multiple loops.

	Overhead	
Loc.	**Memory**	**Aggregation**
(A)	$p \times \mathtt{M}$	$(p \times \mathtt{M}) \times 1$
(B)	$p \times \mathtt{M}$	$(p \times \mathtt{M}) \times \mathtt{N}$
(C)	p	$p \times (\mathtt{N} \times \mathtt{M})$

(b) Memory and aggregation overhead for the different privatization locations shown in Figure 5.21a with respect to the number of parallel contexts p and loop trip counts.

Figure 5.21: Possible privatization locations for the loop nest shown in part 5.21a with the corresponding memory consumption and aggregation overhead listed in part 5.21b.[a]

[a] This Figure was first presented by Doerfert et al. [Doe+15].

While we can limit the number of parallel contexts, thus the number of parallel threads or the vector width, we cannot generally bound the number of reduction locations. Furthermore, the number of necessary locations, as well as the initialization and aggregation work needed, varies with the placement of the privatization code. Consider the example in Figure 5.21a where the different possibilities to exploit loop parallelism, potentially through reduction parallelism, are explicitly marked as (A), (B), and (C). Using location (C) for privatization allows to parallelize the innermost k-loop. In this setting, only p private copies of the reduction location A[j] are needed. There is no benefit in choosing location B because $p \times \mathtt{M}$ privatization locations are then required (we have M different reduction locations modified by the j-loop and p parallel contexts), but there is no gain in the amount of parallelism because the j-loop was already parallel. Finally, choosing location (A) for privatization might be worthwhile. We still only need $p \times \mathtt{M}$ privatized values, but can then execute the i-loop in parallel as well. Additionally, we save aggregations: While for location (C) p values are summed up $\mathtt{N} \times \mathtt{M}$ times, and for location (B) $p \times \mathtt{M}$ values are aggregated N times, location (C) requires the $p \times \mathtt{M}$ values to be summed up only once.

For locations (A) and (B) in Figure 5.21a, the entire array A has to be privatized. While this can be the only possibility to exploit parallelism, e.g., for the BiCG example in Figure 5.7, assuming we do not distribute statements Q and R into separate loop nests, there are several downsides:

- Privatization overhead grows with the problem size.
- The shared cache is polluted with elements from private arrays.
- Depending on the hardware, memory consumption might become an issue.

In general, a trade-off has to be made between memory consumption, aggregation time and exploitable parallelism. Finding a good placement is additionally difficult as it depends on hardware and workload. Furthermore, the choices for privatization code placement in the resulting code is limited by the schedule. The scheduler should consequently be aware of the implications with respect to the efficiency of necessary privatization.

5.2.2.1 Privatization Dependence

```
S: avg = 7;
   for (i = 0; i <= N; i++)
P:   avg = avg + 3 * A[i];
Q: avg = avg / N;
```

Figure 5.22: Reduction and non-reduction accesses to the location avg.

Non-polyhedral optimizations often exploit reduction properties locally and in isolation. Thus, they change the schedule of the reduction computation, e.g., through parallelization, but keep it separate from other operations which involve the reduction location. In this case, privatization can be applied as a post-processing step during code generation. However, polyhedral optimization is generally neither local nor isolated. Instead, polyhedral schedule optimization involves all statement instances, including the ones that participate in the reduction computation and the ones that access the reduction location before or after. A new schedule is considered sound if all validity constraints, constructed from dependences between the instances, are satisfied [Fea92a; Fea92b]. While associativity and commutativity of reduction dependences (ref. Definition 5.13) do not impose an ordering of the dependent instances [PW94], we cannot simply omit the corresponding validity constraints[3]. If we would, non-reduction operations involving the reduction location could get interleaved with reduction operations. To show this situation we can consider the example in Figure 5.22. If we omit the validity constraints for the reduction dependences $\{\, \texttt{P}(i) \rightarrow \texttt{P}(i+1) \mid 0 \leq i < N \,\}$, we only restrict the new schedule through $\{\, \texttt{S}() \rightarrow \texttt{P}(0) \mid N \geq 0 \,\}$, $\{\, \texttt{P}(N) \rightarrow \texttt{Q}() \mid N \geq 0 \,\}$, and $\{\, \texttt{S}() \rightarrow \texttt{Q}() \mid N < 0 \,\}$. Since these dependences do not constrain the instances $\{\, \texttt{P}(i) \mid 0 < i < N \,\}$, they could be scheduled before `S` or after `Q`, either would most likely result in a different final value for `avg`.

```
S: avg = 7; p[0:N] = {0,...,0};
   for (i = 0; i <= N; i++)
P:   p[i] = p[i] + 3 * A[i];
Q: avg = sum(p[0:N]); avg /= N;
```

Figure 5.23: Implementation that causes the privatization dependences we add to ensure correct schedule optimization for the reduction in Figure 5.22.

While we want to ignore the validity constraints caused by reduction dependences during schedule optimization, we have to ensure the relative order between reduction and non-reduction operations is retained. To this end, we introduce *privatization dependences* which overestimate the effect privatization of the reduction computation would have. In Figure 5.23 we show how an explicit implementation would look that causes the privatization dependences we introduce for Figure 5.22. The privatization of `avg` as `p[0:N]` is actually a "maximal" expansion [Fea88a]. Together with the additional accesses in statements `S` and `Q`, all dependences which only start or end in the reduction computation, thus which only start or end in $\{\, \texttt{P}(i) \mid 0 \leq i \leq N \,\}$, are also expanded to start, or respectively end, in all reduction instances.

[3] Note that we here, as we did throughout this section, assume *value-based* dependences and not *memory-based* dependences [Fea91; MAL93; PW93]. This is important as only the latter contain transitive dependences which might be conservatively approximated. Though, most modern polyhedral tools employ value-based dependences.

The computation of privatization dependences γ_π for reductions that manifest in a single-statement instance, e.g., the ones identified by Theorem 5.14 or our earlier work [Doe+15], is shown in Formula 4. All non-reduction dependences γ_ν are transitively extended by all reduction dependences γ_ρ to ensure the reductions are not interleaved with other computations, including other reductions, on the same locations.

$$\gamma_\pi := \left(\gamma_\nu \circ_{dom} (\gamma_\rho)^*\right) \cup \left(\gamma_\nu \circ_{rng} (\gamma_\rho)^*\right) \tag{4}$$

When we transition to multi-statement reductions, as identified by the generalized reduction predicate in Theorem 5.20, we have to modify the privatization dependence definition to account for: dependent access instances which are part of the chain and which should not be transitively extended, as well as the possibility of interleaved execution of reduction computation chains. The first problem can be solved by eliminating all intra-chain dependences from the non-reduction dependences γ_ν before we extend them. If this step is missed, the reduction dependences will be re-introduced into the validity constraints of the schedule problem. The second problem is however harder to tackle in a general way, given the existing interface of the most commonly used schedule optimizer ISL. While our goal is to eliminate reduction dependences (ref. Definition 5.13) in order to allow free reordering of reduction computation chains, we cannot allow two chains to be interleaved. If we would, intermediate values could be overwritten prior to their use, causing a different final result. Since we cannot provide a general answer to this problem, or show how an extended interface for the ISL scheduler should look like, we restrict our evaluation to single-statement instances and leave this task open for future exploration.

5.2.3 Exploiting Reduction Properties

Reduction properties, hence the associativity and commutativity of the reduction dependences, often enable parallel execution of otherwise sequential loops. In non-polyhedral approaches, this use case is arguably the most prominent one. Though, polyhedral schedule optimization can exploit reduction properties also in other ways already hinted at in Section 5.2.2.1. In the following we briefly describe the three main schemes that allow polyhedral optimizers to exploit reductions which we already introduced in our earlier work [Doe+15].

Reduction-Enabled Code Generation

Similar to non-polyhedral-based approaches, a polyhedral optimizer can exploit reductions during the code generation. This non-invasive method does not require modification of the polyhedral representation and dependences. Schedule optimization can consequently be performed with the validity conditions of the reduction dependences in place. During the code generation, loops which only carry reduction dependences are parallelized through the use of privatization (ref. Section 5.2.2) or atomic accesses to the reduction location.

Reduction-Enabled Scheduling

To increasing the freedom for polyhedral schedule optimization, the validity conditions caused by reduction dependences can be eliminated. As discussed in Section 5.2.2.1, this approach requires privatization dependences to be introduced to ensure soundness.

Reduction-Aware Scheduling

Exploiting reductions through privatization or atomic access induces a non-trivial cost (ref. Section 5.2.2). The objective function used during schedule optimization should be aware of this cost in order to select an appropriate transformation and to determine the optimal use and placement of reductions. While this scheme subsumes *reduction-enabled scheduling*, it requires non-trivial modifications to the scheduler and objective function.

At this point it is important to note that current state-of-the-art polyhedral schedulers, e.g., the one in LLVM/Polly, do not directly determine if and where parallelism should be exploited. The schedulers only ensure parallel dimensions to be present in the final schedule. Consequently, a *reduction-enabled code generation* is always necessary to enable parallel execution of reductions.

We can use the example in Figure 5.24a to showcase how *reduction-enabled* and *reduction-aware scheduling* can improve the result of polyhedral schedule optimization. Assuming we want to exploit coarse grained parallelism, the scheduler could interchange the loops surrounding both statements, potentially also fusing the then outer j-loop and l-loop. However, as long as reduction dependences are present, the execution order of the i-loop and k-loop have to remain unchanged. While *reduction-enabled scheduling* does not impose reduction dependence constraints, and thereby allows to fuse the i-loop and k-loop after one is reversed, it also does not model the cost of privatization. Consequently, the initial loop interchange, which moves the native parallelism in the j and l dimension to the outer level, is not required for outermost, coarse grained parallelism. Instead, the reduction carrying dimension(s) could be kept outermost in this scheme, requiring privatization of the entire `A` and `B` array later on. Only with *reduction-aware scheduling* all transformations, namely the interchanges and fusions after one loop reversal, would become beneficial to the scheduler. The loop nest shown in Figure 5.24b illustrates how the result could then look like.

```
     for (i = 0; i <= N; i++)
       for (j = 0; j <= M; j++)
S:        A[j] = A[j] + a * C[  i][j];
     for (k = N; k >= 0; k--)
       for (l = 0; l <= M; l++)
P:        B[l] = B[l] + b * C[N-k][l];
```

(a) Loop nest that allows for loop fusion, interchange and parallelization.

```
     #pragma parallel for
     for (jl = 0; jl < M; jl++)
       for (ik = 0; ik < N; ik++) {
S:        A[jl] = A[jl] + a * C[ik][jl];
P:        B[jl] = B[jl] + b * C[ik][jl];
       }
```

(b) Possible result after *reduction-aware scheduling* transformed the loop nests shown in Figure 5.24a.

Figure 5.24: Example to showcase how *reduction-enabled* and *reduction-aware scheduling* can improve the result of polyhedral schedule optimization.

5.2.4 Evaluation

In the following, we present the evaluation of our earlier work [Doe+15] which was done with a then current version of LLVM/POLLY. *Since the conclusions are only partially transferable to a now current version, we augmented the original discussion to help interpret the results today.*

Throughout this section, we described an extension of our earlier work on memory reductions in the polyhedral model [Doe+15]. Especially the generalized reduction predicate provided in Theorem 5.20 is a novel advancement. While the simple reduction predicate in Theorem 5.14 is described in the same fashion, it is very similar to the reduction detection presented in our earlier approach. Most importantly, the reduction dependences that can be identified are basically the same. A simplified form of the reduction detection presented in our earlier work is available in LLVM/POLLY, and a prototype implementation of the generalized reduction predicate can be found in the `reduction` branch of our research prototype (ref. Section 2.2.3).

An often raised issue with the implementation was the use of transitive dependences. While the transitive closure of an integer relation can be imprecise and costly to compute, our experience in LLVM/POLLY, as well as the discussion by Pugh [Pug91b], indicate that the implementation is sufficient in practise. Pugh and Wonnacott [PW94] even argue that the transitive closure of value-based reduction dependences in real programs can be computed in an easy and fast way.

We implemented *reduction-enabled code generation* as well as *reduction-enabled scheduling* and evaluated the effects on compile time and runtime for the Polybench 3.2 benchmark suite on an *Intel(R) core i7-4800MQ* quad core machine with the default benchmark inputs.

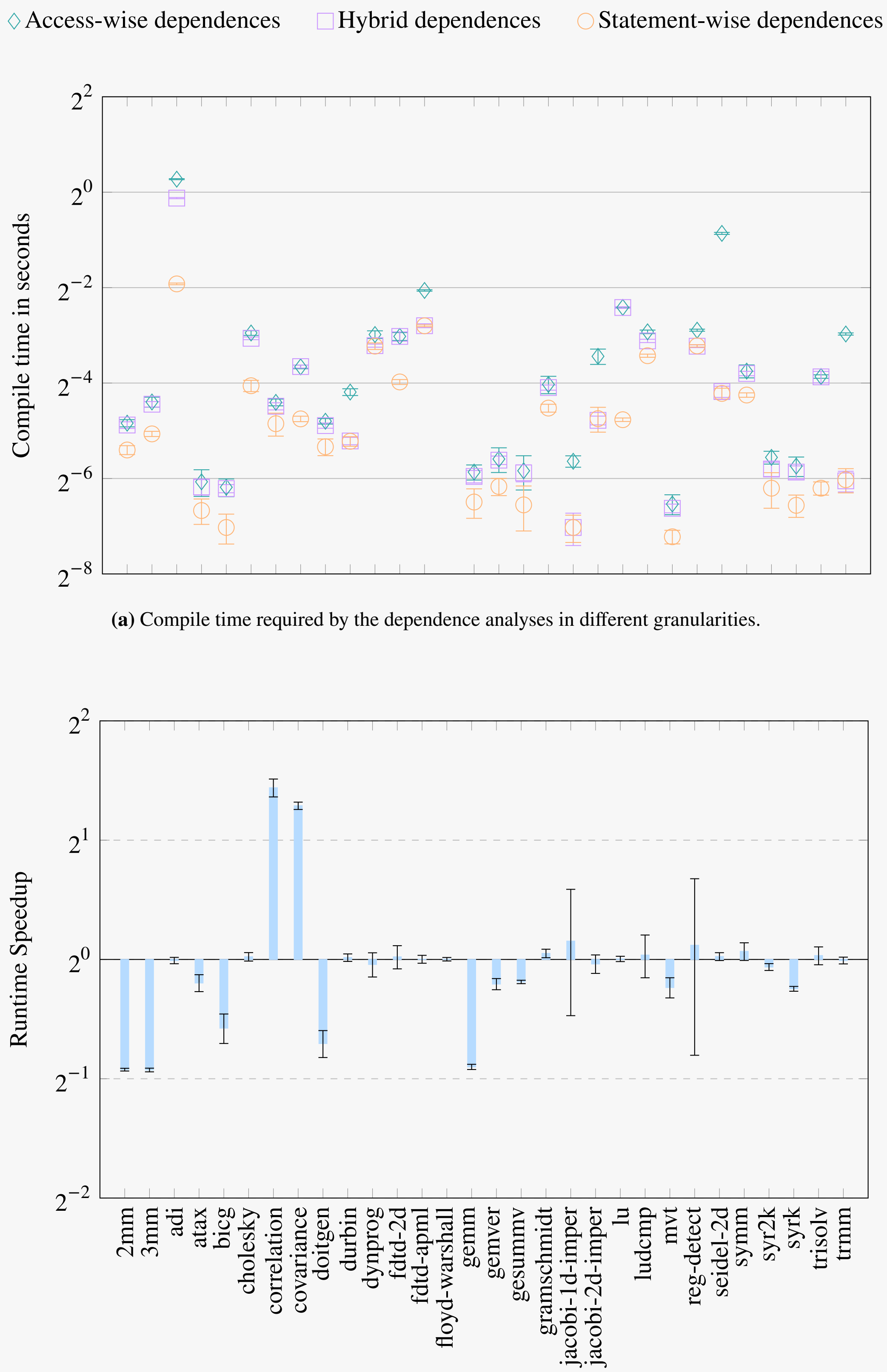
The reduction identification, which is very similar to the simple reduction predicate described in Theorem 5.14, is able to identify 52 arithmetic reductions performed on memory locations distributed over the 30 benchmarks. This is however only the case for older Polybench versions, and if POLLY is run early in the optimization pipeline. Newer benchmark versions, as well as optimization passes in the LLVM pipeline, replace the memory reductions with reductions on scalar variables which are not detected. One reason is that scalar reductions in static single assignment form (SSA) can easily involve multiple locations as discussed in Section 5.2.1.2. Another is the fact that the LLVM/POLLY version at the time did force all communication between polyhedral statement instances to be performed through memory locations. Thus, scalar reductions were demoted to stack allocations prior to the polyhedral modeling.

The detection of reduction-like computations, which are basically reduction computations as defined in Definition 5.12, is actually performed in basic blocks, hence compound polyhedral statements that might contain more than just the reduction computation. As our reduction definition is centered around dependences (ref. Section 5.2.1.1), we can either perform a local, intra-statement dependence analysis, or we virtually split polyhedral compound statements that contain more than a single computation. We choose the latter because it allows to use the general polyhedral dependence analysis [Fea91] and treat all discovered dependences (intra- and inter-statement) alike. However, this approach increases the complexity of the dependence analysis and consequently the compile time. To measure the effect, we timed this particular part of the compilation for each of the benchmarks and compared:

- the default dependence computation between statement instance ○,
- a completely access-wise dependence analysis ♢, and
- our hybrid dependence analysis □, which isolates reduction computations (ref. Definition 5.12) but keeps non-reduction accesses in compound statements.

As shown in Figure 5.25a our hybrid dependence computation takes up to 5× as long (benchmark `lu`) than the default statement-wise dependence computation but in average only 85% more. Access-wise dependence computation however is up to 10× slower than the default and takes in average twice as long as our hybrid approach. Note that both fine-granular approaches do not only compute the dependences (partially) on the access level but also the reduction and privatization dependences as explained in Section 5.2.1 and Section 5.2.2.1. It is also worth to note that LLVM/Polly nowadays splits compound statements by default to increase the scheduling freedom for the contained computations.

Figure 5.25b shows the speedup of the *reduction-enabled code generation* as well as *reduction-enabled scheduling* compared to the a reduction-unaware Polly. The additional scheduling freedom causes speedups for the data-mining applications (`correlation` and `covariance`) but slowdowns especially for the matrix multiplication kernels (`2mm`, `3mm`, and `gemm`). This is due to the way Polly generates vector code. The deepest dimension of the new schedule that is parallel (or now reduction parallel) will be strip-mined and vectorized. Hence the stride of the contained accesses, crucial to generate efficient vector code, is not considered. However, we do not believe this to be a general shortcoming of our approach as there are existing approaches to tackle the problem of finding a good vector dimension [Kon+13] that would benefit from the additional scheduling freedom as well as knowledge of reduction dependences. Another fact that needs to be considered is the immense native parallelism already present in the evaluated benchmarks. Most loop nests contain already at least one parallel loop, even without reduction parallelism. We can consequently not expect significant speedups when reduction awareness allows to parallelize even more loops as one is generally sufficient to exploit the available resources.



(a) Compile time required by the dependence analyses in different granularities.

(b) Performance of Polly with *reduction-enabled code generation* and *reduction-enabled scheduling* normalized to a Polly version without both.

Figure 5.25: Evaluation results for Polybench 3.2 benchmarks.

5.2.4.1 BiCG Case Study

While Polybench is a collection of inherently parallel programs, there is one, the BiCG kernel shown in Figure 5.7, that requires reduction support to expose parallelism if the two statements are not separated. To study the effects of parallelization combined with privatization of multidimensional reductions in this kernel we compared two parallel versions to the non-parallel code POLLY would generate without reduction support. The first version "*Outer*" has a parallel outermost loop and therefore needs to privatize the whole array `s`. The second version "*Tile*" parallelizes the second outermost loop. Due to tiling, only "tile size" (here 32) locations of the `q` array need to be privatized. Table 5.26 shows the speedup compared to the sequential version for both a quad core machine and a 8 × 4-core server. As the input grows larger the threading overhead as well as the inter-chip communication on the server will cause the speedup of *Tile* to stagnate, however on a one chip architecture this version generally performs best. *Outer* on the other hand will perform well on the server but not on the 4-core machine. We therefore believe the environment is a key factor in the performance of reduction-aware parallelization and a reduction-aware scheduler is needed to decide under which run-time conditions privatization becomes beneficial.

	`Input Size`							
	$2^{10} \times 2^{10}$		$2^{12} \times 2^{12}$		$2^{14} \times 2^{14}$		$2^{15} \times 2^{15}$	
Outer	0.19	0.55	2.31	0.75	3.91	0.72	2.19	0.96
Tile	0.03	1.10	0.32	1.54	0.10	1.60	0.16	2.21

Table 5.26: BiCG run-time results. The values are speedups compared to the sequential POLLY version, first for the 32-core machine, then for the 4-core machine.

5.2.5 Related Work

The following discussion of related work is in large parts taken from our previous work [Doe+15], and only augmented with a discussion of more recent approaches.

Reductions are a long lasting research topic, mainly in the context of loop parallelization. Various approaches to detect, model, and optimize reduction computations have been proposed. As our work intersects with all three parts we will discuss them in separation.

5.2.5.1 Detection

Reduction detection started with pattern-based approaches on source statements [JD89; PE95; RP95; RF93; RF00] and evolved into more elaborate techniques using program dependency graphs [PP91; Str+15; Str17], symbolic evaluation [FG94], data dependency graphs [SKN96], or constraint solving [GO17] to find candidates for reduction computations.

For functional programs, Xu, Khoo, and Hu [XKH04] introduced a type system to deduce parallel loops including pattern based reductions. Their typing rules are similar to the conditions we check for in the reduction predicates (ref. Theorem 5.14 and Theorem 5.20).

Sato and Iwasaki [SI11] describe a pragmatic system to detect and parallelize reduction and scan operations based on the ideas introduced by Matsuzaki, Hu, and Takeichi [MHT06]: the representation of (part of) the loop as a matrix multiplication with a state vector. They can handle mutually recursive scan and reduction operations as well as maximum computations implemented with conditionals, but they are restricted to innermost loops and scalar accumulation variables. As an extension Zou and Rajopadhye [ZR12] combined the work with the polyhedral model and the recurrence detection approach of Redon and Feautrier [RF93; RF00]. This combination overcomes many limitations, e.g., multidimensional reductions (and scans) over arrays are handled. However, the applicability is still restricted to scans and reductions representable in State Vector Update Form [KS73].

In our setting we identify actual reductions utilizing the already present dependence analysis, an approach very similar to the what Suganuma, Komatsu, and Nakatani [SKN96] proposed to do. However, we only perform the expensive, access-wise dependence analysis for reduction candidates, and not for all accesses in the SCoP. Nevertheless, both detections do not need the reductions to be isolated in a separate loop as assumed by Fisher and Ghuloum [FG94] or Pottenger and Eigenmann [PE95]. Furthermore, we allow the induced reduction dependences to be of any form and carried by any subset of outer loop dimensions. This is similar to the nested *Recur* operator introduced by Redon and Feautrier [RF93; RF00]. Hence, reduction dependences are not restricted to a single loop dimension, as in other approaches [FG94; JD89; SI11], but can also be carried by multiple loops in a subset of their iterations.

5.2.5.2 Modeling

Modeling reductions was commonly done implicitly, e.g., by ignoring the reduction dependences during a post parallelization step [JD89; PP91; PE95; RP95; RF93; Ven+14; XKH04]. This is comparable to the reduction-enabled code generation described in Section 5.2.3. However, we believe the full potential of reductions can only be exposed when the effects are properly modeled on the dependence level.

The first to do so, namely to introduce reduction dependences, where Pugh and Wonnacott [PW94]. Similar to most other approaches [GR06; RF93; RF94; RF00; SI11; SKN96; ZR12], the detection and modeling of the reduction was performed only on C-like statements and utilizing a precise but costly access-wise dependence analysis (ref. Figure 5.25a). In their work, they utilize both memory and value-based dependence information to identify statements with an iteration space that can be executed in parallel, possibly after transformations like array expansion [Fea88a]. They start with the memory-based dependences and compute the value-based dependences as well as the transitive self-dependence relation for a statement in case the statement might not be inherently sequential. This is very similar to the privatization dependences we introduced in Section 5.2.2.1 to enforce the integrity of a reduction during schedule optimization.

Stock et al. [Sto+14] describe how reduction properties can be exploited in the polyhedral model. However, they do not mention how reductions are detected, nor how omitting reduction dependences may affect other statements (ref. Section 5.2.2).

In the work of Redon and Feautrier [RF94], as well as the extension by Gupta, Rajopadhye, and Quinton [GRQ02], reduction modeling is performed on a system of affine recurrence equations (SAREs) and after array expansion [Fea88a] was applied. Thus, after all *false* dependences (*WAR* and *WAW*), caused by memory reuse, were eliminated. In this setting, the possible interference between reduction computation and other statements is simplified. Though, it might not be practical for general purpose compilers due to memory constraints. As an extension to these scheduling approaches on SAREs we introduced privatization dependences in Section 5.2.2.1. These ensure the integrity of reduction computations without the need for any preprocessing of the input. Only due to the privatization dependences we can allow general polyhedral schedule optimization, involving statement instances that access a location as part of a reduction computation as well as others that access the same location outside of a reduction.

5.2.5.3 Optimization

Optimization in the context of reductions is twofold. There is parallelization of the reduction as it is given in the input, and transformation as well as possible parallelization of the input with awareness of the reduction properties. The former is very similar to the reduction-enabled code generation as described in Section 5.2.3. In different variations, innermost loops [SI11], loops containing only a reduction [FG94; PE95] or recursive functions computing a reduction [XKH04] were parallelized or replaced by a call to a possibly parallel reduction implementation [Ven+14]. The major drawback of such optimizations is that reductions have to be computed either in isolation or with the statements that are part of the source loop that is parallelized. Thus, the reduction statement instances are never reordered or interleaved with other statement instances, even if it would be beneficial. In order to allow powerful transformations in the context of reductions, their effect, hence the reduction dependences, as well as their possible interactions with all other statement instances must be known. The first polyhedral scheduling approach which optimally, e.g., with regards to latency, schedules reductions together with other statements was presented by Redon and Feautrier [RF94]. It assumed that all reduction operations are computable in a single time step. With such atomic reduction computations there are no reduction statement instances which could be reordered or interleaved with other statement instances. Gupta, Rajopadhye, and Quinton [GRQ02] extended that work and lifted the restriction on an atomic reduction computation. As they schedule the instances of the reduction computation together with the instances of all other statements their work can be seen as a reduction-enabled scheduler (ref. Section 5.2.3) that optimally minimizes the latency of the input. Most recently, Prajapati [Pra18] extended this work on non-atomic polyhedral reduction scheduling. The extension contains a reduction aware tiling scheme which is an important step towards a reduction-aware scheduler.

To speed up parallel execution of reductions the runtime overhead needs to be minimized. Pottenger and Eigenmann [PE95] proposed to privatize the reduction locations instead of locking them for each access and Suganuma, Komatsu, and Nakatani [SKN96] described how multiple reductions on the same memory location can be coalesced. If dynamic reduction detection [RP95] was performed, different privatization schemes to minimize the memory and runtime overhead were proposed by Yu and Rauchwerger [YR06]. While the latter is out of scope for a static polyhedral optimizer, the former might be worth investigating once our approach is extended to multiple reductions on the same location. In contrast to these prior works, we are the first to describe the privatization dependence constraints for reductions (ref. Section 5.2.2.1) which allow general polyhedral schedule optimization involving statement instances participating in reduction computations and ones that do not.

In contrast to polyhedral optimization or parallelization, Gupta and Rajopadhye [GR06] exploited reduction properties in the polyhedral model to decrease the complexity of a computation in the spirit of dynamic programming. Their work on reusing shared intermediate results of reduction computations is completely orthogonal to ours.

While array expansion[Fea88a] is not a reduction optimization, it is similar to the privatization step of any reduction handling approach (ref. Section 5.2.2). However the approaches differ in the number of privatized copies introduced, the accumulation of these private copies, as well as the kind of dependences that are removed. While privatization only introduces a new location for each parallel context, e.g., parallel thread or vector lane, general array expansion introduces a new location for each write to the location. In terms of dependences, array expansion will remove *false* dependences (*WAR* and *WAW*) that are introduced by the reuse of memory while reduction dependences are made up of all kinds of dependences including flow (*RAW*) dependences. Because of the actual reuse of formerly computed values, reduction privatization also requires a more elaborate accumulation scheme to combine all private copies.

5.3 Polyhedral Expression Propagation

Polyhedral expression propagation statically replaces uses of scalar variables or memory loads with their defining expression. The goal is to eliminate flow (*RAW*) and anti (*WAR*) dependences to enable more scheduling transformations. At the same time, propagation can independently improve cache utilization if the defining expression does not introduce new cache misses. Additionally, the propagation of all uses of a non live-out location allows to eliminate the original definition including the associated computation. If the definition caused output (*WAW*) dependences they are removed as well. Furthermore, if the location was temporary it might be unused after propagation and thereby subject to elimination. Thus, expression propagation can not only eliminate different kinds of dependences but also the memory requirements. However, it is a double-edged sword. Duplicating a defining expression or moving it into a deeper nested loop can easily increase the code size as well as the overall computation. Nevertheless, we show that the trade-off between recomputation and memory as well as dependence reduction can be beneficial, especially if the program is memory-bound or further loop optimizations can be enabled.

In terms of classical polyhedral optimizations, expression propagation does not fall into the two main categories that describe most approaches. While scheduling optimizations change the time and place at which statement instances are executed [Bon+08; Bon+10; Fea92b; Len93; LCL99] and memory optimizations alter the storage locations of intermediate results [ABD07; BBC16; DSV03; LF98; QR00; Str+98; WR96], expression propagation will instead modify the expressions that compute (intermediate) results. As it is thereby (mostly) orthogonal to schedule and memory optimizations, it can be used as a standalone transformation but also in conjunction with existing techniques for which it acts as a canonicalization and simplification step.

A real world example to illustrate the different uses cases of polyhedral expression propagation is shown in Figure 5.28. The scalar definitions in statements `S0`, `Q0`, `Q1`, and `Q2` will all be eliminated after the defining expressions, here indicated with dotted and dashed rectangles, have replaced the scalar uses. This already eliminates spurious *WAR* and *WAW* dependences that would

Polyhedral Expression Propagation [DSH18] was developed outside the LLVM/POLLY project and is available in the `expression_propagation` branch of our research prototype. We proposed an initial, less powerful version to the LLVM/POLLY community which was, in parts, accepted for integration[a] but never upstreamed. LLVM/POLLY originally had a basic, non-polyhedral, scalar expression propagation (ref. [Gro11, Section 5.6.2]) that was later removed. In a different project [MDH16] we used a similar, but more aggressive version, to avoid scalar dependences crossing synchronization barriers. Recently, a new simple scalar forwarding technique was added to LLVM/POLLY [KG18].

[a] Online review: `https://reviews.llvm.org/D13611`

```
u[k][j-1][i  ][1];
u[k][j-1][i  ][2];
u[k][j-1][i  ][4];
u[k][j  ][i-1][0];
u[k][j  ][i-1][2];
u[k][j  ][i  ][0];
u[k][j  ][i  ][1]
u[k][j  ][i  ][2];
u[k][j  ][i+1][0];
u[k][j  ][i+1][2];
u[k][j+1][i  ][1];
u[k][j+1][i  ][2];
u[k][j+1][i  ][4];
```

Figure 5.27: All accesses to the array `u` in the third and last loop of Figure 5.29.

have prevented LLVM/POLLY from parallelizing the enclosing loop nests. Since expression propagation is rooted on the iteration-wise, polyhedral memory dependences, we are not limited to intra-iteration expression propagation but can also eliminate all accesses to the temporary array `vs`. Similar to the scalar variable cases, the defining expression replaces the uses. However, this time the expression has to be adjusted to account for the loop nest change and the differences in the index functions used to access the array cells. Finally, the defining expression for the elements of `rhs` shown in statement `P0` can be propagated to statement `Q4`, but only if the definition in statement `P0` is afterwards eliminated. If it would not be, the effect of the self-overwrite in statement `P0` would be applied twice which would falsify the result. After expression propagation was applied as described here, the code looks like the one shown in Figure 5.29. All scalar definitions, the temporary array, and the intermediate self-overwrite have been removed after their effect was propagated into their users. Note that this simplified the first two loop nests but complicated the third one. The effect of the propagation is therefore often dependent on hardware specifications such as the number of available (vector) registers and the cache dimensions with regards to the memory footprint. In this example, propagation mostly introduced already existing, or similar, accesses to the array `u` into the third loop body. Thus, the number of cache misses can even decrease. In Figure 5.27 all accesses to the array `u` after expression propagation (ref. Figure 5.29) are shown in order. If the cache hierarchy can hold two complete rows in the `i` dimension, or alternatively hardware prefetchers are able to pick up on the access patterns, only seven other memory locations are accessed in one iteration of the innermost loop in the last loop nest. However, we only show parts of the original BT code in these examples and the full code would require even more registers to hold all values loaded in one iteration. If insufficient registers are available, spilling has to be performed which can again decrease the performance.

In the remainder of this Section we will first define *propagation expressions* and *propagation* dependences in Section 5.3.1. The former are the expressions which actually replace uses and the latter are dependences that identify the connections between definitions and uses which allow propagation. Afterwards, we look at non-surjective propagation dependences in Section 5.3.2. Hence, connections between definitions and some, but not all, instances of a use in the target statement. In Section 5.3.3 the complexity of expression propagation is discussed before we introduce limitations and extensions in Section 5.3.4. Prior to the evaluation in Section 5.3.7, we introduce several heuristics in Section 5.3.5 and implementation choices in Section 5.3.6, including our live-out access analysis to identify temporary memory.

```
      static double vs[KMAX][JMAXP+1][IMAXP+1];

      for (k = 0; k <= grid_points[2]-1; k++) {
        for (j = 0; j <= grid_points[1]-1; j++) {
          for (i = 0; i <= grid_points[0]-1; i++) {
S0:         rho_inv = 1.0/u[k][j][i][0];
S1:         rho_i[k][j][i] = rho_inv;
S2:         vs[k][j][i] = u[k][j][i][2] * rho_inv;
            // [...]
          }
        }
      }
      for (k = 1; k <= grid_points[2]-2; k++) {
        for (j = 1; j <= grid_points[1]-2; j++) {
          for (i = 1; i <= grid_points[0]-2; i++) {
P0:         rhs[k][j][i][2] = rhs[k][j][i][2] + dx3tx1 *
              (u[k][j][i+1][2] - 2.0*u[k][j][i][2] +
               u[k][j][i-1][2]) + xxcon2 * (vs[k][j][i+1] -
               2.0*vs[k][j][i] + vs[k][j][i-1]) - tx2 *
              (u[k][j][i+1][2]*up1 - u[k][j][i-1][2]*um1);
            // [...]
          }
        }
      }
      for (k = 1; k <= grid_points[2]-2; k++) {
        for (j = 1; j <= grid_points[1]-2; j++) {
          for (i = 1; i <= grid_points[0]-2; i++) {
Q0:         vijk = vs[k][j][i];
Q1:         vp1  = vs[k][j+1][i];
Q2:         vm1  = vs[k][j-1][i];
Q3:         rhs[k][j][i][1] = rhs[k][j][i][1] + dy2ty1 *
              (u[k][j+1][i][1] - 2.0*u[k][j][i][1] +
               u[k][j-1][i][1]) + yycon2 * (us[k][j+1][i] -
               2.0*us[k][j][i] + us[k][j-1][i]) - ty2 *
              (u[k][j+1][i][1]*vp1 - u[k][j-1][i][1]*vm1);
Q4:         rhs[k][j][i][2] = rhs[k][j][i][2] + dy3ty1 *
              (u[k][j+1][i][2] - 2.0*u[k][j][i][2] +
               u[k][j-1][i][2]) + yycon2*con43 *
              (vp1 - 2.0*vijk + vm1) - ty2 *
              (u[k][j+1][i][2]*vp1 - u[k][j-1][i][2]*vm1 +
              (u[k][j+1][i][4] - square[k][j+1][i] -
               u[k][j-1][i][4] + square[k][j-1][i]) * c2);
            // [...]
          }
        }
      }
```

Figure 5.28: Excerpt from the C implementation of the BT benchmark [SJL11] that is part of the NAS parallel benchmark suite [Bai+91]. The dotted and dashed lines indicate propagation opportunities. In addition, the definition in statement P0 can be propagated to read access rhs[k][j][i][2] in statement Q4.

```
   for (k = 0; k <= grid_points[2]-1; k++) {
     for (j = 0; j <= grid_points[1]-1; j++) {
       for (i = 0; i <= grid_points[0]-1; i++) {
S1:      rho_i[k][j][i] = 1.0/u[k][j][i][0];              // rho_inv
         // [...]
       }
     }
   }
   for (k = 1; k <= grid_points[2]-2; k++) {
     for (j = 1; j <= grid_points[1]-2; j++) {
       for (i = 1; i <= grid_points[0]-2; i++) {
         // [...]
       }
     }
   }
   for (k = 1; k <= grid_points[2]-2; k++) {
     for (j = 1; j <= grid_points[1]-2; j++) {
       for (i = 1; i <= grid_points[0]-2; i++) {
Q3:      rhs[k][j][i][1] = rhs[k][j][i][1] + dy2ty1 *
           (u[k][j+1][i][1] - 2.0*u[k][j][i][1] +
            u[k][j-1][i][1]) + yycon2 * (us[k][j+1][i] -
            2.0*us[k][j][i] + us[k][j-1][i]) - ty2 *
           (u[k][j+1][i][1] *
            (u[k][j][i+1][2] * 1.0/u[k][j][i+1][0]) -      // vp1
           u[k][j-1][i][1] *
            (u[k][j][i-1][2] * 1.0/u[k][j][i-1][0]));      // vm1
Q4:      rhs[k][j][i][2] = (rhs[k][j][i][2] + dx3tx1 *
           (u[k][j][i+1][2] - 2.0*u[k][j][i][2] +
            u[k][j][i-1][2]) + xxcon2 * (
            (u[k][j][i+1][2] * 1.0/u[k][j][i+1][0]) -      // vs[k][j][i+1]
            2.0*(u[k][j][i][2] * 1.0/u[k][j][i][0]) +      // vs[k][j][i]
            (u[k][j][i-1][2] * 1.0/u[k][j][i-1][0]) -      // vs[k][j][i-1]
            tx2*(u[k][j][i+1][2]*up1-u[k][j][i-1][2]*um1)) +
           dy3ty1 * (u[k][j+1][i][2] - 2.0*u[k][j][i][2] +
            u[k][j-1][i][2]) + yycon2*con43 *
          ((u[k][j][i+1][2] * 1.0/u[k][j][i+1][0]) -       // vp1
          2.0*(u[k][j][i][2] * 1.0/u[k][j][i][0]) +        // vijk
            (u[k][j][i-1][2] * 1.0/u[k][j][i-1][0])) -     // vm1
             ty2 * (u[k][j+1][i][2] *
             (u[k][j][i+1][2] * 1.0/u[k][j][i+1][0]) -     // vp1
             u[k][j-1][i][2] *
             (u[k][j][i-1][2] * 1.0/u[k][j][i-1][0]) +     // vm1
           (u[k][j+1][i][4] - square[k][j+1][i] -
            u[k][j-1][i][4] + square[k][j-1][i]) * c2);
         // [...]
       }
     }
   }
```

Figure 5.29: The code snippet shown in Figure 5.28 after expression propagation was performed. All scalars have been eliminated as well as the temporary array vs.

5.3.1 Propagation Expressions and Dependences

Conceptually, *expression propagation* replaces read accesses with the expressions last written to the read location. These expressions are consequently evaluated later, at a program point where they are needed, as an alternative to earlier evaluation followed by communication of their result. To perform expression propagation we require *propagation expressions*, that will replace read accesses, and *propagation* dependences, that relate read and write statement instances.

A *propagation dependence* $w \rightharpoonup r$ is a subset of a read-after-write (*RAW*) dependence $w \rightarrow r$ for which a *propagation expression* $\overrightarrow{e_w}$ exists. The propagation expression has to evaluate at the target statement instance $\mathbf{j}$ to the same value that was written in the dependent source statement instance $\mathbf{i}$. Thus, if we denote the defining expression written at instance $\mathbf{i}$ as e_w, Formula 5 and 6 define sufficient conditions for both propagation dependences as well as expressions.

$$w \rightharpoonup r \subseteq w \rightarrow r \tag{5}$$

$$\forall (\mathbf{i}, \mathbf{j}) \in w \rightharpoonup r \;:\; [\![\, e_w \,]\!]_{\mathbf{i}} = [\![\, \overrightarrow{e_w} \,]\!]_{\mathbf{j}} \tag{6}$$

Figure 5.30 provides an example for expression propagation together with the propagation dependences and expressions. The original program, shown in part 5.30a, features two read accesses of array `A`, r_0 and r_1, both in statement `T`. The values that are read have been written by the access w in statement `S`. The propagation dependences, $w \rightharpoonup r_0$ and $w \rightharpoonup r_1$, are shown in part 5.30c (left). In this example they are equal to the read-after-write (*RAW*) dependences $w \rightarrow r_0$ and $w \rightarrow r_1$. The propagation expressions, $\overrightarrow{e_{w_0}}$ and $\overrightarrow{e_{w_1}}$, are illustrated in part 5.30c (right). They are derived from the original source expression e_w := `(i+1)*i` written in statement `S` and adjusted to evaluate to the same value that would have been read by r_0 and r_1 in statement `T`. After propagation, thus in part 5.30b, the writes in statement `S` are obsolete if they are not live-out (ref. Section 5.3.6.1).

```
   for (i = 1; i < 2*N-1; i++)
S:   A[i] = (i+1)*i;
   for (j = 1; j < N; j++)
T:   B[j] = A[2*j] + A[2*j-1];
```

(a) Input program with one write w and two reads, r_0 and r_1, of array `A`.

```
   for (i = 1; i < 2*N-1; i++)
S:   A[i] = (i+1)*i;
   for (j = 1; j < N; j++)
T:   B[j] = 8*j*j;
```

(b) Program after propagation and simplification of the expression in statement `T`.

$$w \rightharpoonup r_0 := w \rightarrow r_0 = \{(\mathbf{i}, \mathbf{j}) \mid \mathbf{i} = 2 * \mathbf{j}\} \qquad \overrightarrow{e_{w_0}} := \texttt{((2*j)+1)*(2*j)}$$

$$w \rightharpoonup r_1 := w \rightarrow r_1 = \{(\mathbf{i}, \mathbf{j}) \mid \mathbf{i} = 2 * \mathbf{j} - 1\} \qquad \overrightarrow{e_{w_1}} := \texttt{((2*j-1)+1)*(2*j-1)}$$

(c) Propagation dependences $w \rightharpoonup r$ and expressions $\overrightarrow{e_w}$ for the two read accesses in part 5.30a.

Figure 5.30: Propagation example[a] for the input program in part 5.30a. The propagation dependences and expressions are shown in part 5.30c. The result after propagation and expression simplification is illustrated in part 5.30b.

[a] A similar Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

5.3.1.1 Semantic Preserving Expression Propagation

Expression propagation preserves the semantics if the evaluation of the propagation expression $\overrightarrow{e_w}$ is equal to the value loaded by the access r, for all iterations in the range of the propagation dependence $w \rightharpoonup r$. Thus, soundness is guaranteed if Formula 7 holds.

$$\forall (\mathbf{i}, \mathbf{j}) \in w \rightharpoonup r : \; \llbracket \overrightarrow{e_w} \rrbracket_{\mathbf{j}} = \llbracket r \rrbracket_{\mathbf{j}} \tag{7}$$

If we apply the definition of propagation expressions (Formula 6) we get Formula 8.

$$\forall (\mathbf{i}, \mathbf{j}) \in w \rightharpoonup r : \; \llbracket e_w \rrbracket_{\mathbf{i}} = \llbracket r \rrbracket_{\mathbf{j}} \tag{8}$$

Formula 5 defines propagation dependences $w \rightharpoonup r$ as subsets of *RAW* dependences [Fea91]. Thus, e_w written by w in iteration $\mathbf{i}$ is the value read by r in iteration $\mathbf{j}$, for all $(\mathbf{i}, \mathbf{j}) \in w \rightharpoonup r \subseteq w \rightarrow r$.

5.3.1.2 Propagation Dependences

Propagation dependences and expressions, as defined in Formula 5 and 6, depend on each other. To determine a non-trivial propagation dependence $w \rightharpoonup r$ for a *RAW* dependence $w \rightarrow r$ we require some information on the propagation expression $\overrightarrow{e_w}$. Only then it becomes possible to exclude iteration pairs for which Formula 6 does not hold. While there are infinitely many possible propagation expressions to choose from, we restrict ourselves in two important ways. First, we ensure that the propagation expression $\overrightarrow{e_w}$ will have the same structure as the original expression e_w. Second, a propagation expression $\overrightarrow{e_w}$ will access exactly the same scalar and memory locations as e_w did. Given these restrictions, the maximal propagation dependence is the maximal subset of the *RAW* dependence for which the evaluation of all contained read accesses does not change between related iterations. Thus, if $R(e_w)$ denotes the locations read by e_w, we can can define the maximal propagation dependence $w \rightharpoonup r$ as shown in Formula 9.

$$w \rightharpoonup r := \left\{ (\mathbf{i}, \mathbf{j}) \in w \rightarrow r \mid \forall\, r_{e_w} \in R(e_w).\; \llbracket r_{e_w} \rrbracket_{\mathbf{i}} = \llbracket r_{e_w} \rrbracket_{\mathbf{j}} \right\} \tag{9}$$

A practical and sufficient condition for a read access to evaluate to the same value in two different statement instances is the absence of an intermediate write to the same location. Our *reload test* will determine conditions for the absence of intermediate writes for each read in the propagation expression, thus $\forall\, r_{e_w} \in R(e_w)$. Afterwards we restrict the original *RAW* dependence with regards to all these conditions in order to get a valid propagation dependence. While it can potentially be smaller than the maximal one defined in Formula 9, we believe it is reasonable to expect that writes generally alter the values stored in the written location.

Reload Test

The *reload test* determines conditions that exclude intermediate writes to locations read by the propagation expression $\overrightarrow{e_w}$, or equivalently, by the original expression e_w. These conditions will later restrict a *RAW* dependence $w{\to}r$ to a valid propagation dependence $w{\rightharpoonup}r$ as they ensure that each read access evaluates to the same value in the propagation source as well as target iteration instance. Thus, the reload test derives conditions RT on the propagation target instances $\mathbf{j}$ such that Formula 10 holds for all read access $r_{e_w} \in R\left(e_w\right)$. It is important to note that is only true due to the constraints on the propagation expression. These ensure that the location accessed by $r_{e_w} \in R\left(e_w\right)$ in iteration $\mathbf{i}$ is equal to the one accessed by $\overrightarrow{r_{e_w}}$ in iteration $\mathbf{j}$ for $\left(\mathbf{i},\mathbf{j}\right) \in w{\rightharpoonup}r \subseteq w{\to}r$.

$$\forall\left(\mathbf{i},\mathbf{j}\right) \in w{\to}r.\ \mathbf{j} \in RT(r_{e_w}) \implies [\![r_{e_w}]\!]_{\mathbf{i}} = [\![\overrightarrow{r_{e_w}}]\!]_{\mathbf{j}} \tag{10}$$

To determine RT, we first relate the iteration instances of *potential intermediate writes* (*PIW*) to instances of the propagation target. An access is potentially intermediate, if it is the first succeeding write to a location read in the propagation expression. To identify potential intermediate writes we follow all *WAR* dependences emanating from the read accesses in the propagation expression. The targets of the *WAR* dependences are then related to the statement instances of the propagation target that will read the same location. The *PIW* relation for a propagation expression read $r_{e_w} \in R\left(e_w\right)$ is shown in Formula 11. It is the application of the *RAW* dependences onto the inverted *WAR* dependences of r_{e_w}. Thus, *PIW* relates write instances $\mathbf{l}$, that overwrite the instance $\mathbf{i}$ of r_{e_w}, with the instances $\mathbf{j}$ of the propagation target that will access the same location.

$$\begin{aligned} PIW\left(r_{e_w}\right) &:= \left(r_{e_w}{\to}*\right)^{-1} \circ \left(w{\to}r\right) \\ &= \left\{\left(\mathbf{l},\mathbf{j}\right) \mid \forall\mathbf{i}.\left(\mathbf{i},\mathbf{j}\right) \in w{\to}r \wedge \left(\mathbf{l},\mathbf{i}\right) \in r_{e_w}{\to}*\right\} \end{aligned} \tag{11}$$

An intermediate write exists, if and only if the write, hence the first component of $PIW(r_{e_w})$, precedes the reload, thus the second component. If it does not, the write will be executed after the propagation source as well as target instances and it will consequently not interfere with the propagation. To determine the order of the accesses we employ the schedule θ and the lexicographic ordering of statement instances $\ll_{lex}$. In addition we will require that $\ll_{lex}$ does order accesses inside a single statement to prevent overwrites that happen in the same statement and iteration. However, if the statement granularity ensures a single write per statement, e.g., C/C++ statements without the assignment operator, this is trivially fulfilled. Formula 12 shows the set of overwritten instances OR for a read access r_{e_w} in the propagation expression. Only these instances change between the propagation source and target instance, as only for these instances there is an intermediate write. Thus, these read instances cannot be reloaded at the target.

$$\begin{aligned} OR(r_{e_w}) &:= rng\left(PIW(r_{e_w}) \cap \left\{\left(\mathbf{l},\mathbf{j}\right) \mid \theta\left(\mathbf{l}\right) \ll_{lex} \theta\left(\mathbf{j}\right)\right\}\right) \\ &= \left\{\mathbf{j} \mid \left(\mathbf{l},\mathbf{j}\right) \in PIW(r_{e_w}) \wedge \theta\left(\mathbf{l}\right) \ll_{lex} \theta\left(\mathbf{j}\right)\right\} \end{aligned} \tag{12}$$

The reload test conditions *RT* that exclude intermediate writes and thereby ensure that a read in the propagation expression can be reloaded at the propagation target is shown in Formula 13.

$$\begin{aligned} RT(r_{e_w}) &:= \neg OR(r_{e_w}) \\ &= \{\, \mathbf{j} \mid \; (\mathbf{l},\mathbf{j}) \notin PIW(r_{e_w}) \;\lor\; \theta(\mathbf{j}) = \theta(\mathbf{l}) \;\lor\; \theta(\mathbf{j}) \ll_{lex} \theta(\mathbf{l}) \} \end{aligned} \tag{13}$$

This allows to define the propagation dependence $w \rightharpoonup r$ as shown in Formula 14. It is the subset of a *RAW* dependence $w \rightarrow r$, for which the target instances $\mathbf{j}$ fulfill the reload test conditions *RT* of every read r_{e_w} in the original, or alternatively, the propagation expression.

$$\begin{aligned} w \rightharpoonup r &:= \bigcap_{r_{e_w} \in R(e_w)} \left(w \rightarrow r \cap_{rng} RT(r_{e_w}) \right) \\ &= \left\{ (\mathbf{i},\mathbf{j}) \in w \rightarrow r \mid \forall r_{e_w} \in R(e_w).\; \mathbf{j} \in RT(r_{e_w}) \right\} \end{aligned} \tag{14}$$

Intra-Iteration *WAR* Dependences

```
for (i = 0; i < N; i++)
  A[i] = A[i] + A[i-1];
```

Figure 5.31: Example featuring a loop carried inter-iteration *RAW* dependence as well as an intra-iteration *WAR* dependence.

Propagation dependences are sensitive to intermediate writes that occur between iterations related through a *RAW* dependence. To determine the existence of intermediate writes our reload test uses the same *WAR* dependences that are computed for common polyhedral scheduling optimizations. Since most optimizations do not require intra-iteration dependences, only inter-iteration dependences are computed [Fea91]. The subtle difference is illustrated in Figure 5.31. The only inter-iteration dependence is the loop carried *RAW* dependence of length one. Due to the absence of inter-iteration *WAR* dependences we could falsely assume propagation is sound. However, the self-overwrite, or the consequent intra-iteration *WAR* dependence, should prevent propagation. To ensure soundness we therefore compute intra-iteration *WAR* dependences ourselves and use them in the reload test. Intra-iteration *WAR* dependences are computed as shown in Formula 15. For each statement `S` in the SCoP, the union of all read access functions is inverted and applied to the union of the inverted written access functions. Afterwards, all intra-iteration dependences, thus identities, are extracted.

$$\textit{intra-WAR} := \bigcup_{\mathtt{S} \in SCoP} \left(\left(\bigcup_{r \in \mathtt{S}} f_r \right)^{-1} \circ \left(\bigcup_{w \in \mathtt{S}} f_w \right)^{-1} \right) \cap \{ (\mathbf{i},\mathbf{i}) \} \tag{15}$$

At this point it is important to note that intra-iteration dependences are only propagation preventing if the location is live-out. If not, the write is eliminated after propagation and can therefore not overwrite the propagated read accesses. This special case is needed to propagate the `rhs` array accesses from statement `P0` to statement `Q4` in the motivating example shown in Figure 5.28

5.3.1.3 Propagation Expressions

One task for expression propagation is to identify a valid propagation expression and propagation dependence for a given *RAW* dependence. Once the propagation expression was determined, the valid part of the *RAW* dependence becomes the propagation dependence and expression propagation can be applied. However, there are indefinitely many different syntactic expressions. In order to effectively identify propagation expressions we therefore limit ourselves to the ones that can be constructed from the original expression via induction variable adjustment. This is usually a necessary step as their scope is limited and their values change when the loop progresses.

Expression Rewriting

To derive the propagation expressions we traverse the written expression e_w recursively and rewrite all induction variables $iv \in e_w$ according to *RAW* relation $w{\rightarrow}r$. The goal is to construct a new expression $\overrightarrow{iv}$ that depend on the iteration vector of the target and that evaluates to the same value there as iv evaluated to in the source for dependent iterations.

Without loss of generality we assume iv is the induction variable of the k-th out of n loops surrounding the source statement. The affine function that provides the *value of iv* for an iteration $\mathbf{i}$ of the source statement is defined as

$$v(iv) := \{(\mathbf{i}, iv)\} = \{((i_1, \dots, i_n), iv)\} = \{((i_1, \dots, i_n), i_k)\}.$$

Since $w{\rightarrow}r$ is an affine relation between instances of the statements surrounding w and r it can be written as

$$\{(\mathbf{i},\mathbf{j}) \mid f(\mathbf{i},\mathbf{j})\} = \{((i_1, \dots, i_n),\mathbf{j}) \mid f((i_1, \dots, i_n),\mathbf{j})\}$$

where f is a Presburger formula that defines the constrains under which the dependence exists. To obtain the function $v(\overrightarrow{iv})$ we apply the dependence $w{\rightarrow}r$ to the domain of $v(iv)$:

$$v(\overrightarrow{iv}) := v(iv) \circ w{\rightarrow}r = \{(w{\rightarrow}r(\mathbf{i}), iv) \mid f(\mathbf{i},\mathbf{j})\} = \{(\mathbf{j}, iv) \mid f'(\mathbf{j}, iv)\} \qquad \textbf{(16)}$$

The new Presburger formula f' relates instances of the target statement $\mathbf{j}$ to the value of $iv = i_k$ of the source statement.

Using common polyhedral code generation techniques [GVC15] we can generate the expression $\overrightarrow{iv}$ from $v(\overrightarrow{iv})$ which can then be evaluated in the target statement.

To derive the complete propagation expressions $\overrightarrow{e_w}$ we use the rewrite procedure presented in Algorithm 5.32 on the original expression e_w and the *RAW* dependence $w{\rightarrow}r$.

```
procedure REWRITE( $e_w$ : $\langle rexp \rangle$, $w \to r$ : RAW)
  switch $e_w$ do
    case $c$ : $\langle constant \rangle$:   return $c$
    case $p$ : $\langle param \rangle$:      return $p$
    case $iv$: $\langle iv \rangle$:         return $\vec{iv}$                  see equation (16)
    case l ⊙ m : $\langle rexp \rangle \times \langle rexp \rangle$:            ⊙ is a binary operator
      return REWRITE(l, $w \to r$) ⊙ REWRITE(m, $w \to r$)
    case A[$e_1$][...][$e_n$] : $\langle acc \rangle$:
      $f_k \leftarrow$ REWRITE($e_k$, $w \to r$)                          $1 \le k \le n$
      return A[$f_1$][...][$f_n$]
  end switch
```

Algorithm 5.32: Expression rewrite algorithm[a] that translates the originally written expression e_w according to the *RAW* dependence $w \to r$ to the propagation expression $\vec{e_{w_w}}$.

[a] This Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

5.3.2 Syntactic Read Replacement

We now established that the read access of a propagation dependence can be replaced by the propagation expression for all iterations in the range of the propagation dependence. In order to allow syntactic replacement of the access we additionally require the propagation dependence to be surjective with regards to the iteration domain of the target statement, thus $rng(w \rightharpoonup r) = \mathcal{D}_r$.

```
    for (i = 0; i < N; i++)
      if ((i/2)*2 == i)
S:        A[i] = f(i);

    for (j = 0; j < N; j++)

T:    B[j] = A[j];
```

(a) Program with a propagation opportunity for even elements of A but not for odd ones.

```
    for (i = 0; i < N; i++)
      if ((i/2)*2 == i)
S:        A[i] = f(i);

    for (j = 0; j < N; j++)
      if ((j/2)*2 == j)
T':       B[j] = f(j);
      else
T:        B[j] = A[j];
```

(b) Resulting program after propagation of $f(i)$ to statement T' which was split of from T.

Figure 5.33: Statement splitting and syntactic read replacement for a non surjective propagation dependences[a].

[a] This Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

It is always possible to achieve a surjective propagation dependence by splitting the target statement in two parts as illustrated in Figure 5.33. One part is completely reached by the dependence, here T' in Figure 5.33b, and one is not reached at all. All accesses, as well as the dependences involving the accesses, will be duplicated and then restricted to the domain of the respective part.

5.3.3 Complexity Analysis

Maximal expression propagation is *NP-hard* as it can be used to solve *k-CNF-SAT* formulae. The polynomial encoding for a *CNF* formula into a program in our input language is sketched in Figure 5.34. The outer conjunction is translated to an (arbitrary) operation referencing the results of all disjunctive clauses $c_1, \dots, c_n$ as illustrated in Figure 5.34a. The expression result is stored in the *only non live-out variable* named `UNSAT`. Afterwards, all clause results are overwritten. Consequently, propagation of `UNSAT` is possible if and only if prior propagation happened for all $c_1, \dots, c_n$. Each clause result c_i is defined as an (arbitrary) operation on the contained (negated) literals $l_1, \dots, l_k$ as shown in Figure 5.34b. Propagation of a clause result c_i is only possible if there exits a witness literal l_{wit_i}, which was replaced by its definition and of which the definition does not access the `False` array at position $\gamma(l_{\text{wit}_i})$. Note that $\texttt{wit}_i$ is a parameter and $\gamma(\cdot)$ a constant valued, injective enumeration function for literals that is not present in the code but used only for the construction. We define a literal l_i as x^+ if it is a positive use of `x` and as x^- if it is a negated use. In Figure 5.34c the positive and negative occurrences of a literal `x` are encoded as a constant value and an access to the `False` array depending on the parameter A_x which determines the *assignment of x*. Both literal forms can be propagated to the definition of a clause but only the form that was assigned a constant can justify the propagation of the clause.

```
UNSAT = $c_1 \oplus \dots \oplus c_n$;
$c_1$ = 0; ...; $c_n$ = 0;
SAT = !UNSAT;
```

(a) Encoding of the conjunction $c_1 \wedge \dots \wedge c_n$. `UNSAT` can only be propagated if all disjunctions have been replaced. If that is the case, the formula is satisfiable, otherwise it is not.

```
$c_i$ = $l_1 \oplus \dots \oplus l_k$;
if (wit$_i$ == 1)
  False[$\gamma(l_1)$] = 0, $l_1$ = 0;
else if (wit$_i$ == 2)
...
else if (wit$_i$ == k)
  False[$\gamma(l_k)$] = 0, $l_k$ = 0;
```

(b) Encoding for a disjunction $c_i = l_1 \vee \dots \vee l_k$ that can only be propagated if the (negated) literal at position $\texttt{wit}_i$ does not access the `False` array.

```
$x^+$ = 1; $x^-$ = 1;
if ($A_x$)
  $x^-$ = False[$\gamma(x^-)$];
else
  $x^+$ = False[$\gamma(x^+)$];
```

(c) Encoding for a literal `x` in positive (x^+ := x) and negative (x^- := ¬ x) form. The condition A_x is a parameter that determines the assignment of `x`.

Figure 5.34: Encoding rules[a] for a *k-CNF-SAT* formula as a program that allows maximal polyhedral expression propagation to solve satisfiability. All variables except `UNSAT` are live-out. The $\gamma(\cdot)$ function is a constant valued, injective enumeration for literals l_i which can be positive x^+ or negative x^- uses of a variable `x`, thus $\gamma(\cdot)$ is not present in the final program.

[a] This Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

Maximal expression propagation will explore all possible assignment combinations by splitting statements based on the values of the parameters A_x and $\texttt{wit}_i$. Though, only the literals with a constant value are able to justify the propagation of a clause c_i under the condition that $\texttt{wit}_i$ is equal to such a literals position in the clause. If and only if there exists an assignment of the literals such that all disjunctions can be propagated, the `UNSAT` value, representing the conjunction, can be eliminated as well. Each fulfilling variable assignment is determined by the parameter values of $\mathtt{A}_x$ in the iteration domains of the statement splits in which `UNSAT` was made obsolete after propagation. To this end, we assume the propagation algorithm will record all iteration domains for which the non live-out variable `UNSAT` can be eliminated and thereby also record all fulfilling assignments.

Note that the encoding has to happen first for all literals (ref. Figure 5.34c), then for all disjunctions (ref. Figure 5.34b) and then for the conjunction (ref. Figure 5.34a).

5.3.4 Limitations & Extensions

The definition of propagation dependences and propagation expressions (ref. Formula 5 and 6) are kept general on purpose. Later extensions to our technique can thereby easily reuse our framework and theoretical results. Especially the following two current limitations could be tackled in the future to extend the applicability and benefit of polyhedral expression propagation:

1. Propagation expressions will read exactly the same memory locations as the original source expression. We especially do not introduce new memory locations or reuse existing ones to communicate overwritten values as needed to eliminate the array `tmp` in Figure 5.35a.
2. Propagation dependences are determined using the given schedule which is not altered during the process. While expression propagation can still be applied before or after polyhedral schedule optimization, we did not explore the possibility to use schedule changes to eliminate propagation prohibiting *WAR* dependences as illustrated in Figure 5.35b.

```
tmp[0] = 1; tmp[1] = 1;
for (i = 2; i < N; i++)
  tmp[i] = tmp[i-1] + tmp[i-2];
out = tmp[N];
```

(a) Naive Fibonacci computation that requires two new scalar variables to hold intermediate results in order to allow propagation and consequent elimination of the temporary array `tmp`.

```
   for (i = 0; i < N; i++)
     tmp[i] = A[i];
   for (j = 0; j < N; j++) {
S:   A[j] = 0;
T:   B[j] = tmp[j];
   }
```

(b) Propagation prohibiting overwrite in statement `S` that could be avoided by an interchange of statement `S` and `T`.

Figure 5.35: Examples[a] illustrating the limitations of the propagation expression and dependence construction as described in Section 5.3.1.3 and Section 5.3.1.2.

[a] This Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

Note that both limitations can limit propagation on general purpose code, but neither impacted our evaluation on the image processing pipelines which never reuse temporary memory locations (ref. Section 5.3.7). Also the propagation of scalar variables in static single assignment (SSA) programs is consequently not affected by them.

5.3.5 Propagation Heuristics & Algorithm

While expression propagation, as presented here, will never increase the memory requirements or introduce new dependence, it can easily cause performance degradation. An increased cache miss rate, due to additional memory accesses, and additional computations, caused by the re-evaluation of defining expressions, are the most common causes. Furthermore, statement splitting can introduce complex loop structures that are hard to execute efficiently. In order to determine if propagation will be beneficial we devised several heuristics to guide our propagation algorithm. The heuristics and the actual propagation algorithm are explained in the following.

5.3.5.1 Propagation Heuristics

In our evaluation, expression propagation is only performed if all five heuristics described in the following deem it beneficial. However, dependent application on a broader class of code will require more tuning effort and a more evolved cost model.

Non Live-Out Memory Heuristic

The optimization potential of expression propagation stems mainly from the elimination of *non live-out*, e.g., temporary or overwritten, memory accesses. If all *RAW* dependences emanating from such accesses have been eliminated, the original computation, as well as the write accesses, become obsolete. Access elimination can also lead to fewer *WAR* and *WAW* dependences. Additionally, the overall memory requirement and cache contention might decrease. While propagation of live-out accesses is not harder, there is generally less performance to gain. To this end, we will optimistically assume that only propagation of non live-out memory is beneficial.

Cache Miss Heuristic

The goal of the *cache miss heuristic* is to limit the number of required cache lines per iteration. To this end, we conservatively approximate this number under two assumptions. First, the cache is approximated for one iteration in isolation. Second, each access is located in the middle of a cache line. Consequently, only subsequent accesses to close-by memory locations will cause cache hits. We take these assumptions as the final schedule, and thereby actual memory access order also with regards to other statements, is not necessarily determined when the heuristic is evaluated. However, all accesses contained in the statement are known to be executed in order. For the example shown in Figure 5.36, all locations between `A[i-3]` and `A[i+3]` are assumed to be cached after `A[i]` was accessed. The heuristic will then iterate order over all memory accesses contained in the target statement in lexicographic, assuming propagation already replaced all reads to one array with their respective propagation expressions. The number of cache lines needed per iteration is then equal to the number locations which are accessed but were not assumed to be already in the cache. Note that only if an access is considered a cache miss, the locations surrounding the access will be assumed cached for all accesses to come.

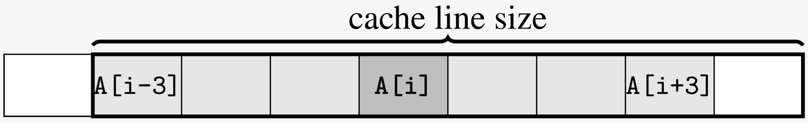


Figure 5.36: Memory locations that are assumed to be cached (gray) after `A[i]` (dark gray, center) was accessed. The cache line size is here 8 times the element size of `A`.[a]

[a] This Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

The number of required cache lines computed by our heuristic is not only approximative but there is additionally a hardware dependent component caused by prefetchers and tolerable misses due to computation latencies. We therefore determined a cache miss limit for each architecture using measurements similar to the ones shown in Section 5.3.7.1.

Code Complexity Heuristic

A key advantage of our propagation scheme is the ability to propagate expressions to some, but not all, iterations of a statement (ref. Section 5.3.2). However the required statement splitting can not only increase the number of statements, but also the number of dependences, exponentially. Additionally, splits can severely increase the complexity of the generated code. In the worst case a loop is split into three parts if the specialization happens in the middle iteration(s) of a loop dimension. The resulting loops, before and after the specialized iteration(s), are less often vectorized, due to a smaller trip-count, and can also increase the synchronization overhead, if they are separately parallelized. To limit code complexity we restrict *single instance specialization* in the following way: Instances are only specialized for a single, thus fixed, iteration if the specialized iteration will not create two non-fixed, thus loop dimensions, at the same nesting depth.

Two examples to showcase a different code complexity increase are illustrated in Figure 5.37. In the first part (ref. Figure 5.37a), the complexity is only slightly increased as the fixed single iterations will be the outermost ones of statement `T`. In the second example (ref. Figure 5.37b), propagation will increase the complexity more as the iteration domain of statement `T` will be split into two parts with `M/2` iterations each and a third one for the remaining `N-M` iterations.

```
    for (i = 1; i < N; i++)
      tmp[i] = In[i-1] + In[i+1];
    for (j = 0; j <= N; j++)
T:    Out[j] = tmp[j];
```

(a) Little complexity increase after specialization of all but the outermost iterations of statement `T`.

```
    for (i = M/2; i < M; i++)
      tmp[i] = In[i]*i;
    for (j = 0; j < N; j++)
T:    Out[j] = tmp[j];
```

(b) Large complexity increase after specialization of `M/2` out of `N` iterations of statement `T`.

Figure 5.37: Examples to showcase differences in code complexity increase that would arise from expression propagation with statement splitting.

Vectorization Heuristic

Loop vectorization is nowadays crucial to achieve good performance. However, expression propagation can propagate accesses, e.g., data dependent ones, that prevent (efficient) vectorization from one to multiple locations. To limit this effect propagate expressions is not allowed if it decreases the vectorization potential. Thus, no propagation shall increase the number of loops with non-vectorizable accesses, scaled by their nesting depth. Though, we take into account that the target statements might already contain non-vectorizable accesses and that the source statement could be eliminated after propagation.

Arithmetic Complexity Heuristic

If expressions are propagated into a deeper nested loop, the arithmetic complexity of the program is increased. Since this is generally not advantages, we do not allow the target to be deeper nested than the source. However, if the target domain would be specialized (ref. Section 5.3.2), then the number of remaining non-fixed, thus loop, dimensions is compared against the number of source statement loop dimensions.

5.3.5.2 Propagation Algorithm

To this point, we discussed expression propagation as an optimization performed per *RAW*, or propagation dependence. However, that could lead to the situation where only some of the outgoing *RAW* dependences of the writes to a non live-out location get propagated, while the rest is considered not beneficial. Since only the elimination of all non live-out write accesses allows to eliminate temporary memory completely, we want to avoid such partial propagations. To this end, we applied the heuristics and consequently also the propagation, not to a single dependence at a time, but always to the set of *RAW* dependences emanating from all writes to a non live-out array or scalar location[4]. Propagation is therefore only performed if it is *possible* and deemed *beneficial* for *all RAW* dependences involving a temporary location. Consequently, propagation will always allow to remove all writes to the temporary location afterwards.

To keep the algorithmic complexity low, we only try to propagate each non live-out array or scalar location once. The order in which the locations are visited is described below.

Propagation Order

Since our heuristics are checked for each non live-out location only once, the order in which they are visited is of special importance. This is true since propagation can change the heuristic results for later locations but potentially even their legality. In Figure 5.38, both effects are illustrated.

[4] Common scalar variables in a static single assignment (SSA) program have only one definition. However, our polyhedral representation of a phi node uses one write per incoming edge to emulate the assignment on the edge.

In the first example, Figure 5.38a, the propagation of the scalar `s0` and the scalar `s1` are mutually exclusive. Once either was propagated, the other cannot be, due to the *WAR* dependence between the two accesses to the array `A`. In the second example, Figure 5.38b, propagation of either scalar, `t0` or `t1`, will increase the number of required cache lines for statement `S` to three. If our heuristic only allows four cache misses per statement, the second scalar will not be propagated into `S`.

For our implementation we chose to propagate the temporary location first that cause the least number of different arrays accessed in the target statements. If there are multiple locations for which this number is equal, the one with the least amount of outgoing *RAW* dependences is chosen. If there is still no unique location, we pick the one with less: source statements, read accesses, dimensions and then number of incoming RAW dependences. The final tie breaker is the syntactic ordering in the source.

```
s0 = A[i];
s1 = s0 + B[i];
A[i] = ...;
C[i] = s1;
```

(a) Propagation of one scalar, `s0` or `s1`, prohibits propagation of the other one.

```
   t0 = A[i] + C[i];
   t1 = B[i] + D[i];
S: Out[i] = t0 + t1;
```

(b) Propagation of one temporary location, `t0` or `t1`, prevents beneficial propagation for the other one if only 4 cache misses are allowed.

Figure 5.38: Examples[a] illustrating the impact of the propagation order on propagation legality (part 5.38a) and the cache miss heuristic (part 5.38b).

[a] This Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

5.3.6 Implementation Details

Our expression propagation prototype is implemented in LLVM's polyhedral loop optimizer POLLY but it uses a fine-grained statement granularity which is also employed by high-level polyhedral optimizers such a Pluto [Bon+08]. To maximize performance in the presence of propagated expressions and statement splits we additionally had to augment LLVM and POLLY as described below.

Loop Parallelization

POLLY is capable of parallelizing loops using OpenMP [Rag11]. If enabled, the outermost parallel loop is chosen to be executed in parallel, even if the loop trip count prevents full utilization of the machine. To avoid undersubscription we decided to parallelize outermost loops with small loop trip counts only if there is no suitable nested loop available. This can lead to parallelization of innermost loops, something POLLY will not do by default. Finally, we also allow to parallelize loops with non-affine write accesses, if parallel annotations present in the benchmark sources guaranteed the absence of dependences (ref. Section 5.1.2).

Scheduling and Tiling

Polly does perform polyhedral scheduling and tiling [GGL12] optimizations as described by Bondhugula et al. [Bon+08]. However, it cannot perform smart loop fusion [Bon+10], choose suitable tile sizes [BPB12; Ham+17; Ham+18; Yuk+10] or do a combination thereof [MVB15]. Instead, the default scheduling choice will perform aggressive loop distribution/fission and loops are always tiled rectangularly with the default tile size of 32. In our experiments we regularly noticed performance regressions when tiling was done with the default tile size. Additionally, the default scheduling scheme did decrease the execution time when statements splits were placed into multiple loop nests.

To avoid these performance artifacts we disabled loop tiling altogether and modified the scheduling objective. While Polly tries to create independent loop nests for each statement, including all statement splits, we fuse all write accesses to the same array into one loop nest. This scheduling choice is similar to the naive inputs [MVB17]. However, our scheduling and tiling choices are far from optimal and they need to be revisited. Nevertheless, the evaluation in Section 5.3.7.3 shows that they almost always result in better performance than Polly's defaults.

Higher-Order Recurrences

Recurrences are scalar variables that communicate values from one iteration to the next [Kev90]. In order to employ recurrences, we augmented Polly's code generation to recognize consecutive accesses in the innermost loop that have been replaced by propagation expressions. Generally, all consecutive read accesses can be communicated via recurrences into the next iteration. Though, we decided to keep memory reads because they will most definitively cause low-level cache hits while recurrences will inevitably increase the register pressure of the whole loop. Higher-order recurrences, i.e., recurrences that communicate values across multiple loop iterations, are especially useful to avoid recomputation. An example in which higher-order recurrences reduce the number of evaluations of the function f after propagation was performed is shown in Figure 5.39. To vectorize higher-order recurrences we had to extend the LLVM loop vectorizer as it is by default limited to single-level recurrences only.

```
for (i = 0; i <= N; i++)
  tmp[i] = f(i);

for (j = 1; j < N; j++)
  Out[j] = tmp[j-1] + tmp[j]
         + tmp[j+1];
```

(a) Consecutive accesses that can be eliminated by propagation.

```
t0 = f(0); t1 = f(1);
for (j = 1; j < N; j++) {
  t2 = f(2);
  Out[j] = t0 + t1 + t2;
  t0 = t1; t1 = t2;
}
```

(b) Higher-order recurrences used to communicate the result of f across two loop iterations.

Figure 5.39: Use of higher-order recurrences to reduce computation overhead after expression propagation was performed[a]. Naive code generation would triple the number of instances of f, but recurrences allow propagation with only one evaluation of f.

[a] This Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

5.3.6.1 Live-Out Access Analysis

A live-out access analysis [RM88] determines if a write access can potentially change the values read after a certain program point. If so, a write is called live-out. Non live-out write accesses can be omitted if the stored value is also not read prior to the program point in question. Thus, if all reads of a non live-out location inside a SCoP can be eliminated through propagation, the write, and thereby the original computation, can be omitted as well.

Due to the lack of a suitable analysis in the LLVM framework we implement one ourselves[5]. It detects the following non-live out locations with regards to the currently optimized SCoP.

Scalar Locations

In static single assignment (SSA) programs a scalar is only live-out if it is used outside the SCoP.

Stack Memory Locations

Stack locations are only accessible during the current function invocation. If a stack memory location does not escape, thus its address is never stored in a global variable or passed to another function, it can consequently only be used inside the defining function. We identify stack locations as non live-out if they do not escape and there is no user reachable from the SCoP.

Global Memory Locations

Global memory locations are only considered non live-out if their address is not taken, all known uses are inside the SCoP and their declaration is "internal" to the optimization unit, e.g., `static` allocations in C/C++.

Unknown Memory Locations

A heap memory location or one with unknown origin, e.g., a pointer argument, is generally considered live-out. However, if we can determine that the content cannot legally be read[6] or the first reachable outside user post-dominates the SCoP and also deallocates the memory, we can conclude the memory is not live-out.

Overwritten Memory Locations

If a write to *any* location is overwritten inside the SCoP it is not live-out. To this end, we use the write-after-write (*WAW*) dependences that are already computed during the polyhedral dependence analysis [Fea91] to identify all overwritten locations.

[5] The identification of all but overwritten memory locations was implemented by Shrey Sharma.

[6] LLVM employs special lifetime intrinsics to mark memory that can be assumed dead.

5.3.7 Evaluation

Polyhedral expression propagation is evaluated on the seven benchmarks listed in Table 5.41 and on the four architectures listed in Table 5.40. The different architectures allow to showcase both scalability (4−20 threads) as well as portability with regards to the cache size, ISA, and execution model, thus In-Order vs. Out-of-Order (OoO). The benchmarks were first used to evaluate the PolyMage tool [MVB15] and are available online [MVB17] in a naive parallel and an optimized version. Note that the input for the LLVM/Polly based schemes was always the naive version of the benchmark. We compare expression propagation to the following polyhedral-model-based but scheduling-centric optimizations:

- vanilla[7] Polly, the polyhedral optimizer of the LLVM/Clang compiler which is also the basis for our approach,
- optimized code versions generated by PolyMage [MVB15], the state-of-the-art polyhedral optimization tool for concatenated stencil computations,
- Halide [Rag+13], a state-of-the-art DSL for schedule optimizations on process pipelines,
- our Polly fork[8] that was modified as described in Section 5.3.6 but without expression propagation, and
- our Polly fork with automatic expression propagation, as explained in Section 5.3.5.2, and guided by the heuristics presented in Section 5.3.5.1.

LLVM/Polly based approaches, as well as PolyMage, were compiled with the same LLVM[9] version (close to v4.0.1[10]). We choose this setup to compare the effects caused by the specific optimizations rather than artifacts that arise due to different vectorization or register allocation schemes employed by the compilers. Additionally, we present PolyMage results compiled with GCC[9] (v7.2.0) and ICC[9] (v18.0.1 20171018) on the Xeon E3-1225v3 CPU in Table 5.42.

CPU	# Cores / Threads	Vec. Bits	L1 Cache	LLC	Exec. Model
Cortex A53	4 / 4	128	16 KiB	0.5 MiB	In-Order
Cortex A57	4 / 4	128	24 KiB	2 MiB	OoO
Xeon E3-1225v3	4 / 4	256	128 KiB	8 MiB	OoO
Core i9-7900X	10 / 20	512	320 KiB	13 MiB	OoO

Table 5.40: Architecture details[a] including the CPU, number of cores and threads, vector size in bits, first (L1) and last level cache size (LLC) as well as the execution model.

[a] This Table was first presented by Doerfert, Sharma, and Hack [DSH18].

[7] Modifications to the SCoP detection and modeling were necessary to represent and optimize the benchmarks.

[8] Our Polly fork, including expression propagation, and evaluation scripts are available online at `https://github.com/cdl-saarland/PolyhedralExpressionPropagation`.

[9] LLVM/Clang and GCC were invoked with the options: “-fopenmp -ffast-math -march=native -O3”. ICC was run with the options: “-qopenmp -fp-model fast=1 -ftz -xhost -O3”.

[10] Our LLVM/Clang is a slightly modified version (ref. Section 5.3.6) based on git commit 1aa4ba7.

#	Name	LOC	*#Arrays*			*#Loops*				*#Statements*				*#Accesses*			
			N/P	**PM**	**EP**	**N**	**P**	**PM**	**EP**	**N**	**P**	**PM**	**EP**	**N**	**P**	**PM**	**EP**
(1)	unsharp mask	96	5	2	2	12	12	14	3	4	4	4	1	20	21	20	28
(2)	harris	153	12	2	2	22	22	8	2	11	11	3	1	65	61	159	26
(3)	bilateral grid	502	7	5	4	25	25	29	18	8	7	8	6	90	39	90	57
(4)	camera pipe	1114	30	6	5	67	67	84	4	41	35	41	13	288	146	276	675
(5)	pyramid blending	1501	43	11	14	123	123	184	31	58	52	58	23	272	268	272	967
(6)	interpolate	1801	39	27	19	186	186	292	71	182	182	182	53	294	294	294	299
(7)	local laplacian	7018	69	44	37	461	461	687	168	814	626	537	441	1533	1212	1070	3846

Table 5.41: Benchmark details[11] including the number of arrays, loops, statements and accesses for the *Naive* (N), POLLY (P), *PolyMage* (PM) and *Expression Propagation* (EP) version. The lines of code (LOC) are measured after code formatting.

#	Name	naive	vanilla Polly	forked Polly	expr. prop.	PM Clang	PM GCC	PM ICC	Halide
(1)	unsharp mask	47.79 ms	57.55 ms	47.55 ms	**8.53 ms**	9.11 ms	9.01 ms	9.00 ms	n/a
(2)	harris	82.12 ms	96.82 ms	82.23 ms	7.29 ms	**5.99 ms**	6.59 ms	9.61 ms	n/a
(3)	bilateral grid	24.53 ms	25.55 ms	24.00 ms	16.29 ms	23.21 ms	28.52 ms	26.20 ms	**15.25 ms**
(4)	camera pipe	40.40 ms	47.38 ms	35.13 ms	10.46 ms	14.09 ms	13.95 ms	11.68 ms	**5.76 ms**
(5)	pyramid blending	113.97 ms	145.554 ms	114.19 ms	53.73 ms	47.76 ms	45.94 ms	**41.40 ms**	n/a
(6)	interpolate	72.14 ms	78.83 ms	70.77 ms	**26.51 ms**	56.62 ms	40.99 ms	41.08 ms	36.64 ms
(7)	local laplacian	140.46 ms	n/a	270.30 ms	111.62 ms	97.11 ms	86.29 ms	**82.39 ms**	96.71 ms

Table 5.42: Raw performance results[11] for the Xeon E3-1225v3 architecture. The best performer is highlighted in bold. The Vanilla POLLY result is missing for benchmark (7) due to a code generation issue. For Halide only the results of available benchmarks (`https://github.com/halide/Halide/tree/release_2017_05_03/apps`) are reported. Note that PolyMage is listed as (PM).

[11] These Tables were first presented by Doerfert, Sharma, and Hack [DSH18]

5.3.7.1 Heuristics Comparisons

While a comparison of the different optimization schemes will follow in Section 5.3.7.3, we first want to discuss the effects of the heuristics we employed (ref. Section 5.3.5.1), but also of the ones we eventually discarded. In addition to the heuristics we described before, we experimented with others, including an approximation of the number of required registers and propagation based on the number of accesses. We also altered the propagation orders based on these and similar factors. While some combination of heuristics did result in performance improvements of up to 20% for individual benchmarks, the overall results were always inconclusive. To provide a better intuition, we illustrated the results of an extensive search over the optimization space graphically in Figure 5.43. The four plots show measurements taken after expression propagation of randomly chosen combinations of temporary arrays for the Xeon E3-1225v3 architecture. The size of the combinations varied between 1 and 31. If the benchmark allowed it we generated up to 31 unique sets of temporary arrays for each size. Expression propagation was then asked to propagate all *RAW* dependences that were caused by accesses to these temporary arrays without regards for any heuristic. If that was possible, the temporary array was afterwards removed. The data points in the plot report the median runtime results of 31 runs for each combination with regards to different static properties of the resulting code. The lines are linear regressions over

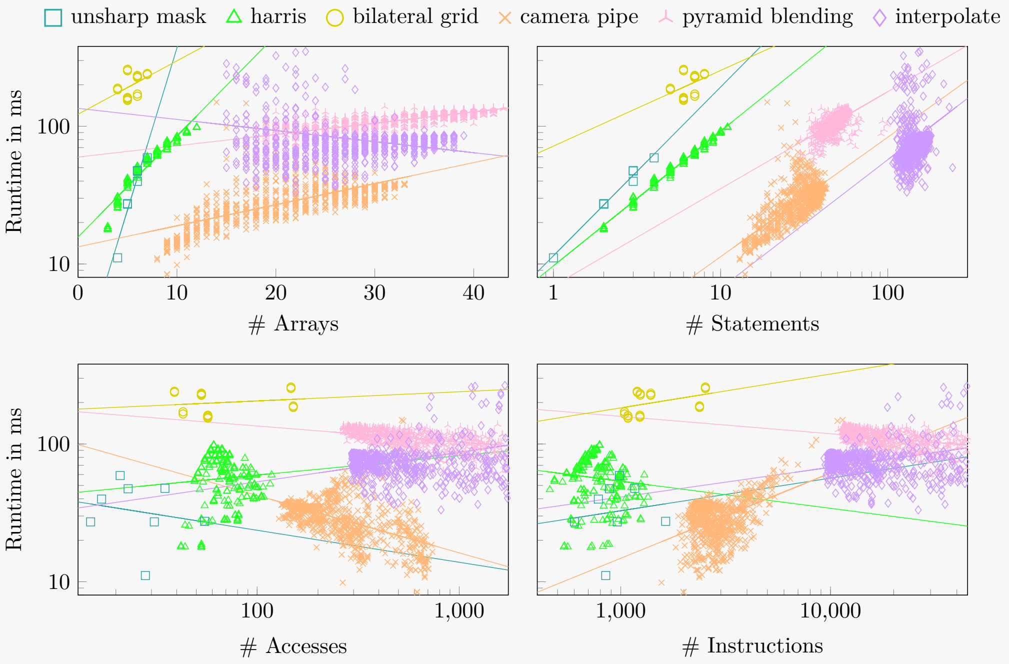


Figure 5.43: Correlation[a] between the runtime on the Xeon E3-1225v3 architecture and the number of: arrays (upper left), statements (upper right), accesses (lower left), and instructions (lower right). The marks indicate experimental results and the lines show the linear regression per benchmark. A positive regression line slope indicates that minimizing the number generally improves performance. However, the absolute value of the slope is highly dependent on the scaling and the shown range.

[a] This Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

all benchmark results. Note that we are interested in the sign of the regression line slope not the absolute value as the latter is heavily impacted by the scaling of the plot and the shown range.

The two graphs in the top row show the runtime with regards to the number of arrays and statements. The regression lines in these plots indicate that propagation, and the consequent array and statement elimination, has a positive impact on the performance. All benchmarks, except interpolate (6), generally perform better when expression propagation reduces the number of arrays and statements. Interpolate is different because propagation can easily cause too many statement splits that might reduce the number of arrays but complicate the schedule and thereby increase the runtime. If the number of statements is minimized (ref. second graph), propagation is beneficial. As a lesson of these findings we took the general assumption that propagation, and consequent elimination, of temporary arrays is beneficial if not deemed otherwise by a heuristic.

The two bottom graphs show that neither the number of static accesses nor the number of static instructions have a clear impact on the performance. Three of the seven benchmarks show speedups even though the number of accesses increases. The same holds true for two benchmarks with regards to the number of instructions. Consequently, we did not employ heuristics that reasoned about the number of static instructions or array accesses.

In Table 5.44 we provide the performance results of the naive parallel version, the heuristic guided polyhedral expression propagation we evaluated against other approaches, as well as the best and worst observed version in aforementioned experiment. The numbers provide two interesting insights about expression propagation and the heuristics we chose. First, if an unfavorable selection of temporary arrays was removed, performance regressed for each benchmark. The resulting slowdown reached from 1.05× for bilateral grid (3) up to 17.8× for interpolate (6). Second, even though a large number of combinations was tried, only camera pipe (4) performed better than our heuristic guided propagation. Since the search was exhaustive for the smaller benchmarks (1)-(3), we know that our heuristics worked perfectly for them on the tested architecture. For the remaining two benchmarks, pyramid blending (5) and interpolate (6), the random search got close to the results of our heuristic guided propagation but it did not quite reach it.

Version \ Benchmark	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**
naive implementation	47.8	82.1	24.5	40.4	114.0	72.1
expression propagation	8.5	7.2	16.3	10.5	53.7	26.5
rnd. propagation best	8.5	7.2	16.3	8.4	55.2	33.2
rnd. propagation worst	59.0	98.6	25.7	149.4	138.6	1285

Table 5.44: Performance results[a] (in ms) including the limits observed in the random propagation tests shown in Figure 5.43. See Table 5.41 for the benchmark names.

[a] This Table was first presented by Doerfert, Sharma, and Hack [DSH18].

To determine the effectiveness of the heuristics we chose to use (ref. Section 5.3.5.1), we performed several experiments. While the benchmarks only allow for propagation of non live-out memory, we did try different thresholds for the cache miss heuristic and also rerun our experiments without the code complexity heuristic.

Figure 5.45 shows the effects of different cache miss limits for three of the tested architectures. While the two smallest benchmarks, unsharp mask (1) and harris (2), performed best after propagation of all temporary arrays, others showed different behaviors. Especially bilateral grid (3) and pyramid blending (5) are interesting in this experiment because they are not restricted by the code complexity heuristic. Instead, the performance changes if an increased cache miss limit allows more propagations. Depending on the architecture, the performance either improves or regresses if more propagations are performed. For bilateral grid (3), the best performance is achieved for a cache miss limit between 20 and 48 on the Intel architectures (Core i9 and Xeon), while on the ARM Cortex A57 the performance drops already after a cache miss limit of 24. The reason is the different amount of available resources, e.g., registers, issuing ports and prefetcher streams. On the ARM architecture we can observe slight improvements for much higher cache miss limits because array elimination generally decreases the runtime while the resource contention does not necessarily increase. Pyramid blending (5) shows steady runtime improvements for higher cache miss limits on the Core i9 architecture, the most resource rich CPU we tested. However, on the Xeon CPU the runtime flattens out and on the Cortex A57 the performance is significantly worse when the cache miss limit exceeds 32. Our heuristic guided expression propagation uses a cache miss limit of 32 for the Intel CPUs and 22 for both ARM architectures.

Expression propagation without the code complexity heuristic has a similarly negative effect as a very large cache miss limit, though mainly interpolate (6) and local laplacian (7) are affected. Without the complexity restriction the performance of interpolate varies significantly on the Xeon CPU. Depending on the cache miss limit we see a speedup of up to 3.6× but also slowdowns of up to 17.8×, both with regards to the native parallel version. As a comparison, our default heuristics achieve a speedup of 2.7×. These number show again the vast performance differences that can be caused by expression propagation, even for a fixed benchmark and architecture. Consequently, the heuristics, including the propagation order (ref. Section 5.3.5.2), have to be chosen carefully and potentially tuned to the architecture in order to prevent regressions. At the same time it is crucial to account for other factors, e.g., tiling and scheduling and the use of recurrences (ref. Section 5.3.6), since they have a non-trivial impact on the profitability of a propagation.

We learned from these studies that a good performance model for expression propagation is a challenging task and a research topic on its own. To this end, we chose the relatively simple combination of heuristics explained in Section 5.3.5.1 to demonstrate the potential of expression propagation without regressions. Though, to maximize performance it is crucial to use a more elaborate cost model and to revisit the scheduling and tiling choices.

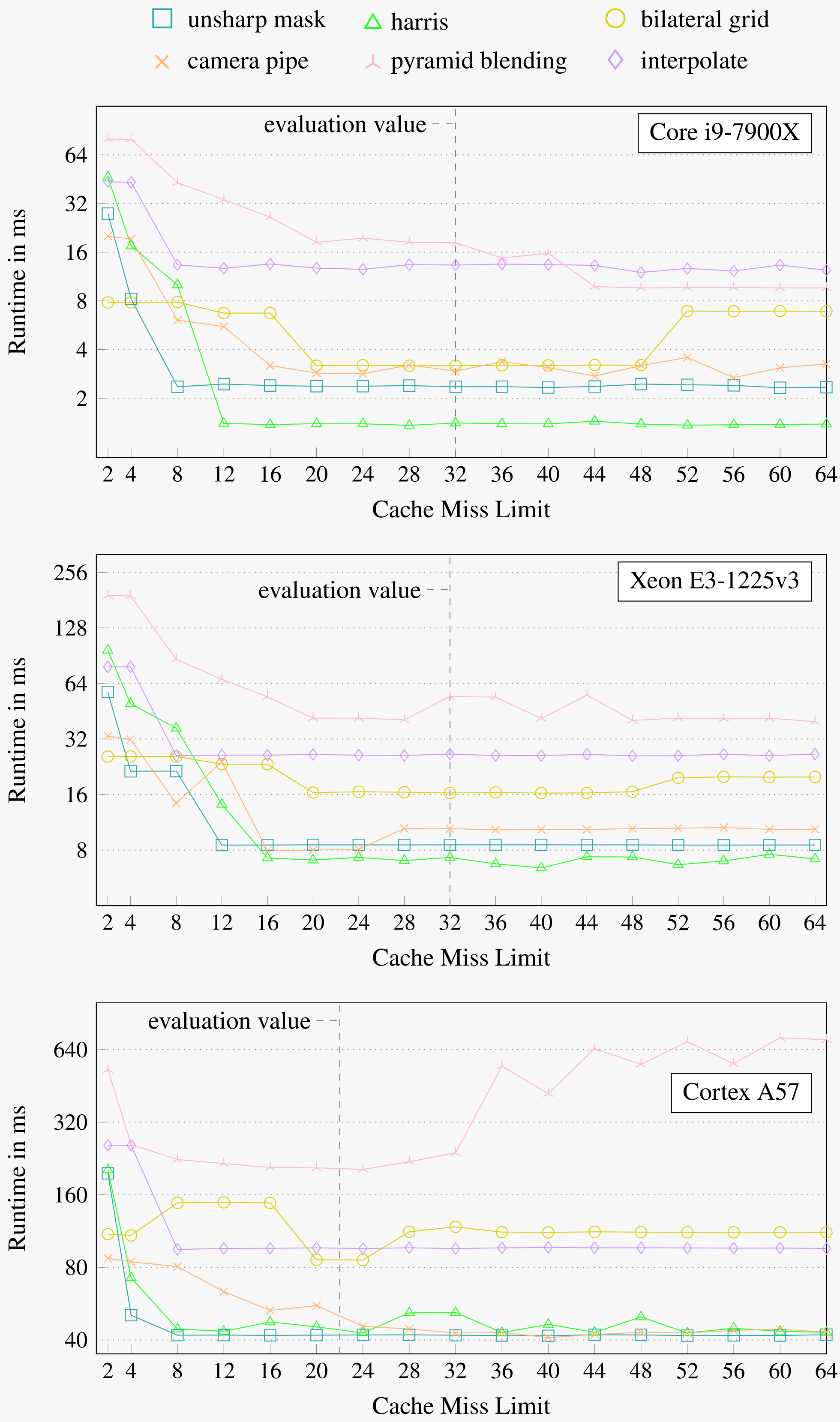


Figure 5.45: Effect of different cache miss limits[a] (Section 5.3.5.1) on the runtime for the Intel Core i9-7900X, Intel Xeon E3-1225v3 and, ARM Cortex A57 architectures.

[a] Parts of this Figure were first presented by Doerfert, Sharma, and Hack [DSH18].

5.3.7.2 Memory Requirement and Code Complexity

Expression propagation is not only a performance optimization but, as it can eliminate temporary memory locations, also a memory optimization. In Table 5.41 the number of parametric sized arrays before and after expression propagation is shown, together with other static properties of the optimized program. Both PolyMage [MVB15] and expression propagation decrease the number of parametric sized arrays for each benchmark. However, expression propagation completely eliminates temporary arrays while PolyMage often replaces them with smaller, constant-sizes, ones that hold the intermediate results for one tile at a time.

As noted earlier, two of the benchmarks perform best if all temporary arrays have been removed. While this is certainly not true for the three largest benchmarks, it already does not hold for the rather small bilateral grid (3) implementation. This benchmark performs best if only two of the three temporary arrays are eliminated and one is kept. Such effects are not the only problem when minimal memory usage is used to guide expression propagation. As shown in Table 5.41, there is a significant increase in the number of accesses per statement for each benchmark. While the cache miss heuristic does indirectly limit this number, we already observe a notable growth that causes an even bigger increase in the number of dependences. These dependences do not only limit later schedule optimizations but also increase the compile time for all subsequent steps. Consequently, expression propagation purely guided by array elimination does not scale well for programs above the size and complexity of pyramid blending (5) or interpolate (6).

5.3.7.3 Automatic Expression Propagation

In Figure 5.46 the performance of all five optimizations schemes is shown for each evaluated architecture. The graphs are normalized to the results of a naive parallel implementation generated by PolyMage and available online [MVB17]. This naive parallel implementation[12] is also the input for all three POLLY-based optimization schemes. Measurements were taken 51 times and the median result is reported. The PolyMage version [MVB15] was optimized for a processor similar to the Xeon E3-1225v3 and it was not specialized for the other architectures. However, it is important to note that vanilla POLLY does not employ target dependent tile sizes and that our POLLY fork, as well as expression propagation, do not perform tiling at all (ref. Section 5.3.6).

Except for bilateral grid (3) on the ARM Cortex A53, our POLLY fork performs consistently better than vanilla POLLY. For this benchmark and architecture the tiling employed by vanilla POLLY improves the performance. At the same time, our POLLY fork only performs better than the heuristic guided expression propagation for local laplacian (7) on the ARM Cortex A57.

[12] We stripped OpenMP pragmas from the input to allow POLLY's SCoP detection to recognize the loop nests.

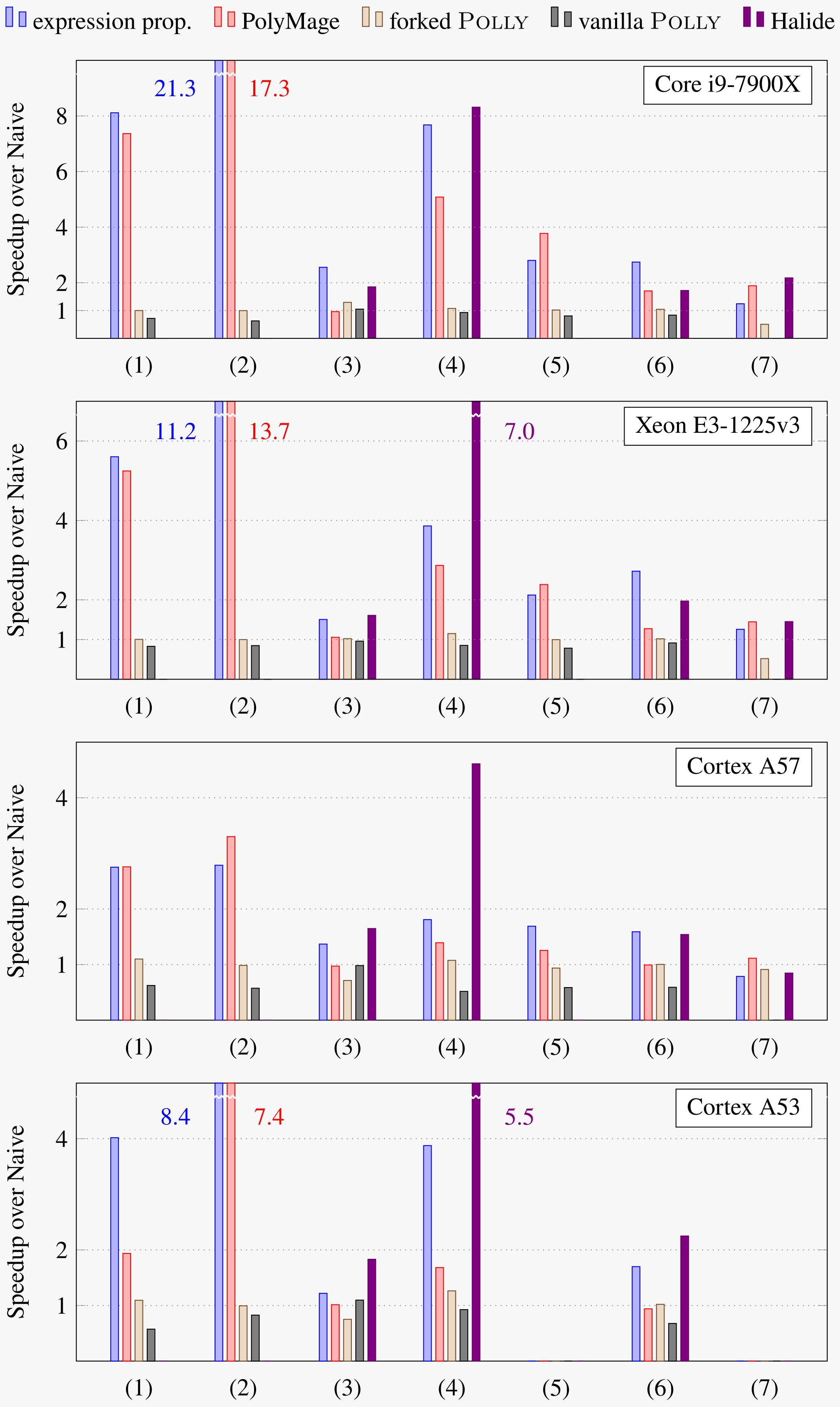


Figure 5.46: Performance[a] of different schemes (ref. Section 5.3.7) normalized with regards to the naive parallel version generated by PolyMage. The vanilla POLLY bar is missing for benchmark (7) due to a code generation issue. The bars for on the Cortex A53 are missing due to insufficient main memory. For Halide, only the results for benchmarks available in the tested release (`https://github.com/halide/Halide/tree/release_2017_05_03/apps`) are reported.

[a] This Figure was first presented by Doerfert, Sharma, and Hack [DSH18].

PolyMage is faster than the expression propagation optimization in 7 cases and consistently better for local laplacian (7). Expression propagation outperforms PolyMage in 18 cases and is, except for local laplacian on the ARM Cortex A57, consistently faster than the naive implementation. The highest speedups are achieved for the rather simple harris (2) benchmark. Though, depending on the system, either the tiled PolyMage code with two constant sized temporary arrays or our fully propagated version performs best.

Halide [Rag+13] results are only available for the benchmarks contained in the 2017/05/03 release[13]. While the performance for camera pipe (4) is consistently the best among all optimization schemes, our approach comes close on the Core i9 architecture. The performance of the remaining benchmarks is comparable to expression propagation and it depends on the platform which one performs best. However, it is important to consider that Halide and PolyMage perform more complex optimizations, including tiling and (non-trivial) scheduling. Both have not been explored in this work (ref. Section 5.3.6). Note that all expression propagation speedups are only due to the elimination of temporary arrays. There is especially no increase in parallelism, e.g., due to propagated scalars, in these benchmarks.

Real World Application

Expression propagation is often limited by the available live-out information (ref. Section 5.3.6.1). In real world code, scalars are often the only known non live-out values. Nevertheless, propagation of scalars is important to eliminate false dependences (*WAR* and *WAW*) which can decrease scheduling freedom. For the SPEC2000/2006 benchmark suites, expression propagation is able to eliminate 63%/60% of all scalar read accesses in SCoPs found by our POLLY fork.

5.3.8 Related Work

Polyhedral optimization techniques perform iteration domain splitting to facilitate specialized schedule optimization and code generation for the different statement splits [GFL99; MDH16]. In contrast, we only split statements to enable syntactic read replacement (ref. Section 5.3.2) and did not further investigate the effects on scheduling. Alternatively, we even ensure that all statement splits are fused into a single loop nest (ref. Section 5.3.6). The idea to split iteration domains to create surjective (propagation) dependences was already used by Wonnacott [Won00] as well as Vanbroekhoven, Corporaal, and Catthoor [VCC03].

The benefit of recurrences to avoid recomputation of values in subsequent loop iterations is well known [Kev90]. In contrast to most classical techniques, we only use recurrences to communicate expressions that replaced consecutive read accesses in the innermost loop (ref. Section 5.3.6).

[13] Online: `https://github.com/halide/Halide/tree/release_2017_05_03/apps`

While we could employ recurrences for consecutive innermost read accesses that were not replaced by a propagation expression, the recomputation cost of a single load is significantly less than for an arbitrary complex propagation expression.

Writes that do not change the values of any subsequent read are dead assignments which can be eliminated (ref. Section 5.3.6.1). Different techniques exist to identify, and consequently remove, non live-out scalar assignments [KRS94] as well as non live-out array writes [Ver15b]. In contrast to such elaborate approaches, we do not iterate the detection of dead assignments in order to identify transitively dead ones. While this might become necessary for more complex programs it was not needed for our evaluation.

As polyhedral scheduling optimizations are limited by the dependences that have to be preserved, there are various techniques that aim to identify and eliminate spurious and redundant ones. Especially false dependences (*WAR* and *WAW*) do not always need to be fulfilled to preserve the observable behavior of a program. To this end, approaches commonly identify situations where dependences exist which can be broken, or relaxed, in order to increase the scheduling freedom [Bag+13; Cal+97; Doe+15; MY16]. Expression propagation explicitly removes only true (*RAW*) dependences. Though, it implicitly eliminates false dependences as well, if they are caused by non live-out definitions that can be propagated. As an example consider the motivating example in Figure 5.28. The scalar definitions in statements `S0`, `Q0`, `Q1`, and `Q2` all induce spurious false dependences which would enforce the original loop iteration order. In addition, propagation of the array writes in statements `S2` and `P0` remove the dependences between the loop nests, potentially allowing us to execute them in parallel [Rac16].

Programs, especially if they have a precise polyhedral representation, can be translated into dynamic single assignment (DSA) form to eliminate all false dependences [Fea88a; Van+07]. In DSA form, every memory location is written at most once, which prohibits both *WAR* and *WAW* dependences. In contrast to other propagation techniques [Heg+14; Mul+16; Rag+12; VCC03], we do not require the program to be in DSA form. Instead, we take possible intermediate writes into account when we construct the propagation dependences (ref. Section 5.3.1.2). An example for which the original program version contains intermediate writes that prohibit propagation while the DSA form program would not is shown in Figure 5.35b on Page 157.

Transforming a program into DSA form will generally increase the memory requirement. To alleviate this increase, memory reduction techniques [BBC16; DSV03; DIY16; LF98; WR96] are often used after all DSA based optimizations have been applied. These memory reduction techniques perform employ polyhedral scheduling techniques on the memory access functions in order to reuse locations for different intermediate results. In contrast, expression propagation never changes the accessed memory but only the “time and place” where intermediate results are computed. Since our heuristics ensure that propagation removes all users of a non live-out definition we can eliminate it afterwards. If all definitions of a scalar or array have been removed, the actual location becomes obsolete. To this end, expression propagation can be used as a pre-processing step to reduce the amount of temporary memory locations while keeping all intermediate computations manageable for the target hardware (ref. Section 5.3.5.1). Future work could also combine memory scheduling techniques and expression propagation to overcome limitations such as the one illustrated by Figure 5.35a.

PolyMage [MVB15] and Halide [Mul+16] perform, among other things, a limited form of expression propagation. In contrast to our technique, both will require that there is a *single* user of a temporary location and that the loop surrounding the user is fused with the loop surrounding the definition. Additionally, PolyMage requires both access functions to be equal. If it is possible to generate a schedule to equalize the access functions and to fuse the loops containing the value producer and consumer, expression propagation becomes a redundant load optimization limited to a single loop iteration. If the loops are fused but the access functions are not equalized, there are alternative techniques that perform redundant load elimination across different iterations of a single loop [DGS93; Rag+13]. The Julia language [Bez+12] even offers syntax to force loop fusion, but it relies on the underlying compiler to eliminate redundant loads afterwards.

Compared to existing techniques that propagate scalars or array definitions, polyhedral expression propagation is more powerful and provides a set of dedicated heuristics (ref. Section 5.3.5.1) as well as code generation improvements (ref. Section 5.3.6). Due to the lack of precise dependence information, many techniques are limited to programs in DSA form [Heg+14; Mul+16; Rag+13]. Others restrict propagation expressions to special cases such as constants [RHR05; SK98; Won99], single array reads [VCC03; Won00], or pure scalar definitions that do not read from memory [KG18]. In the case of constants or array reads in DSA form program [Heg+14; Mul+16; Rag+13; VCC03], propagation dependences are trivially equal to the *RAW* dependences. For scalar definitions [KG18], propagation dependences are identical to def-use chains. As an alternative to prior verification, Wonnacott [Won00] proposed to perform propagations and to verify them afterwards by comparing the initial and resulting flow dependences.

Chapter 6

Ongoing & Future Work

> *Even though the future seems far away,*
> *it is actually beginning right now.*
>
> Mattie Stepanek

In this chapter we provide brief descriptions of our ongoing research after we present interesting opportunities for future work. The former were not included in the other chapters because these approaches and results are preliminary. This is also the reason why we omit detailed evaluations and elaborate discussions of related work. Instead, we focus on general ideas and describe the current state of our prototype implementations. We believe this to be worthwhile as our ongoing work nicely augments the theme of the thesis, namely *automatic*, *applicable*, and *sound* polyhedral optimizations for real world programs.

While our ongoing research is actually tightly connected, we present it as four separate ideas. The first one is the inter-procedural static control part (SCoP) representation described in Section 6.1. In this section we briefly discuss the problems that can accompany function calls and how flow-sensitive access summaries can be built to improve the approximation scheme presented in Section 3.4.3. Afterwards, Section 6.2 discusses a novel polyhedral-model-driven inlining scheme. It is required to reuse the our intra-procedural loop optimization frameworks to perform transformations on loops that are spread out over multiple functions. In contrast to classical inlining heuristics, our approach will inline a function only if the polyhedral optimization potential (ref. Section 3.1.5) of the surrounding SCoPs is thereby increased. In Section 6.3, we introduce polyhedral program slicing, a generalization of the runtime check hoisting described in Section 3.7. In contrast to the latter, polyhedral program slicing is more liberal when it comes to the identification of error states. This allows us to build a (partial) polyhedral representation when non-affine loops, unknown function calls, or complexity issues would have otherwise prevented it. Our last ongoing effort is the polyhedral value analysis presented in Section 6.4. The idea behind this analysis is to apply techniques used to built the polyhedral program representation in a generalized way to low-level programs. There are several advantage over the polyhedral modeling used today, including: the support for complex loop iterators and control flow, demand driven analysis of interesting code parts, and variable scoping in combination with an optimistic representation that eases bilateral connection with non-polyhedral analyses. At the core of the polyhedral value analysis are closed form expressions for scalar variable which, under some condition, have a piecewise defined, affine evolution in a loop.

Before we discuss ongoing work, we want to elaborate on other interesting research opportunities in the context of *automatic*, *applicable*, and *sound* polyhedral optimization.

Representation and Optimization Granularity

The granularity of a polyhedral program representation is determined by the statement granularity as well as the statement instance that are precisely represented. A fine-grained representation generally allows for more transformations while a coarse-grained model will decrease compile time. So far there is little research on the potential choices and their respective benefits. Instead, approaches usually determine the granularity out of convenience [Bon+08; Fea92b; GGL12] or as a means to an end [MY15; MDH16; Sto+14]. We believe that statements should be built based on the properties of the referenced accesses. Similarly, a polyhedral representation might not need to distinguish the iterations of all loops and schedule optimization could be applied to parts of the model at a time.

Target and Input Specific Optimizations

Throughout this thesis we used statically taken and dynamically verified assumptions to ensure applicability and soundness (ref. Section 3.5). However, the diversity of modern architectures as well as the varying input sizes in real world applications provide perfect opportunities for further program versioning and specialization. One of the main disadvantages of code versioning, namely the binary size increase, is nowadays often less problematic. As a consequence it is time to investigate how aggressive specialization can be used to improve performance based on dynamic properties like the available hardware, the input sizes, or the current workload of the system [BB14].

Cooperation of Polyhedral and Non-Polyhedral Analyses

Polyhedral tools were historically applied as pre-optimizations prior to the actual compilation. As such they not only had limited access to existing program analyses but also were limited in the information they could pass on to later transformations. GRAPHITE in GCC and POLLY in LLVM are two examples of polyhedral optimizers that already employ non-polyhedral analyses. The latter even annotates the optimized code to improve the results of following passes. However, neither provides a clean interface that would enable non-polyhedral analyses and transformations to take advantage of the precise, high-level information available. Similarly, non-polyhedral optimizations often employ highly tunes heuristics that could augment the objective function used for schedule optimizations.

6.1 Inter-Procedural SCoP Representation

The presence of function calls is one of the most severe limitation for polyhedral-model-based approaches (ref. Section 3.1.2). If the side-effects of a function call are unknown, it acts as an optimization barriers. Thus, there is not much to gain through approximations (ref. Section 3.4.3) because the induced (conservative) dependences will prevent transformations to move any side-effect that was initially executed prior to the call after it, and vise versa. Even if the side-effects of a function call are known, or can be tightly approximated, control flow conditions and access relations have to be considered dynamic (or non-affine) if they reference it. The third downside to algorithms that are distributed over multiple functions is the separation of loops. Classically, loop optimizations are applied intra-procedurally to a sequence of loop nests. However, if the loops are spread out over multiple functions, one cannot simply perform the same transformations without solving additional problems, e.g., the effect on outside callers. The inter-procedural static control part (SCoP) representation we briefly describe in the following deals only with the first of these three problems. Thus, a tight representation of side effects for calls to known (partially) polyhedral functions. The second problem is dealt with by the polyhedral value analysis described in Section 6.4 and a solution to the transformation difficulties is presented in Section 6.2.

```
void foo(float *A, float *B, int N) {
  if (!update)
    return;
  for (int i = 0; i < N; i++)
    if (reverse)
      B[i] += A[i];
    else
      A[i] += B[i];
}
```

Figure 6.1: Example function with an early exit and a path that reads both, but writes only one of the arrays passed as arguments.

Function call approximations, as presented in Section 3.4.3, summarize side-effects in terms of reads and writes to entire arrays. While this is acceptable for certain situations, it lacks range information and it is not well suited if accesses are only performed conditionally. To enable transformations for real world applications, we often require *flow-sensitive access summaries*. As an example consider the code shown in Figure 6.1. Depending on the value of the global variables `update` and `reverse`, either no side-effects occur, or alternatively on of the arrays is written and both are read. Though, in the latter case, only elements between `&A[0]/&B[0]` and `&A[N-1]/&B[N-1]` are accessed. To exploit these facts at the call sites of the function `foo`, we build a polyhedral representation for the entire function. In contrast to the default mode of LLVM/POLLY, this function SCoP representation is not meant to be used

> In this section we discuss the problems that come with function calls and presents our initial prototype for an inter-procedural static control part (SCoP) representation. This is a joint effort with Sebastian Hack, Tobias Grosser, and Torsten Hoefler. The current implementation is available in the `interprocedural` branch of our research prototype.

for schedule optimizations but instead as an intermediate step for a flow-sensitive access summary. Note that our prototype implementation can only build access summaries if the whole function is represented by a single SCoP. Consequently, we enable all applicability extensions of LLVM/Polly and disable profitability restrictions that would otherwise be enforced (ref. Section 3.2.2). Especially for real world applications, a function wide polyhedral representation might not be possible. In this case, we can employ polyhedral program slicing, a generalization of the runtime check removal (ref. Section 3.7), which is detailed in Section 6.3.

Even if we can construct a function SCoP, the precision additionally depends on the call site. To illustrate the problem we provide a minimal example function, inc in Figure 6.2, for which a precise polyhedral representation can be built. However, the memory accesses which are affine in the context of inc, might not be in a larger scope. Only if the values passed as LB and UB are affine expressions in the SCoP surrounding the call site, the access summary of inc can be precisely modeled in the context of the caller. In fact, all parts of the polyhedral function representation might become non-affine or invalid in a larger scope. This especially includes access relations and iteration domains, but also the assumptions that were taken during the modeling (ref. Section 3.5). To determine the representation at the call site, we have to instantiate the function parameters that occur (as part of) SCoP parameters in the callee representation with the values passed as arguments at the call site. In our prototype, an access summary is only build if all instantiations result in affine expressions. We additionally need to deal with other issues, e.g., the invariance of loads that are represented as parameters (ref. Section 3.6).

```
void inc(int LB, int UB) {
  for (int i = LB; i < UB; i++)
    A[i]++;
}
```

Figure 6.2: Contrived example function with a memory access that is affine in the scope of the function but might not have an affine access relation in the context of the caller.

To derive the actual read and write effects of a call, we compose [Ver16, Chapter 4.3.4] the access relation ranges of the memory accesses in the callee with a call site translation function f_c. This step will instantiate the parameters and additionally transition the access relations to the context of the statement S that contains the call. To achieve the second part, we set the domain of f_c to the iteration domain $\mathcal{D}_S$ of S and the range to the parameter space ρ of the callee. Thus, the call transition function maps statement instances of S to the parameters of the callee, or $f_c \subset \mathcal{D}_{\mathtt{S}} \to \rho$. For the first part, we equate each parameter of the callee with its polyhedral value in the context of the caller based on the argument-parameter relation at the call site. For the example call in Figure 6.3, this would result in the call transition function shown in Formula 1.

```
   for (int j = 0; j < N; j++)
S:   inc(j/2, j); // Figure 6.2
```

Figure 6.3: Quasi-affine call site for the code illustrated in Figure 6.2.

$$f_c \coloneqq \left\{ j \to (\mathrm{LB}, \mathrm{UB}) \mid 0 \le j < N \ \wedge\ \mathrm{LB} = \lfloor j/2 \rfloor \ \wedge\ \mathrm{UB} = j \right\} \subset \mathcal{D}_{\mathtt{S}} \to \rho \qquad \textbf{(1)}$$

The relation into the memory space M is achieved by reinterpreting the access relation range $rng(f_m)$ of each memory access m in the callee. Since the range is a parametric subset of M, it is also a relation from the parameter space ρ to the accessed memory locations. For the memory accesss in our example `inc`, this interpretation is denoted as f_M and illustrated in Formula 2.

$$f_M \coloneqq \left\{ (\mathrm{LB}, \mathrm{UB}) \ \rightarrow \ \mathtt{A}(\mathbf{o}) \ \mid \ \mathrm{LB} \leq o < \mathrm{UB} \right\} \subset \rho \rightarrow M \tag{2}$$

In the final step, the memory space relation f_M is applied to the range of the call transition function f_c. The result maps instances of the call to the memory locations accessed in that invocation. For our example this final access relation is shown in Formula 3. Since the read and write access in `inc` have equal access relations, it identifies the memory cells of the array `A` that are both read and written. Since the call transition relation contained equations for all callee parameters, here `LB` and `UB`, they can be eliminated from the result. This simplification, show in Formula 4, ensures that the access relations for the call only depend on parameters of the caller, here `N`.

$$\begin{aligned} f_M \circ_{rng} f_c &\coloneqq \left\{ j \ \rightarrow \ l \ \mid \ (j, p) \in f_c \ \wedge \ (p, l) \in f_M \right\} \\ &= \left\{ j \ \rightarrow \ \mathtt{A}(\mathbf{o}) \ \mid \ 0 \leq j < N \ \wedge \ \mathrm{LB} = \lfloor j/2 \rfloor \ \wedge \ \mathrm{UB} = j \ \wedge \ \mathrm{LB} \leq o < \mathrm{UB} \right\} \qquad (3) \\ &= \left\{ j \ \rightarrow \ \mathtt{A}(\mathbf{o}) \ \mid \ 0 \leq j < N \ \wedge \ \lfloor j/2 \rfloor \leq o < j \right\} \subset \mathcal{D}_{\mathrm{S}} \rightarrow M \qquad (4) \end{aligned}$$

Note that our implementation builds the polyhedral representations in a bottom-up traversal of the call graph. While additional context-sensitivity could further improve the results, e.g., when a constant passed at a call site would allow for an affine representation in the callee, it would also increase the already substantial compile time required to build the polyhedral representations.

6.2 Polyhedral-Model-Driven Inlining

At its core, polyhedral-model-driven inlining is simply an inline heuristic. However, in contrast to commonly used schemes that consider various features including code size, the number of calls sites, and branch probabilities, our scheme is selective to the polyhedral optimization potential. The heuristic should guarantee two important properties: First, inlining a call site will not invalidate the surrounding static control part (SCoP)[1]. Second, inlining increases the profitability of the surrounding SCoP such that more non-trivial transformations are potentially possible. Since existing analysis and transformation passes in LLVM, e.g., the ones used by and implemented in LLVM/Polly, are intra-procedural, an inlining approach is required. While more general inter-procedural SCoP representation, e.g., one that represents the callee SCoP precisely and not only as a summary, could enhance the inlining choices, it cannot easily substitute inlining itself.

```
void pos(int N) {
  for (int i = 0; i < N; i++)
    S(i);
}
void neg(int N) {
  for (int j = N; j < 0; j++)
    P(j);
}
void non_profit(int N) {
  pos(N);
  neg(N);
}
void profit(int N) {
  non_profit(N);
  non_profit(-N);
}
```

Figure 6.4: Multiple functions that only allow for non-trivial loop transformations if all are inlined into the `profit` function.

We first want to demonstrate how the polyhedral-model-driven inlining approach can selectively inline functions to enable polyhedral optimizations. The simple example in Figure 6.4 features the two statements `S` and `P`. Each is surrounded by an affine loop and contained in its own function. As indicated by theirs names, the `pos` function will only execute the statement `S` for positive values of the argument `N` while the `neg` function will do the same for `P` if `N` is negative. In addition, there are two loop-less functions, `non_profit` and `profit`, that either directly or indirectly call `pos` and `neg`. Given this situation, we are not able to perform any interesting loop transformations as there is no intra-procedural static control part (SCoP) that contains at least two loops. Even if we inline the calls in the `non_profit` function, the result would be a SCoP with two statements of which at most one would be executed for any given value of `N`. However, if we inline the `non_profit` calls in the `profit` function and afterwards also the `pos` and `neg` calls that thereby emerge in `profit`, we created an intra-procedural SCoP with four statements of which always two are executed for non-zero values of `N`. Polyhedral-model-driven inlining will therefore only perform the last inlining sequence, consequently leaving the `non_profit` function untouched.

In this section we describe when function call inlining is beneficial and when not. This is a joint effort with Sebastian Hack, Tobias Grosser, and Torsten Hoefler. The current implementation is available in the `interprocedural` branch of our research prototype.

[1] Due to implementation artifacts this is currently not guaranteed by our prototype.

To define the actual inline heuristic with the properties described above, we rely on the inter-procedural SCoP representation (ref. Section 6.1) and the profitability function Ψ that we introduced in Section 3.1.5. While the initial use case for this function was to generate a boolean predicate, we now employ it to determine if inlining will increase the potential for non-trivial loop transformations. To this end, we iterate over the call sites in a valid but potentially unprofitable SCoP, take the profitability functions of the callee SCoPs, and combine them with the initial profitability function of the caller. Afterwards, the result is checked for profitability as described already in Section 3.1.5. Thus, we determine for which parameter valuations a predetermined minimal number of affine loops, represented in our models, will be executed. If the threshold is never reached, the caller SCoP is deemed unprofitable. However, if parameter valuations meet the requirements for profitability, we have to identify the call sites that contributed affine loops and inline them to expose these loops to our intra-procedural transformations.

In the following we denote the caller SCoP as $\mathtt{S}$, the set of contained call sites as Calls, a specific call site as $\mathtt{cs}$, the iteration domain of the statement containing the call as $\mathcal{D}_{\mathtt{cs}}$, the callee SCoP as $\mathtt{S}_{\mathtt{cs}}$, and the profitability function of a callee as $\Psi^{\mathtt{cs}}$. We also make use of the call transition function $f_{\mathtt{cs}}$ which relates callee parameters to the caller context. This transition function and its application is described in Section 6.1

```
procedure RecursiveProfitabilityFunctionGeneration(S : SCoP)
  Get the initial profitability function that does not take the call site potential into account.
  $\Psi_{S} \leftarrow$ ProfitabilityFunctionGeneration(S, false)    see Algorithm 3.11 on Page 34
  Then iterate over the call sites directly contained in the SCoP S and combine theirs profitability
  function with the one for S. To this end, we first have to transition the callee profitability to the
  caller context as described in Section 6.1. The final call profitability relation $\Psi_{S}^{cs}$, is then defined
  as the callee profitability function, transitioned to the caller context, specialized to the iteration
  domain of the call, and projected onto the parameter space $\rho$ (to eliminate loop constraints).
  for cs in Calls do
    $\Psi^{cs} \leftarrow$ RecursiveProfitabilityFunctionGeneration($S_{cs}$)
    $\Psi_{S}^{cs} \leftarrow \pi_{\rho}\left(\mathcal{D}_{cs} \cap \left(\Psi^{cs} \circ_{rng} f_{cs}\right)\right)$
    $\Psi_{S} \leftarrow \lambda c : \left(\Psi_{S}\, c\right) + \left(\Psi_{S}^{cs}\, c\right)$
  end for
  Return the final profitability relation that would hold if all call sites were transitively inlined.
  return $\Psi_{S}$
```

Algorithm 6.5: Algorithm to determine the profitability function of a SCoP under the assumption all call sites, including transitive ones, have been inlined. The resulting piecewise affine function $\Psi_{\mathtt{S}}$ maps parameter valuations to the number of represented and potentially transformable loops that are executed for the given input.

To make an educated inlining decision we first construct the profitability function $\Psi_{\mathtt{S}}$ of the caller under the assumption that all call sites, also transitive ones in the callees, were inlined. To this end, we apply the recursive profitability function generation presented in Algorithm 6.5 to the currently analyzed SCoP $\mathtt{S}$. In this step, the initial profitability function of $\mathtt{S}$ (ref. Section 3.1.5) is combined it with the recursively computed profitability information of the SCoPs that are directly called by $\mathtt{S}$. Note that this process, similar to the inter-procedural SCoP representation, does not allow (transitively) recursive functions, thus strongly connected components in the call graph.

From the function $\Psi_{\mathtt{S}}$, which describes the profitability after aggressive transitive inlining, we determine all parameter conditions under which non-trivial loop transformations can be applied. As already in Section 3.1.5, we define non-trivial as involving at least two affine loops. Thus, we are interested in the parameter context P for which $\Psi_{\mathtt{S}}$ evaluates to a value greater than one.

$$P \coloneqq \left\{ c \mid (\Psi_{\mathtt{S}}\, c) > 1 \right\} \tag{5}$$

While the context P, as defined in Formula 5, allows to determine when non-trivial transformations are possible, it is not sufficient to identify call sites that need to be inlined. We additionally require the call profitability relation $\Psi_{\mathtt{S}}^{\mathtt{cs}}$ as define in line 5 of Algorithm 6.5. Only if this call site specific profitability relation is non-zero for a parameter valuation that is also considered profitable, it is beneficial to inline the call. If not, the call is keep and represented by an access summary as described in Section 6.1. Thus, a direct call site $\mathtt{cs}$ in the analyzed ScoP $\mathtt{S}$ is inlined only if the set defined in Formula 6 is not empty.

$$\Psi_{\mathtt{S}}^{\mathtt{cs}}(P) \cap \left\{ c \mid c > 0 \right\} \subset \mathbb{N} \tag{6}$$

As shown in the example in Figure 6.4, the affine loops we are looking for might be contained in functions further down the call chain. We consequently need to apply the above reasoning recursively for all call sites contained in functions we decided to inline.

The final caveat is the order in which the SCoPs are traversed and also the inlining is performed. The two obvious choices, thus top-down or bottom-up traversal of the call graph, are both suboptimal: An inlining decision that improved the profitability of a function does not have to improve the profitability of the callers. Thus, if we traverse the call graph in a bottom-up fashion, early inlining decisions will be propagated to callers, regardless of the benefit to them. However, if we traverse it top-down, we potentially analyze and, if appropriate, inline every single call site multiple times. As we currently do not have access to enough empirical data to make an educated decision, we leave the traversal and inlining order unspecified for now.

6.3 Polyhedral Program Slicing

Programs are often comprised of various path, some of which are more, others less amenable to polyhedral optimization. While the use of approximations for function calls, non-affine control flow conditions and memory access expressions (ref. Section 3.4) often allows for a polyhedral representation to be built, the usefulness can vary gravely. Approximations often increase the complexity of, and the uncertainty inherent in, the polyhedral representation. Consequently, the compile time required to build, analyze and optimize the program is increased. Since approximations can easily introduce optimization barriers, they also might not allow more program transformations. Finally, applicability improvements through aggressive approximations can, in addition to compile time, also increase the number of required assumptions, thus cause more restrictions on the set of valid input parameter valuations. Consequently, there is a higher chance of statically infeasible assumptions (ref. Section 3.5.5.3) that prevent optimizations all together as well as misspeculations only detectable at runtime (ref. Section 3.5.5.1).

Since the goal is most often the optimization of a program and not a complete polyhedral representation, we can focus on program parts that are, to some degree, amenable to polyhedral modeling. While the simplest way to do so is to disable or limit approximations, it will cause us to miss out on optimizeable program paths that are entangled with non-optimizeable ones. Alternatively, we propose to slice the program into a part that is represented and optimized in the polyhedral model and a part that is not. This can be beneficial if the programs parts not suited to polyhedral representation are unlikely to be executed. In this context, unsuitable parts might include optimization barriers, e.g., approximations of unknown function calls or non-affine loops, as well as iteration domains or access relations which are too complex for precise analysis and representation.

As an example consider the control flow graph (CFG) of the `muxha` function from the SPEC2006 462.libquantum benchmark that is shown in Figure 6.6 (Doerfert and Hack [DH17a] presented a similar example). Note that the presented state is a result of prior polyhedral inlining (ref. Section 6.2) and makes use of inter-procedural SCoP representation (ref. Section 6.1). As shown in the legend, we use different shapes and colored patterns to indicate different kinds of basic blocks which are generally equal to our polyhedral statements. Green blocks with a horizontal stripes identify basic blocks that contain memory writes. Gray blocks without pattern are those that do not. Control flow is determined by the nodes with octagon shape and all nodes with a double

> Polyhedral program slicing is an ongoing research project in cooperation with Sebastian Hack, Tobias Grosser, and Torsten Hoefler. It is an extension of the runtime check elimination we described in Section 3.7. The implementation of polyhedral program slicing is available in the `interprocedural` branch.

border contain function calls. Note that we used the inter-procedural SCoP analysis presented in Section 6.1 to allow calls to functions with a partial polyhedral representation. Thus, a non-recursive call to the shown `muxha` function would also be representable. We indicate loops with a gray background and non-affine control regions (ref. Section 3.4.2) with dashed lines. Finally, red nodes with diagonal stripes are so called *error blocks* for which we cannot built a precise polyhedral representation. In this CFG, error blocks are caused by function calls to unknown functions or transitively, if all predecessors, or successors, are already considered error blocks.

For the `muxha` function, as well as various otherss in the 462.libquantum benchmark [DH17a], we can create a precise polyhedral representation of all loops, However, there is no SESE region which contains multiple loops but no error blocks, i.e., optimization barriers induced by unknown function calls. The profitability relation of this function, as described in Section 3.1.5, is piece-wise defined and consists of at least five pieces. Each represents a different path that can be taken from the first control flow inducing block that directly succeeds the function entry. Since the left two paths do not contain (affine) loops, they are deemed unprofitable. However, the others contain up to three affine loops and are therefore worth to consider. Note that this scheme only works if the control flow conditions that precede error blocks are statically affine. If not, the whole non-affine control region that is built around them has to be considered an error block as well. This will again require a statically affine control flow condition further up the CFG. If none is found, there is no statically known parameter context which would guarantee that only represented, thus non-error blocks, are executed. In contrast to the runtime check hoisting, we propagate error block information already during the SCoP detection and not only in the modeling step (ref. Figure 2.3). This allows us to identify SCoPs early that would execute an error block on every path. While runtime check hoisting generates statically infeasible assumptions for these situations, our program slicing scheme will discard the analyzed region and continue with smaller subregions instead.

For the functions in the 462.libquantum benchmark we can find a context that ensures no error block will be reached. It constraints the values of assumed invariant loads (ref. Section 3.6) that occur both in the optimized function and also transitively in the represented calls. This context is then combined with the other assumptions that were necessary to model the SCoP and used to version the code (ref. Section 3.5).

The schedule optimization algorithm employed by LLVM/POLLY lacks a loop fusion heuristic and instead distributes loops by default. However, there are various methods that improve this scheme [Bon+10; MVB15]. For our experiment we simply enforced loop fusion through a command line option. We achieved a speedup of 1.96× for 462.libquantum on the reference input and an Intel Core i7-4800MQ CPU. The described technique was applied to five SCoPs in the `test_sum` function as well as the function SCoPs (ref. Section 6.1) we built for `muxfa`, `muxfa_inv`, `muxha`, `muxha_inv`, `madd_inv`, and `addn_inv`.

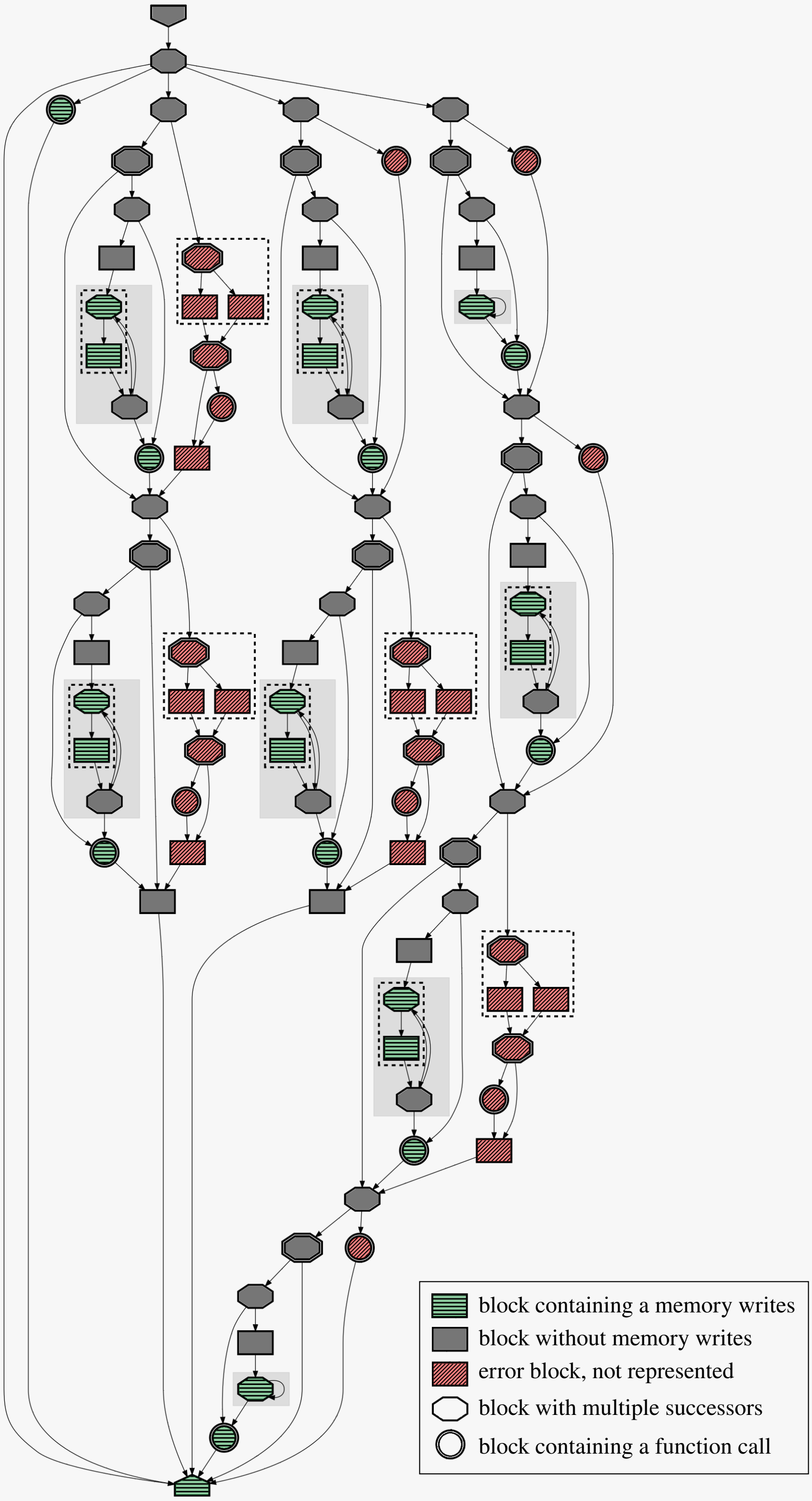

block containing a memory writes
block without memory writes
error block, not represented
block with multiple successors
block containing a function call

Figure 6.6: The control flow graph (CFG) of the `muxha` function, part of the SPEC2006 462.libquantum benchmark, after polyhedral inlining (ref. Section 6.2) was performed. All nodes represent basic blocks. The red nodes with a diagonal pattern are considered error blocks and consequently not part of the SCoP. The green nodes with horizontal stripes contain memory write accesses while the gray nodes without pattern do not. Nodes with a double border contain function calls (ref. Section 6.1). Loops are indicated by a gray background and non-affine subregions (ref. Section 3.4.2) are marked with dashed lines.

6.4 Polyhedral Value Analysis

The polyhedral value analysis (PVA) [DH17a; DH17b] is our latest endeavor to improve the applicability and robustness of polyhedral techniques on low-level code. It is intended to fulfill two distinct purposes. First, as a replacement of the SCALAR EVOLUTION analysis [BWZ94; PCS05] in polyhedral tools like LLVM/POLLY and GCC/GRAPHITE. As such it especially allows us to derive polyhedral representations for piecewise defined, quasi-affine expressions. Second, it is designed to integrate well with non-polyhedral analyses and transformations. Due to various design choices explained in the following, it can be easily used to augment information derived from SCALAR EVOLUTION or other sources. The goal is to finally provide polyhedral information to the non-polyhedral parts of an industry strength compilation pipeline.

In a nutshell, PVA performs two entangled tasks which are in LLVM/POLLY divided into multiple steps and partially outsourced to SCALAR EVOLUTION. The first, and arguably main function of the PVA is to derive a polyhedral representation for the value of a LLVM-IR instruction. This step requires also the second task which is the generation of iteration domains for basic blocks. We require these iteration domains to generate precise piecewise expressions, e.g., for phi nodes which have different values depending on the control flow edge that was traversed to reach them. Since we require the polyhedral representation of control flow conditions to build iteration domains (ref. Section 3.3), the two functionalities of the PVA depend on each other.

As LLVM/POLLY also has to perform the tasks implemented in the PVA , we used existing code and concepts as a starting point. We especially rely on the generic iteration domain generation that we added to LLVM/POLLY and described in Section 3.3. In addition, we inherit the concept of an assumed context for which the representation is actually valid (ref. Section 3.5), as well as the techniques to ensure a correct representation of low-level operations presented in Section 4.2 and 4.3. However, in contrast to LLVM/POLLY, all parts of the PVA are demand driven and optimistic, meaning that a non-affine or dynamic operand of an expression e will not cause us to give up, but instead it will become a parameter in the polyhedral representation of e. This is desirable as PVA users might be able to substitute these parameters with a value (range) derived from other sources (ref. Section 3.4.1) or even prove their invariance (ref. Section 3.6). The PVA can itself eliminate such parameters as described during in the discussion of Figure 6.9.

> The polyhedral value analysis [DH17a; DH17b] is a polyhedral-model-backed alternative to the SCALAR EVOLUTION analysis [BWZ94; PCS05] available in GCC and LLVM. The development is a cooperation with Tina Jung, Alexander Matz, and Sebastian Hack. Conceptually, it is similar to the inter-procedural polyhedral analysis developed in the bachelor thesis of Jung [Jun15]. The implementation is available in the pva branch of our LLVM fork: `https://github.com/jdoerfert/llvm`.

In the following we will focus on the polyhedral representation of LLVM-IR instructions as iteration domain generation is similar and the general concepts were already shown in Section 3.3.

In addition to an instruction `I`, the PVA will require the user to specific the *scope* and *use location*, both in terms of a loop. The scope s determines up to which point the PVA will try to build a polyhedral representation. All (transitive) operands outside the scope become parameters. The use location specifies the program point at which the value of `I` is sought-after. Thus, the PVA will try to create a piecewise quasi-affine expression $e_{\mathtt{I}}$ which describes the value of `I` at the use location l with regards to iterations of loops surrounding l and values defined outside of s. Note that a scope can also be the entire function but it always has to contain the use location. If it is a function, the parameters of $e_{\mathtt{I}}$ will only be arguments and non-affine or dynamic instructions.

```
i = 5;
do {
  j = i;
  do {
    x = 3*j + i;
    S(x);
  } while (j++ < 2*i);
  P(x);
} while (i++ < N);
Q(x);
```

Figure 6.7: Example to show how scope and use location change the derived polyhedral representation.

To illustrate how the scope and use location change the result we provide a simple example in Figure 6.7. The scalar variable `x` is defined as an affine expression of the surrounding loop iterations variables i and j. It is then used three times, once in each loop nesting depth. The different polyhedral representations computable by the PVA are shown in Table 6.8. If we for example take the i-loop as a scope, thus represent the value in one fixed iteration of this loop, we can compute two different integer relations that represent the value of `x`. The first, $\{ (j) \to (x) \mid x = 4\mathtt{i} + 3j \}$, is computed for the use location `S` and depends on the loop iteration of the j-loop. Note that we use different fonts to indicate that `i` is here an existentially qualified parameter which represents the value of the variable `i` and not the iteration of the outer loop which is denoted as i and starts at zero not five. Since `P`, the second possible use location if we keep the i-loop as a scope, is not surrounded by the j-loop, the polyhedral representation of `x` at this point will not depend on its iteration. Instead, we have to determine the value of `x` in the last iteration of the j-loop. To do this we first determine the relation for the same scope and the j-loop use location. To be more precise, we are interested in the use location for the innermost loop that surrounds the definition of the scalar variable, here `x`. In addition we require the iteration domain of the basic block that contains this definition. In the following we denote the former as $e_{\mathtt{x}} = \{ (j) \to (x) \mid x = 4\mathtt{i} + 3j \}$ and the latter as $\mathcal{D}_{\mathtt{x}} = \{ (j) \mid 0 \leq j \leq \mathtt{i} \}$[2]. To derive the value in a different loop, here the i-loop, we compute the lexicographic maximum of the dimensions of $\mathcal{D}_{\mathtt{x}}$ that are not shared by the definition and the use location. If we intersect the relation $e_{\mathtt{x}}$ with the result and project out the now fixed inner dimensions we end up with a representation that describes the value only in

[2] Note that the inner loop iterates `i` + 1 times because `j` is incremented after the exit condition `j<2*i` is evaluated. This also means that `x` is last defined with `j=2*i`, thus in the last iteration `x` is equal to `3*(2*i)+i`.

terms of shared loop iterations. For our example the lexicographic maximum in the j-dimension with regards to the i-loop scope is $\{\, (j) \mid j = \mathtt{i} \,\}$. The intersection with the domain of $e_{\mathtt{x}}$ will result in $\{\, (j) \to (x) \mid j = \mathtt{i} \wedge x = 4\mathtt{i} + 3j \,\}$, and after we eliminate the now fixed input dimension we get $\{\, () \to (x) \mid x = 7\mathtt{i} \,\}$. It is worth to note that the lexicographic maximum can be piecewise defined, especially if the loop has multiple exit edges. This is important as the instantiation will then account for the case that the variable we are interested in was (under some conditions) not defined in the last loop iteration. Similarly, if the loop has multiple back edges, the use of the iteration domain of the containing basic block will ensure that the lexicographic maximum describes the last statement instance for which the variable was actually defined.

scope/use	`S` **(j-loop)**	`P` **(i-loop)**	`Q` **(function)**
j-loop	$\{() \to (x) \mid x = \mathtt{i} + 3\mathtt{j}\}$	n/a	n/a
i-loop	$\{(j) \to (x) \mid x = 4\mathtt{i} + 3j\}$	$\{() \to (x) \mid x = 7\mathtt{i}\}$	n/a
function	$\{(i,j) \to (x) \mid x = 4i + 3j + 20\}$	$\{(i) \to (x) \mid x = 7i + 5\}$	$\{\, () \to (7N - 7) \,\}$

Table 6.8: Different polyhedral representations for the value of `x` in Figure 6.7, depending the loops used as scope (rows) and use location (columns).

To make the PVA easily compatible with non-polyhedral passes we designed the interface after the one provided by Scalar Evolution. This includes functions to compare, modify, and evaluate polyhedral expressions, as well as functionality to derive high-level information, e.g., loop trip counts from the iteration domains of loop headers. To alleviate compile time costs the PVA is, similar to Scalar Evolution, demand driven and caching. The main advantage of the PVA is the ability to represent piecewise defined values. Combined with the natural flow and iteration sensitivity of the polyhedral model, as well as an optimistic representation of values, this enables the PVA to generate partial piecewise representations for scalar variable with (partially) statically quasi-affine definitions. While both analyses incorporate non-representable or dynamic values as "parameters" into their respective representation, only the piecewise definition employed by the PVA allows to eliminate them later on to derive independent information.

The code shown in Figure 6.9 demonstrates how an optimistic, piecewise defined representation can improve analysis results. The loop, which executes until the pointers p and u meet, could be part of the partition routine in a sorting algorithm like quick sort. Since in each iteration either `p` is incremented or `u` is decremented, we should be able to infer that the loop iterates `u - p` times. However, due to the data-dependent conditional, neither Scalar Evolution nor LLVM/Polly are able to make this conclusion. The key advantage of the PVA is the optimistic representation of both pointers at

```
assume p < u;
do {
  v = *p;
  if (v < pivot)
    p++;
  else
    *(u--) = v;
  c = (p == u);
} while (!c);
```

Figure 6.9: Example to show how a dynamic value is first incorporated into and later eliminated from the polyhedral representations.

their use in the definition of `c`. These representations, shown with loop scope in Formula 7 and 8, both reference the data-dependent value `v`. If we interpret the equality in the definition of `c` as a difference between the two operands, thus `(u - p) == 0`, we can eliminate this data-dependent parameter. The loop trip count can then be derived from the result for two consecutive iterations, thus the evolution of the exit condition. However, our implementation is not yet able to perform this last reasoning step.

$$e_{\mathtt{p}} = \{ () \to (x) \mid x = (\mathtt{v} < \text{pivot} \; ? \; \mathtt{p} + 1 \; : \; \mathtt{p}) \} \qquad \textbf{(7)}$$

$$e_{\mathtt{u}} = \{ () \to (x) \mid x = (\mathtt{v} < \text{pivot} \; ? \; \mathtt{u} \; : \; \mathtt{u} - 1) \} \qquad \textbf{(8)}$$

As mentioned earlier, the PVA implementation is based on code for alike functionalities in LLVM/Polly. Especially, the transitioning from most LLVM-IR instructions to polyhedral values is very similar. If an operation is encountered, the representations of the operands are recursively built and afterwards combined to a piecewise defined, quasi-affine relation. While this process is similar in the PVA and LLVM/Polly for all binary operations[3], the PVA has to deal with phi nodes itself whereas LLVM/Polly relies completely on Scalar Evolution to build a closed form additive recurrence. Scalar Evolution can do so for variables with a polynomial evolution in a loop nest, but it is not well suited to handle phi nodes that have a piecewise defined value. This especially includes phi nodes after conditionals, if they are not recognized as minimum or maximum computations, as well as phi nodes in the header blocks of more complicated loops.

```
i = 0;
if (skipfirst)
  i = 1;
for (; i < N; i+=2)
  B(i);
```

Figure 6.10: Example to show how the polyhedral value analysis determines closed form expressions for piecewise defined loop iteration variables.

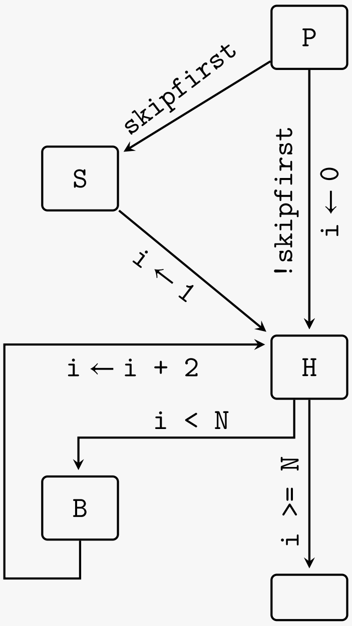
The PVA is generally superior to Scalar Evolution if a loop iteration variable has different constant strides, the corresponding phi has different initial values[4], or the loop contains complicated affine control flow conditions that guard loop back and exiting edges. To explain how the PVA computes closed form expression for loop iteration dependent values, and to showcase one of the advantages over Scalar Evolution, we can consider the code presented in Figure 6.10. To determine the trip count of the loop, the PVA requires the iteration domain $\mathcal{D}_{\mathtt{H}}$ of the header block in a scope that subsumes the loop. Since we reuse the methods described in Section 3.3, we can compute this domain using the conditions under which the loop is entered and the ones under which it is left. Though, while the former do not depend on values changed by the loop in question, the latter most often do.

[3] In the PVA we replicated the detection of various bit-manipulation patterns already available in the Scalar Evolution analysis. These allow us to create the same polyhedral representation for various syntactic forms of an expression. Examples that illustrate such a case are shown in Figure 4.15 on Page 110.

[4] While this is uncommon in the initial SSA-form representation generated by a language front-end, it can easily occur in high-level source code and after the low-level program has been optimized, e.g., through "jump-threading".

```
P: i = 0;
   if (skipfirst)
S:   i = 1;
   do {
H:   c = i < N;
     if (!c) break;
B:     i += 2;
   } while (true);
```

(a) Code shown in Figure 6.10 in a version close to the control flow graph (CFG) representation.



(b) Control flow graph (CFG) for the code shown in Figure 6.11a.

Figure 6.11: The top shows the example from Figure 6.10 in a version that is close to the control flow graph (CFG) representation outlined in the bottom.

To facilitate the following explanation, we provide the example code shown in Figure 6.10 in a lowered form again in Figure 6.11a. This representation is closer to the input of the PVA, thus the control flow graph (CFG) of the program which we sketched in Figure 6.11b. The PVA will create a polyhedral representation of a loop carried, thus recursively defined, value in two steps. First note that we cannot simply recurs on the operands and combine the result as we do with other operations. However, if we restrict the analysis scope to the loop in question, we can determine the evolution of a phi node in one single iteration via recursion on the values assigned on the back edges. For our example, the PVA will compute $\{\,() \to (x) \mid x = (\texttt{i} + 2)\,\}$ as the new value of i. While we are especially interested in the constant part, as it describes the evolution of the variable, we also have to ensure that the factor of the loop carried phi is either minus one, zero, or one. If it is not, the variable evolves polynomially, hence non-affine. If we restrict[5] us for now to a unit factor and a single constant offset c, we can describe the evolution in the loop with the relation $\{\,(i) \to (c * i)\,\}$. Afterwards, we analyze the entering edges to identify the initial values. For our example, the PVA constructs the piecewise defined relation $\{\,() \to (x) \mid x = (\texttt{skipfirst}\ ?\ 1 : 0)\,\}$ based on the definitions on the incoming edges. In the last step we add the two intermediate results to obtain the final representation of a loop carried phi. For the variable i in our example, the result of the PVA is consequently $\{\,(i) \to (2i + x) \mid x = (\texttt{skipfirst}\ ?\ 1 : 0)\,\}$. With this precise representation we can compute the loop exit conditions as explained in Section 3.3. Afterwards, the loop header domain is determined from which we then derive the precise loop iteration count: $\texttt{skipfirst}\ ?\ \lfloor (N + 1)/2 \rfloor : \lfloor N/2 \rfloor$.

While this short introduction missed caveats, details, and features of the polyhedral value analysis, we hope that the descriptions were sufficient to get an impression of capabilities and restrictions.

[5] Our PVA prototype can also deal with alternating evolutions (a factor of minus one) as well as strongly connected phi node chains. The support for multiple constants strides is under development.

Chapter 7

Conclusion

> *There is no real ending.*
> *It's just the place where you stop the story.*
>
> Frank Herbert

The ever-growing complexity of architectures, especially with regards to the memory subsystem as well as the number and diversity of processing units, puts more and more pressure on compilers. The era in which local optimizations applied to the user provided algorithms were sufficient to achieve high utilization of the hardware are almost over. Modern machines come with many, potentially different, cores, accelerators, and flavors of memory. As a consequence, we need to constantly customize our algorithms and data to take full advantage of this diversity. Since manual adaption will become more and more infeasible, compiler optimizations need to advance. However, the required transformations are only possible with a holistic view of the program and the ability for structural but also fine-grained program changes.

The polyhedral model, as a mathematical program representation and transformation framework, is well equipped to deal with many challenges presented by modern algorithms and architectures. It is consequently not surprising that many new domain specific optimizers use polyhedral representations and techniques to generate high-performance code [Bag+15; Bag+18; MVB15; Pra+17; Vas+18]. Since these approaches often come with a domain specific language (DSL) that was designed with the polyhedral model in mind, it can be easily applied. However, it is unrealistic to expect a large portion of the existing software to be rewritten in such new programming languages. There are many reasons for this, most notably correctness concerns and the required effort. Manually changing code is an error-prone task on its own, but rewriting complex software, while keeping high confidence in the functionality, is significantly harder. There are two solutions to this problem. First, one can automate (parts of) the code rewriting process in order to reduce the code base that needs to be trusted effectively to the rewriter tool. Alternatively, one can reimplement (parts of) the domain specific optimizations for existing, low-level languages which would allow optimization through “mere” recompilation. Though, in either case, we need applicable, robust, and correct polyhedral analyses suitable for low-level languages to begin with.

In this thesis we describe several advances towards *automatic*, *applicable*, and *sound* polyhedral representation and optimization of low-level code. Our efforts, which are all publicly available and often already integrated into LLVM/POLLY, make it possible to apply polyhedral analysis and transformations to existing programs without rewriting them. The presented techniques allow us to handle inputs that do not satisfy the strict syntactic (ref. Section 3.3 and Section 6.4) and semantic (ref. Section 3.4, Section 3.6, and Chapter 4) requirements polyhedral-model-based tools often entail. At the core of our correctness assuring and many applicability enhancing techniques is a code versioning framework that is natively embedded into the polyhedral pipeline (ref. Section 3.5). The use of Presburger formulae based assumptions has proven to be well suited to cope with the lack of static information which is often a limiting factor when it comes to a holistic program representation and structural program transformations. While various simplification techniques allow us to keep the runtime verification cost of our assumptions low, we additionally welcome information provided by the program, either statically, through code annotations (ref. Section 5.1.2), or dynamically, via profiling (ref. Section 3.5.2). The evaluations of the proposed enhancements on real world applications clearly show a significant increase across various applicability metrics (ref. Section 3.1). These improvements allow polyhedral optimizers to cope not only with scientific code, that were written by experts to be amenable to high-level optimizations, but also legacy applications. With the BT benchmark from the NAS parallel benchmark suite (ref. Figure 3.37 and Figure 5.28) and the 462.libquantum benchmark from SPEC2006 (ref. Figure 6.6), we showcased two examples that already required almost all presented extensions to be represented and optimized in the polyhedral model. In order to advance the latter, we investigated how reduction dependences can be effectively identified and exploited by polyhedral tools (ref. Section 5.2). Finally, we presented a polyhedral-model-based optimization that alters the time and place intermediate results are computed (ref. Section 5.3). This transformation not only eliminates dependences, which improves the scheduling freedom, but it can also improve hardware utilization and decrease the memory requirements of a program.

We hope that our efforts, both concluded and ongoing, will eventually contribute to the adoption of polyhedral-model-based techniques for a larger variety of code. While we believe that our results already indicate the feasibility of such a development, we acknowledge that we have yet to reach the point where polyhedral techniques will become standard in the analysis and optimization of general purpose applications.

List of Algorithms, Figures, and Tables

Appendices

Appendix A: Dimensionality Adjustment

The dimensionality of iteration domains depends on their loop nesting depth within the SCoP. As an edge from a basic block to a successor block might change the loop nesting depth, the dimensionality of the propagated control flow constraints has to be changed too. The adjustment is performed by the ADJUSTDIMENSIONS procedure shown in Algorithm A.1. It takes domain constraints constructed for the loop nesting depth of the Old basic block and adjusts them to the nesting depth of the New block. If the Last flag is set, the loop dimensions that are left, and consequently projected out of the constraint set, are first specialized to refer the lexicographic last iteration in these dimensions.

```
procedure ADJUSTDIMENSIONS(Dom : ISet, Old : BB, New : BB, Last : bool)
  Get the innermost affine loops that are part of the SCoP to determine the necessary adjustment.
  OldLoop ← getSurroundingLoopInSCoP(Old)
  NewLoop ← getSurroundingLoopInSCoP(New)
  If the immediate surrounding loop did not change no adjustment is necessary.
  if OldLoop == NewLoop then return Dom
  If the loops changed we distinguish three cases based on the difference in the loop depth. Note that
  the CFG edge from Old to New implies that at most one new loop was entered but at the same time
  multiple ones can be left (ref. Theorem A.2).
  DepthDiff ← getDepth(NewLoop) - getDepth(OldLoop)
  if DepthDiff == 0 then                                 one loop was left, one loop was entered
    Dom ← Dom.projectOutDimension(1, Last)
    return Dom.appendDimensions(1)
  else if DepthDiff > 0 then                              no loop was left, one loop was entered
    assume DepthDiff == 1
    return Dom.appendDimensions(1)
  else if DepthDiff < 0 then                        at least one more loop was left than entered
    if not NewLoop or NewLoop.contains(Old) then                                 only left loops
      return Dom.projectOutDimension(-DepthDiff, Last)
    else                                    1+|DepthDiff| loops left and one new loop entered
      Dom ← Dom.projectOutDimension(1 - DepthDiff, Last)
      return Dom.appendDimensions(1)
    end if
  end if
```

Algorithm A.1: Dimensionality adjustment procedure for domain constraints. The given Dom was constructed for the loop nesting depth of Old and will be adjusted to the nesting depth of New. The Last flag indicates if the loop dimensions that were left, and which are consequently projected out of the constraint set, should be specialized to the lexicographic last iteration.

Loop Nesting Depth Adjustment **Theorem A.2**

Given a reducible control flow graph (CFG) [HU74] with nodes **V** and edges **E**.

An edge $\mathbf{e} := (\mathbf{s}, \mathbf{t}) \in \mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ can

1) potentially leave any number of loops that surround **s**, but

2) enter at most one loop which contains **t** but not **s**.

Proof.

Part 1.

To prove that an edge can leave any number of loops we provide a construction for a CFG with k loops in which there is an edge that leaves l loops, for each $0 \leq l \leq k$. The general scheme is presented in Figure A.3. Each of the k nested loops contains a conditional jump to the label R. Note that the node for R has loop depth 0 and no outgoing edges. For each loop depth l the jump to R leaves exactly l loops. At the same time the control flow is reducible since each loop has a unique entry, or header, node [HU74].

Part 2.

Given a reducible CFG, we know that each loop has a distinct and identifying loop header block [HU74]. Entering such a reducible loop can only happen through this header block. Since **t** cannot be the header block of more than one loop, there is at most one loop that could have been entered when traversing **e**, namely the one of which **t** is the header block. ■

The domain adjustment algorithm makes use of the reducibility property we require for all loops in the analyzed region as well as Theorem A.2. While reducibility also guarantees that two loops are either disjoint or properly nested, Theorem A.2 states that any number of loops can be left, but at most one loop can be entered when an edge in a reducible CFG is traversed. Note that we did not investigate the possibility to enforce reducible control flow [HU74] through duplication as it seems of little practical use (see the IR rejection reason in Table 3.19 on Page 44).

```
   for (i1 = 0; i1 < N; i1++) {
    if (p1) goto R;
     [...]
     for (ik = 0; ik < N; ik++)
      if (pk) goto R;
     [...]
   }
R: return;
```

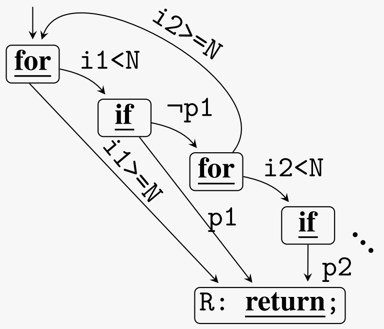


Figure A.3: CFG construction scheme that features edges which leave an arbitrary number of loops. The high-level code representation is sketched on the left and the corresponding CFG is shown on the right.

Appendix B: Glossary

This glossary provides brief definitions of commonly used terms. References to appropriate literature and thesis sections are provided if applicable. In contrast to the rest of the thesis, all occurrences of terms contained in the glossary are highlighted in the definitions that follow.

Access

Accesses are reads or writes of scalar variables (scalar access) or (multidimensional) memory locations (memory access) contained in a statement. In the polyhedral model, communication, and thereby dependences, between any two *different* statement instances is only possible through accesses. Scalar accesses are identified solely by the scalar variable while memory accesses are defined by the accessed array (or pointer) and the access offset which is expressed as an access relation. A memory access is static affine if the accessed array is statically known and the access relation is statically affine.

Access Relation

Each memory access has an *access relation* that maps iterations of the loops surrounding the containing statement to offsets into the accessed array. Thus, a memory access m to an n-dimensional array which is contained in the statement $\mathtt{S}$ with the iteration domain $\mathcal{D}_{\mathtt{S}}$ has an access relation f_m which is a subset of $\mathcal{D}_{\mathtt{S}} \rightarrow \mathbb{Z}^n$.

Affine Expression

An *affine expression* is a linear expression with a constant offset. An affine expression over the variables $x_1, \ldots, x_n$ and with constant integers $(c_0, \ldots, c_n) \in \mathbb{Z}^{n+1}$ can be written as:

$$c_0 + \sum_{1 \leq i \leq n} (c_i * x_i)$$

Note that we regularly use the term affine expression for historical (and brevity) reasons but actually allow piecewise defined, quasi-affine expressions.

Aliasing

Aliasing, or array overlapping, occurs if differently named arrays (or pointers) share the same (physical) memory locations (ref. Section 4.1). Aliasing is especially common in low-level languages such as C/C++ and LLVM Intermediate Representation (LLVM-IR).

Assumption Λ

We use the term *assumption* to denote a context that contains all parameter combinations which we allow to occur during program execution. Assumptions can consequently be interpreted as preconditions specified by the compiler to enable program optimizations (ref. Section 3.5). The complement of an assumption is a restriction.

Basic Block

Basic blocks are the nodes in a control flow graph (CFG). They consist of an ordered list of instructions without intermediate control-flow. Thus, all instructions are executed in their order of appearance, every time the block is executed.

Context

A *context* is an integer set that only refers to program parameters. The "known context" contains parameter constraints derived from the input program, e.g., from type ranges. Assumptions and restrictions are examples for generated contexts (ref. Section 3.5).

Control Flow Condition

A *control flow condition* is the condition of a branch or switch instruction that is evaluated in order to determine which basic block is executed next.

Control Flow Graph (CFG)

A *control flow graph* (CFG) is a program representation often employed in compilers. The nodes of a CFG are basic blocks which are connected through a directed control flow edge if the blocks can potentially be executed directly after another. If a block has multiple successors in the CFG, the dynamic value of the control flow condition will determine which one is executed next.

Dependence

A *dependence* is a connection between two statements, or statement instances, that indicates a potential flow of information, e.g., values. Dependences can be divided into control and data dependences. In the polyhedral model, the former are implicitly captured through the schedule, while the latter are explicitly computed on a statement instance basis [Fea91]. There are three important kinds of data dependences, read-after-write (*RAW*), write-after-read (*WAR*), and write-after-write (*WAW*).

Dominance & Post-Dominance

Dominance and *post-dominance* are strict, partial orders defined over the nodes in a flow graph [LM69; Pro59]. Given a control flow graph (CFG) and two basic blocks A and B, we say that A dominates B if and only if every path from the entry (=source) node of the CFG to B also contains A. Similarly, B post-dominates A if and only if every path from A to an exit (=sink) node of the CFG contains B.

Dynamic (Runtime)

We use *dynamic*, or runtime, to refer to facts that are only known during program execution. In contrast, facts that are known beforehand are statically known.

Dynamic Single Assignment (DSA)

In contrast to static single assignment (SSA), *dynamic single assignment* (DSA) ensures that each memory location is written at most once at runtime [Fea88a; Van+07]. The accesses in a DSA program can consequently only induce read-after-read (*RAR*) and read-after-write (*RAW*) dependences.

Integer Set

We denote symbolic subsets of the n-dimensional integer space $\mathbb{Z}^n$ as *integer sets* if they are defined by a Presburger predicate c with n free variables. An integer vector $\mathbf{x} \in \mathbb{Z}^n$ is included in the set, if only if, $c(\mathbf{x})$ is *true*. We denote integer sets as shown below and describe operations on them in Section 2.1.2.

$$\{\, (x_1, \dots, x_n) \in \mathbb{Z}^n \mid c(x_1, \dots, x_n) \,\} = \{\, (\mathbf{x}) \mid c(\mathbf{x}) \,\} \subseteq \mathbb{Z}^n$$

Integer Relation

We call a symbolic relation between $\mathbb{Z}^n$ and $\mathbb{Z}^m$ an *integer relation* if it is defined by a Presburger predicate c with $n + m$ free variables. As such it can also be seen as integer set in the integer space $\mathbb{Z}^{n+m}$. We write integer sets as shown below and describe operations on them in Section 2.1.2.

$$\{\, (i_1, \dots, i_n) \in \mathbb{Z}^n \rightarrow (j_1, \dots, j_m) \in \mathbb{Z}^m \mid c(i_1, \dots, i_n, j_1, \dots, j_m) \,\} \subseteq \mathbb{Z}^n \rightarrow \mathbb{Z}^m$$

Integer Wrapping

Integer wrapping occurs if the result of an operation, e.g., addition, requires more bits than provided by the target location or variable. A detailed discussion on integer wrapping can be found in Section 4.2.

Iteration Domain

The *iteration domain* $\mathcal{D}_{\mathtt{S}}$ of a statement $\mathtt{S}$ is an integer set that contains all iteration vectors for which $\mathtt{S}$ is executed. Examples for iteration domains are provided in Section 2.1.3 and the iteration domain generation of LLVM/Polly is described in Section 3.3.

Iteration Vector

An *iteration vector* $\mathbf{i}$ is an element of an iteration domain $\mathcal{D}_{\mathtt{S}}$ for a statement $\mathtt{S}$. The dimensionality of $\mathbf{i}$ is equal to the loop depth of $\mathtt{S}$ relative to the static control part (SCoP). Iteration vectors are mostly used to identify loop iterations but they can potentially also constrain the parameters of the SCoP. We write iteration vectors in bold but use italics for the elements, e.g., $\mathbf{i} = (i_1, \dots, i_n)$.

LLVM

LLVM is a compiler framework that combines various language front-ends, a unified middle-end comprising analyses and optimizations, as well as machine-code generation for multiple target architectures [LA04]. See also Section 2.2.1.

LLVM Intermediate Representation (LLVM-IR)

The LLVM intermediate representation (LLVM-IR) is an assembly-like language in static single assignment (SSA) form [Adv+03]. It is used by analyses and optimizations in the middle-end of the LLVM compiler. The language documentation can be found online: `https://llvm.org/docs/LangRef.html`

Live-Out

A variable (or memory location) is considered *live-out* with regards to a region R, if the (stored) value can potentially be used outside of R. In our setting R is most often the extent of the static control part (SCoP). (ref. Section 5.3.6.1)

Memory Access

A *memory access* is a polyhedral access to a memory location.

Parameter

Scalar variables that are invariant during the execution of a static control part (SCoP) are called *parameters*. Note that we consider computations that are pure, and only involve parameters, also as parameters.

Polly

POLLY is the polyhedral loop optimizer integrated into the LLVM compiler [GGL12]. POLLY can automatically detect, model, and optimize all static control parts (SCoPs) present in a function. The input and output of POLLY are functions in the LLVM-IR, the low-level intermediate language of LLVM. The techniques presented in this thesis were all implemented for POLLY. To explicitly refer to the open source version of POLLY, and not our research prototype (ref. Section 2.2.3), we generally write LLVM/POLLY.

Polyhedral Model

The *polyhedral model* is a mathematical program abstraction for static control parts that is based on Presburger arithmetic [FL11]. Its name stems from the fact that conjunctions over affine expressions define polyhedra, thus convex geometrical shapes. However, due to disjunctions and piecewise defined affine expressions we can actually deal with unions of polyhedra. In case quasi-affine expressions are allowed, each polyhedron might additionally contain regularly occurring holes.

Presburger Arithmetic

Presburger arithmetic is a decidable first-order logic over integers with addition first described by Presburger [Pre31] [1]. It is the mathematical foundation the polyhedral model and briefly explained in Section 2.1.2.

[1] Note that an annotated English translation of the originally German paper was made available by Stansifer [Sta84].

Presburger Condition, Predicate, and Formula

A *Presburger condition*, *predicate*, or *formula* is an expression in Presburger arithmetic which, given an assignment for the free variables, can be proven *true* or *false*. We generally assume Presburger conditions to be logical combinations of equalities and inequalities over piecewise defined, quasi-affine expressions as described in Section 2.1.2.

Quasi-Affine Expression

A *quasi-affine expression* is an affine expression that additionally allows division and modulo operations with a constant divisor [Fea91; Qui84]. The general structure of quasi-affine expression is explained in Section 2.1.2.

Referential Transparency

We call the definition of a variable *referentially transparent* if the program behavior is not changed if all variable uses, that would otherwise evaluate to the result of the defining expression, are syntactically replaced by the defining expression.

Restriction

Complementary to assumptions, *restrictions* express constraints on program parameters which are assumed to never occur during program execution. The differentiation can be beneficial due to the representation of integer sets (ref. Section 3.5.3).

Scalar Access

A *scalar access* is an access that does not involve memory but a scalar variable.

Scalar Variable

Unlike arrays, *scalar variables* are containers for a single value of their respective type.

Schedule

The *schedule* θ is an integer relation that maps statement instances to multidimensional timestamps which define their execution order. A brief introduction of schedules is provided in Section 2.1.3.

Schedule Optimization

Schedule optimizations are transformations of the program that involve a change in the execution order of the statement instances, thus their schedule. Polyhedral optimizers classically use integer linear programming to compute a new schedule relation with properties superior to the original [Bon+08; Fea92a; Fea92b; Kon+13].

Single-Entry Single-Exit (SESE) Region

A *single-entry single-exit* (SESE) region is a connected sub graph of a CFG defined by two distinct basic blocks: The entry, that dominates all blocks in the SESE region, and the exit, which post-dominates all of them.

Statement

We mostly use the term *statement* to refer to polyhedral statements which are part of the polyhedral representation of a static control part (SCoP) and identified by their name. Statements comprise an iteration domain and a list of accesses. Alternatively, statements can also denote source code statements.

Statement Granularity

Statement granularity describes the extent of polyhedral statements. Common choices are usually syntactically defined, e.g., source code statements [BRS10; Bon+08; Fea92b; GL96; Ver15b] or basic blocks in a control flow graph (CFG) [GGL12; Pop+06]. However, semantic choices, e.g., one memory write access per statement or alternatively larger compound statements, are interesting to simplify algorithms [Doe+15], reduce compile time [MY15], extend applicability [MDH16], or increase the scheduling freedom [Sto+14].

Statement Instance

A *statement instance* is a dynamic execution of a statement. It is described by the statement (name) and an iteration vector that identifies iterations of the surrounding loops.

Static (Compile Time)

We use the term *static* to distinguish the compile time of a program from the dynamic invocation at runtime. We also use it to denote the fact that variables are unknown but fixed during one invocation, e.g., of a static control part (SCoP). Since the polyhedral model is a static program abstraction we allow pointers and parameters with unknown but invariant values. The uncertainty that is thereby induced makes it harder to reason about the semantics, e.g., potential aliasing of arrays (ref. Section 4.1), as well as the benefit of transformations, e.g., parallelization.

Static Control Part (SCoP)

A *static control part* is a maximal program part that can be represented in the polyhedral model [CGT04; Fea91]. In this thesis we also use the term SCoP to denote to the polyhedral representation that is built for such a program part. SCoPs consist of a list of statements and a schedule as explained in Section 2.1.3. Common restrictions on SCoPs are detailed in Section 3.2.

Static Single Assignment (SSA)

Programs in *static single assignment* (SSA) form come with two important properties: A scalar variable is defined exactly once and each use of a scalar variable is dominated by the unique definition. SSA form is commonly used for compiler intermediate languages because it simplifies certain analyses with moderate precision loss [Boi+08; HG06]. To convert conventional programs into SSA form phi-nodes are placed at points where multiple definitions of a live variable converge [Bra+13; Cyt+89].

Bibliography

[Abe+13] Abel, Andreas, Benz, Florian, Doerfert, Johannes, Dörr, Barbara, Hahn, Sebastian, Haupenthal, Florian, Jacobs, Michael, Moin, Amir H., Reineke, Jan, Schommer, Bernhard, and Wilhelm, Reinhard. "Impact of Resource Sharing on Performance and Performance Prediction: A Survey". In: *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. 2013, pp. 25–43. url: `https://doi.org/10.1007/978-3-642-40184-8%5C_3` (cit. on p. 2).

[AB15] Acharya, Aravind and Bondhugula, Uday. "PLUTO+: near-complete modeling of affine transformations for parallelism and locality". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. 2015, pp. 54–64. url: `http://doi.acm.org/10.1145/2688500.2688512` (cit. on pp. 22, 65, 93, 96).

[Adv+03] Adve, Vikram S., Lattner, Chris, Brukman, Michael, Shukla, Anand, and Gaeke, Brian. "LLVA: A Low-level Virtual Instruction Set Architecture". In: *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*. 2003, pp. 205–216. url: `https://doi.org/10.1109/MICRO.2003.1253196` (cit. on p. 206).

[Ahm03] Ahmad, Dave. "The Rising Threat of Vulnerabilities Due to Integer Errors". In: *IEEE Security & Privacy* 1.4 (2003), pp. 77–82. url: `https://doi.org/10.1109/MSECP.2003.1219077` (cit. on pp. 105, 107).

[ADT13] Ahn, Wonsun, Duan, Yuelu, and Torrellas, Josep. "DeAliaser: alias speculation using atomic region support". In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*. 2013, pp. 167–180. url: `http://doi.acm.org/10.1145/2451116.2451136` (cit. on pp. 102, 103).

[ASU86] Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. isbn: 0-201-10088-6. url: `http://www.worldcat.org/oclc/12285707` (cit. on pp. 79, 84, 87).

[ABD07] Alias, Christophe, Baray, Fabrice, and Darte, Alain. "Bee+Cl@k: an implementation of lattice-based array contraction in the source-to-source translator rose". In: *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07), San Diego, California, USA, June 13-15, 2007*. 2007, pp. 73–82. url: `http://doi.acm.org/10.1145/1254766.1254778` (cit. on p. 146).

[Alv+15] Alves, Péricles, Gruber, Fabian, Doerfert, Johannes, Lamprineas, Alexandros, Grosser, Tobias, Rastello, Fabrice, and Pereira, Fernando Magno Quintão. "Runtime Pointer Disambiguation". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 2015, pp. 589–606. url: `http://doi.acm.org/10.1145/2814270.2814285` (cit. on pp. 6, 22, 65, 96, 101, 102).

[Ami+12] Amini, Mehdi, Creusillet, Béatrice, Even, Stéphanie, Keryell, Ronan, Goubier, Onig, Guelton, Serge, McMahon, Janice Onanian, Pasquier, François-Xavier, Péan, Grégoire, and Villalon, Pierre. "Par4all: From convex array regions to heterogeneous computing". In: *Second International Workshop on Polyhedral Compilation Techniques*. IMPACT'12. 2012 (cit. on p. 63).

[AI91] Ancourt, Corinne and Irigoin, François. “Scanning Polyhedra with DO Loops”. In: *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Williamsburg, Virginia, USA, April 21-24, 1991*. 1991, pp. 39–50. url: http://doi.acm.org/10.1145/109625.109631 (cit. on pp. 9, 14).

[Atz+16] Atzeni, Simone, Gopalakrishnan, Ganesh, Rakamaric, Zvonimir, Ahn, Dong H., Laguna, Ignacio, Schulz, Martin, Lee, Gregory L., Protze, Joachim, and Müller, Matthias S. “ARCHER: Effectively Spotting Data Races in Large OpenMP Applications”. In: *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. 2016, pp. 53–62. url: https://doi.org/10.1109/IPDPS.2016.68 (cit. on pp. 24, 56).

[BWZ94] Bachmann, Olaf, Wang, Paul S., and Zima, Eugene V. “Chains of Recurrences - a Method to Expedite the Evaluation of Closed-form Functions”. In: *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC ’94, Oxford, UK, July 20-22, 1994*. 1994, pp. 242–249. url: http://doi.acm.org/10.1145/190347.190423 (cit. on pp. 17, 50, 57, 110, 124, 186).

[Bag+11] Baghdadi, Riyadh, Cohen, Albert, Bastoul, Cédric, Pouchet, Louis-Noël, and Rauchwerger, Lawrence. “The Potential of Synergistic Static, Dynamic and Speculative Loop Nest Optimizations for Automatic Parallelization”. In: *CoRR* abs/1111.6756 (2011). arXiv: 1111.6756. url: http://arxiv.org/abs/1111.6756 (cit. on pp. 62, 77).

[Bag+13] Baghdadi, Riyadh, Cohen, Albert, Verdoolaege, Sven, and Trifunovic, Konrad. “Improved loop tiling based on the removal of spurious false dependences”. In: *TACO* 9.4 (2013), 52:1–52:26. url: http://doi.acm.org/10.1145/2400682.2400711 (cit. on pp. 9, 173).

[Bag+15] Baghdadi, Riyadh, Beaugnon, Ulysse, Cohen, Albert, Grosser, Tobias, Kruse, Michael, Reddy, Chandan, Verdoolaege, Sven, Betts, Adam, Donaldson, Alastair F., Ketema, Jeroen, Absar, Javed, Haastregt, Sven van, Kravets, Alexey, Lokhmotov, Anton, David, Robert, and Hajiyev, Elnar. “PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming”. In: *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*. 2015, pp. 138–149. url: https://doi.org/10.1109/PACT.2015.17 (cit. on pp. 56, 63, 93, 96, 121, 122, 191).

[Bag+18] Baghdadi, Riyadh, Ray, Jessica, Romdhane, Malek Ben, Sozzo, Emanuele Del, Suriana, Patricia, Kamil, Shoaib, and Amarasinghe, Saman P. “Tiramisu: A Code Optimization Framework for High Performance Systems”. In: *CoRR* abs/1804.10694 (2018). arXiv: 1804.10694. url: http://arxiv.org/abs/1804.10694 (cit. on pp. 3, 122, 191).

[BB14] Bagnères, Lénaïc and Bastoul, Cédric. “Switchable Scheduling for Runtime Adaptation of Optimization”. In: *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. 2014, pp. 222–233. url: https://doi.org/10.1007/978-3-319-09873-9_19 (cit. on pp. 77, 176).

[Bag+16] Bagnères, Lénaïc, Zinenko, Oleksandr, Huot, Stéphane, and Bastoul, Cédric. “Opening polyhedral compiler’s black box”. In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*. 2016, pp. 128–138. url: http://doi.acm.org/10.1145/2854038.2854048 (cit. on p. 3).

[Bai+91] Bailey, David H., Barszcz, Eric, Barton, John T., Browning, D. S., Carter, Robert L., Dagum, Leonardo, Fatoohi, Rod A., Frederickson, Paul O., Lasinski, T. A., Schreiber, Robert, Simon, Horst D., Venkatakrishnan, V., and Weeratunga, Sisira. “The Nas Parallel Benchmarks”. In: *IJHPCA* (1991). url: https://doi.org/10.1177/109434209100500306 (cit. on p. 148).

[BPB12] Bandishti, Vinayaka, Pananilath, Irshad, and Bondhugula, Uday. “Tiling stencil computations to maximize parallelism”. In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC ’12, Salt Lake City, UT, USA - November 11 - 15, 2012*. 2012, p. 40. url: https://doi.org/10.1109/SC.2012.107 (cit. on p. 162).

[BRS10] Baskaran, Muthu Manikandan, Ramanujam, J., and Sadayappan, P. “Automatic C-to-CUDA Code Generation for Affine Programs”. In: *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. 2010, pp. 244–263. url: https://doi.org/10.1007/978-3-642-11970-5_14 (cit. on pp. 8, 208).

[Bas+03] Bastoul, Cédric, Cohen, Albert, Girbal, Sylvain, Sharma, Saurabh, and Temam, Olivier. “Putting Polyhedral Loop Transformations to Work”. In: *Languages and Compilers for Parallel Computing, 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003, Revised Papers*. 2003, pp. 209–225. url: https://doi.org/10.1007/978-3-540-24644-2_14 (cit. on pp. 21–23, 49, 63, 65, 93).

[Bas04a] Bastoul, Cédric. “Code Generation in the Polyhedral Model Is Easier Than You Think”. In: *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*. 2004, pp. 7–16. url: http://doi.ieeecomputersociety.org/10.1109/PACT.2004.10018 (cit. on pp. 3, 14).

[Bas04b] Bastoul, Cédric. “Improving Data Locality in Static Control Programs”. PhD thesis. University Paris 6, Pierre et Marie Curie, France, 2004 (cit. on p. 14).

[Ben+10] Benabderrahmane, Mohamed-Walid, Pouchet, Louis-Noël, Cohen, Albert, and Bastoul, Cédric. “The Polyhedral Model Is More Widely Applicable Than You Think”. In: *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. 2010, pp. 283–303. url: https://doi.org/10.1007/978-3-642-11970-5_16 (cit. on pp. 22, 51, 56, 62, 93, 123).

[BCM94] Bernstein, David, Cohen, Doron, and Maydan, Dror E. “Dynamic memory disambiguation for array references”. In: *Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, California, USA, November 30 - December 2, 1994*. 1994, pp. 105–111. url: https://doi.org/10.1109/MICRO.1994.717426 (cit. on p. 102).

[Bez+12] Bezanson, Jeff, Karpinski, Stefan, Shah, Viral B., and Edelman, Alan. “Julia: A Fast Dynamic Language for Technical Computing”. In: *CoRR* abs/1209.5145 (2012). arXiv: 1209.5145. url: http://arxiv.org/abs/1209.5145 (cit. on p. 174).

[BBC16] Bhaskaracharya, Somashekaracharya G., Bondhugula, Uday, and Cohen, Albert. “SMO: an integrated approach to intra-array and inter-array storage optimization”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 526–538. url: http://doi.acm.org/10.1145/2837614.2837636 (cit. on pp. 9, 22, 23, 146, 173).

[BA13] Bhattacharyya, Arnamoy and Amaral, José Nelson. “Automatic speculative parallelization of loops using polyhedral dependence analysis”. In: *Proceedings of the First International Workshop on Code Optimisation for Multi and Many Cores, COSMIC@CGO 2013, Shenzhen, China, February 24, 2013*. 2013, p. 1. url: http://doi.acm.org/10.1145/2446920.2446921 (cit. on p. 24).

[Ble89] Blelloch, Guy E. “Scans as Primitive Parallel Operations”. In: *IEEE Trans. Computers* 38.11 (1989), pp. 1526–1538. url: https://doi.org/10.1109/12.42122 (cit. on p. 125).

[BE95] Blume, William and Eigenmann, Rudolf. “Symbolic range propagation”. In: *Proceedings of IPPS ’95, The 9th International Parallel Processing Symposium, April 25-28, 1995, Santa Barbara, California, USA*. 1995, pp. 357–363. url: https://doi.org/10.1109/IPPS.1995.395956 (cit. on p. 57).

[BA98] Bodík, Rastislav and Anik, Sadun. “Path-Sensitive Value-Flow Analysis”. In: *POPL ’98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 1998, pp. 237–251. url: http://doi.acm.org/10.1145/268946.268966 (cit. on p. 87).

[BGS99] Bodík, Rastislav, Gupta, Rajiv, and Soffa, Mary Lou. “Load-Reuse Analysis: Design and Evaluation”. In: *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*. 1999, pp. 64–76. url: http://doi.acm.org/10.1145/301618.301643 (cit. on pp. 79, 87).

[BGS00] Bodík, Rastislav, Gupta, Rajiv, and Sarkar, Vivek. “ABCD: eliminating array bounds checks on demand”. In: *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. 2000, pp. 321–333. url: http://doi.acm.org/10.1145/349299.349342 (cit. on pp. 77, 88, 91).

[Boi+08] Boissinot, Benoit, Hack, Sebastian, Grund, Daniel, Dinechin, Benoît Dupont de, and Rastello, Fabrice. “Fast liveness checking for ssa-form programs”. In: *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*. 2008, pp. 35–44. url: http://doi.acm.org/10.1145/1356058.1356064 (cit. on p. 208).

[Bon+08] Bondhugula, Uday, Hartono, Albert, Ramanujam, J., and Sadayappan, P. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 2008, pp. 101–113. url: http://doi.acm.org/10.1145/1375581.1375595 (cit. on pp. 8, 14, 15, 23, 35, 93, 96, 146, 161, 162, 176, 207, 208).

[Bon+10] Bondhugula, Uday, Günlük, Oktay, Dash, Sanjeeb, and Renganarayanan, Lakshminarayanan. “A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler”. In: *19th International Conference on Parallel Architecture and Compilation Techniques, PACT 2010, Vienna, Austria, September 11-15, 2010*. 2010, pp. 343–352. url: http://doi.acm.org/10.1145/1854273.1854317 (cit. on pp. 3, 9, 21, 23, 49, 146, 162, 184).

[Bra+13] Braun, Matthias, Buchwald, Sebastian, Hack, Sebastian, Leißa, Roland, Mallon, Christoph, and Zwinkau, Andreas. “Simple and Efficient Construction of Static Single Assignment Form”. In: *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 102–122. url: https://doi.org/10.1007/978-3-642-37051-9%5C_6 (cit. on p. 208).

[Bru+07] Brumley, David, Song, Dawn Xiaodong, Chiueh, Tzi-cker, Johnson, Rob, and Lin, Huijia. “RICH: Automatically Protecting Against Integer-Based Vulnerabilities”. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*. 2007. url: http://www.isoc.org/isoc/conferences/ndss/07/papers/efficient_detection_integer-based_attacks.pdf (cit. on pp. 105, 112).

[BLH12] Bygde, Stefan, Lisper, Björn, and Holsti, Niklas. “Fully Bounded Polyhedral Analysis of Integers with Wrapping”. In: *Electr. Notes Theor. Comput. Sci.* 288 (2012), pp. 3–13. url: https://doi.org/10.1016/j.entcs.2012.10.003 (cit. on p. 105).

[C11] C. *The ANSI C standard (C11)*. WG14 N1570. ISO, 2011. url: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf (cit. on p. 68).

[CWC16] Caamaño, Juan Manuel Martinez, Wolff, Willy, and Clauss, Philippe. “Code Bones: Fast and Flexible Code Generation for Dynamic and Speculative Polyhedral Optimization”. In: *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*. 2016, pp. 225–237. url: https://doi.org/10.1007/978-3-319-43659-3_17 (cit. on p. 24).

[Caa+17] Caamaño, Juan Manuel Martinez, Selva, Manuel, Clauss, Philippe, Baloian, Artyom, and Wolff, Willy. “Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones”. In: *Concurrency and Computation: Practice and Experience* 29.15 (2017). url: https://doi.org/10.1002/cpe.4192 (cit. on pp. 9, 62, 77).

[CCK90] Callahan, David, Carr, Steve, and Kennedy, Ken. "Improving Register Allocation for Subscripted Variables". In: *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*. 1990, pp. 53–65. url: http://doi.acm.org/10.1145/93542.93553 (cit. on p. 30).

[Cal+97] Calland, Pierre-Yves, Darte, Alain, Robert, Yves, and Vivien, Frédéric. "Plugging Anti and Output Dependence Removal Techniques Into Loop Parallelization Algorithm". In: *Parallel Computing* 23.1-2 (1997), pp. 251–266. url: https://doi.org/10.1016/S0167-8191(96)00108-1 (cit. on p. 173).

[Cha+16] Chatarasi, Prasanth, Shirako, Jun, Kong, Martin, and Sarkar, Vivek. "An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection". In: *Languages and Compilers for Parallel Computing - 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28-30, 2016, Revised Papers*. 2016, pp. 106–120. url: https://doi.org/10.1007/978-3-319-52709-3_10 (cit. on p. 56).

[Che+09] Chen, Ping, Wang, Yi, Xin, Zhi, Mao, Bing, and Xie, Li. "BRICK: A Binary Tool for Run-Time Detecting and Locating Integer-Based Vulnerability". In: *Proceedings of the The Forth International Conference on Availability, Reliability and Security, ARES 2009, March 16-19, 2009, Fukuoka, Japan*. 2009, pp. 208–215. url: https://doi.org/10.1109/ARES.2009.77 (cit. on p. 105).

[CH00] Cheng, Ben-Chung and Hwu, Wen-mei W. "Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation". In: *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. 2000, pp. 57–69. url: http://doi.acm.org/10.1145/349299.349311 (cit. on p. 87).

[Che16] Cheng, Xi. "RABIEF: range analysis based integer error fixing". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 1094–1096. url: http://doi.acm.org/10.1145/2950290.2983961 (cit. on p. 113).

[Che+17] Cheng, Xi, Zhou, Min, Song, Xiaoyu, Gu, Ming, and Sun, Jiaguang. "IntPTI: automatic integer error repair with proper-type inference". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 2017, pp. 996–1001. url: https://doi.org/10.1109/ASE.2017.8115718 (cit. on p. 113).

[Cho+98] Chow, Fred C., Kennedy, Robert, Liu, Shin-Ming, Lo, Raymond, and Tu, Peng. "Register Promotion by Partial Redundancy Elimination of Loads and Stores". In: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*. 1998, pp. 26–37. url: http://doi.acm.org/10.1145/277650.277659 (cit. on pp. 79, 84, 87).

[Cla98] Clark, David. "OpenMP: a parallel standard for the masses". In: *IEEE Concurrency* 6.1 (1998), pp. 10–12. url: https://doi.org/10.1109/4434.656771 (cit. on p. 15).

[Cla+11] Clauss, Philippe, Garbervetsky, Diego, Loechner, Vincent, and Verdoolaege, Sven. "Polyhedral Techniques for Parametric Memory Requirement Estimation". In: *Energy-Aware Memory Management for Embedded Multimedia Systems: A Computer-Aided Design Approach*. Chapman & Hall/Crc Computer and Information Science. 2011. url: https://hal.inria.fr/hal-00671226 (cit. on p. 9).

[Cli95] Click, Cliff. "Global Code Motion / Global Value Numbering". In: *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*. 1995, pp. 246–257. url: http://doi.acm.org/10.1145/207110.207154 (cit. on pp. 79, 84, 87).

[CGT04] Cohen, Albert, Girbal, Sylvain, and Temam, Olivier. "A Polyhedral Approach to Ease the Composition of Program Transformations". In: *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, Proceedings*. 2004, pp. 292–303. url: https://doi.org/10.1007/978-3-540-27866-5%5C_38 (cit. on pp. 3, 8, 12, 13, 27, 208).

[CH13] Coker, Zack and Hafiz, Munawar. "Program transformations to fix C integers". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 2013, pp. 792–801. url: https://doi.org/10.1109/ICSE.2013.6606625 (cit. on pp. 105, 113).

[Col95] Collard, Jean-Francois. "Automatic parallelization of *while* -loops using speculative execution". In: *International Journal of Parallel Programming* 23.2 (1995), pp. 191–219. url: https://doi.org/10.1007/BF02577789 (cit. on p. 62).

[CBF95] Collard, Jean-Francois, Barthou, Denis, and Feautrier, Paul. "Fuzzy Array Dataflow Analysis". In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Santa Barbara, California, USA, July 19-21, 1995*. 1995, pp. 92–101. url: http://doi.acm.org/10.1145/209936.209947 (cit. on pp. 56, 62).

[CH78] Cousot, Patrick and Halbwachs, Nicolas. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. 1978, pp. 84–96. url: http://doi.acm.org/10.1145/512760.512770 (cit. on pp. 9, 77, 84, 105).

[Cou+05] Cousot, Patrick, Cousot, Radhia, Feret, Jérôme, Mauborgne, Laurent, Miné, Antoine, Monniaux, David, and Rival, Xavier. "The ASTREÉ Analyzer". In: *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. 2005, pp. 21–30. url: https://doi.org/10.1007/978-3-540-31987-0_3 (cit. on pp. 105, 113).

[Cou+13] Cousot, Patrick, Cousot, Radhia, Fähndrich, Manuel, and Logozzo, Francesco. "Automatic Inference of Necessary Preconditions". In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. 2013, pp. 128–148. url: https://doi.org/10.1007/978-3-642-35873-9_10 (cit. on pp. 77, 78).

[CI95] Creusillet, Béatrice and Irigoin, François. "Interprocedural Array Region Analyses". In: *Languages and Compilers for Parallel Computing, 8th International Workshop, LCPC'95, Columbus, Ohio, USA, August 10-12, 1995, Proceedings*. 1995, pp. 46–60. url: https://doi.org/10.1007/BFb0014191 (cit. on p. 63).

[Cyt+89] Cytron, Ron, Ferrante, Jeanne, Rosen, Barry K., Wegman, Mark N., and Zadeck, F. Kenneth. "An Efficient Method of Computing Static Single Assignment Form". In: *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. 1989, pp. 25–35. url: http://doi.acm.org/10.1145/75277.75280 (cit. on p. 208).

[Dam+15] Damschen, Marvin, Riebler, Heinrich, Vaz, Gavin, and Plessl, Christian. "Transparent offloading of computational hotspots from binary code to Xeon Phi". In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*. 2015, pp. 1078–1083. url: http://dl.acm.org/citation.cfm?id=2757063 (cit. on p. 9).

[DYR02] Dang, Francis H., Yu, Hao, and Rauchwerger, Lawrence. "The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops". In: *16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings*. 2002. url: https://doi.org/10.1109/IPDPS.2002.1015493 (cit. on pp. 62, 125).

[Dan+10] Dannenberg, Roger B., Dormann, Will, Keaton, David, Seacord, Robert C., Svoboda, David, Volkovitsky, Alex, Wilson, Timothy, and Plum, Thomas. “As-If Infinitely Ranged Integer Model”. In: *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*. 2010, pp. 91–100. url: https://doi.org/10.1109/ISSRE.2010.29 (cit. on p. 112).

[DSV03] Darte, Alain, Schreiber, Robert, and Villard, Gilles. “Lattice-based memory allocation”. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2003, San Jose, California, USA, October 30 - November 1, 2003*. 2003, pp. 298–308. url: http://doi.acm.org/10.1145/951710.951749 (cit. on pp. 146, 173).

[DSV05] Darte, Alain, Schreiber, Robert, and Villard, Gilles. “Lattice-Based Memory Allocation”. In: *IEEE Trans. Computers* 54.10 (2005), pp. 1242–1257. url: https://doi.org/10.1109/TC.2005.167 (cit. on p. 9).

[DI15] Darte, Alain and Isoard, Alexandre. “Exact and Approximated Data-Reuse Optimizations for Tiling with Parametric Sizes”. In: *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 151–170. url: https://doi.org/10.1007/978-3-662-46663-6%5C_8 (cit. on p. 9).

[DIY16] Darte, Alain, Isoard, Alexandre, and Yuki, Tomofumi. “Extended lattice-based memory allocation”. In: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. 2016, pp. 218–228. url: http://doi.acm.org/10.1145/2892208.2892213 (cit. on pp. 9, 22, 23, 173).

[DMW98] Debray, Saumya K., Muth, Robert, and Weippert, Matthew. “Alias Analysis of Executable Code”. In: *POPL ’98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 1998, pp. 12–24. url: http://doi.acm.org/10.1145/268946.268948 (cit. on p. 87).

[Die+12] Dietz, Will, Li, Peng, Regehr, John, and Adve, Vikram S. “Understanding integer overflow in C/C++”. In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 2012, pp. 760–770. url: https://doi.org/10.1109/ICSE.2012.6227142 (cit. on pp. 105, 112).

[DMM98] Diwan, Amer, McKinley, Kathryn S., and Moss, J. Eliot B. “Type-Based Alias Analysis”. In: *Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*. 1998, pp. 106–117. url: http://doi.acm.org/10.1145/277650.277670 (cit. on pp. 87, 97, 102).

[Doe+13] Doerfert, Johannes, Clemens, Hammacher, Kevin, Streit, and Sebastian, Hack. “SPolly: Speculative Optimizations in the Polyhedral Model”. In: IMPACT’13. 2013 (cit. on pp. 6, 9, 21, 22, 24, 39, 40, 65, 69, 93, 96).

[DGP15] Doerfert, Johannes, Grosser, Tobias, and Pop, Sebastian. “Assumption Tracking for Optimistic Optimizations”. In: *LLVM HPC Workshop at SC*. 2015. url: https://llvm-hpc2-workshop.github.io/slides/Doerfert.pdf (cit. on pp. 3, 6, 121).

[Doe+15] Doerfert, Johannes, Streit, Kevin, Hack, Sebastian, and Benaissa, Zino. “Polly’s Polyhedral Scheduling in the Presence of Reductions”. In: *International Workshop on Polyhedral Compilation Techniques*. IMPACT’15. 2015. url: http://arxiv.org/abs/1505.07716 (cit. on pp. 6, 15, 23, 125–130, 135, 137, 139, 142, 173, 208).

[DGH17] Doerfert, Johannes, Grosser, Tobias, and Hack, Sebastian. “Optimistic Loop Optimization”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. 2017, pp. 292–304. url: http://dl.acm.org/citation.cfm?id=3049864 (cit. on pp. 3, 6, 9, 15, 21–23, 65–68, 71, 72, 74–77, 79, 83, 88, 91, 93, 94, 96, 99, 102, 105–107, 109, 112, 120).

[DH17a] Doerfert, Johannes and Hack, Sebastian. “Polyhedral Driven Optimizations on Real Code”. In: *LLVM Performance Workshop at CGO*. 2017. url: https://llvm.org/devmtg/2017-02-04/Polyhedral-Driven-Optimizations-on-Real-Codes.pdf (cit. on pp. 6, 183, 184, 186).

[DH17b] Doerfert, Johannes and Hack, Sebastian. “Polyhedral Value & Memory Analysis”. In: *US LLVM Developers Conference*. 2017 (cit. on pp. 6, 9, 17, 49, 186).

[DSH18] Doerfert, Johannes, Sharma, Shrey, and Hack, Sebastian. “Polyhedral expression propagation”. In: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. 2018, pp. 25–36. url: http://doi.acm.org/10.1145/3178372.3179529 (cit. on pp. 6, 22, 23, 30, 31, 146, 150, 155–157, 159, 161, 162, 164–167, 169, 171).

[Don87] Dongarra, Jack J. “The LINPACK Benchmark: An Explanation”. In: *Supercomputing, 1st International Conference, Athens, Greece, June 8-12, 1987, Proceedings*. 1987, pp. 456–474. url: https://doi.org/10.1007/3-540-18991-2_27 (cit. on p. 74).

[DGS93] Duesterwald, Evelyn, Gupta, Rajiv, and Soffa, Mary Lou. “A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations”. In: *Proceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*. 1993, pp. 68–77. url: http://doi.acm.org/10.1145/155090.155097 (cit. on p. 174).

[Eld+14] Elder, Matt, Lim, Junghee, Sharma, Tushar, Andersen, Tycho, and Reps, Thomas W. “Abstract Domains of Affine Relations”. In: *ACM Trans. Program. Lang. Syst.* 36.4 (2014), 11:1–11:73. url: http://doi.acm.org/10.1145/2651361 (cit. on p. 106).

[EGH94] Emami, Maryam, Ghiya, Rakesh, and Hendren, Laurie J. “Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers”. In: *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*. 1994, pp. 242–256. url: http://doi.acm.org/10.1145/178243.178264 (cit. on p. 102).

[Fea88a] Feautrier, Paul. “Array expansion”. In: *Proceedings of the 2nd international conference on Supercomputing, ICS 1988, Saint Malo, France, July 4-8, 1988*. 1988, pp. 429–441. url: http://doi.acm.org/10.1145/55364.55406 (cit. on pp. 23, 30, 31, 62, 136, 144, 145, 173, 205).

[Fea91] Feautrier, Paul. “Dataflow analysis of array and scalar references”. In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53. url: https://doi.org/10.1007/BF01407931 (cit. on pp. 3, 8, 9, 12, 14, 15, 57, 62, 97, 105, 136, 140, 151, 153, 163, 204, 207, 208).

[Fea92a] Feautrier, Paul. “Some efficient solutions to the affine scheduling problem. I. One-dimensional time”. In: *International Journal of Parallel Programming* 21.5 (1992), pp. 313–347. url: https://doi.org/10.1007/BF01407835 (cit. on pp. 8, 9, 14, 15, 49, 136, 207).

[Fea92b] Feautrier, Paul. “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time”. In: *International Journal of Parallel Programming* 21.6 (1992), pp. 389–420. url: https://doi.org/10.1007/BF01379404 (cit. on pp. 8, 9, 14, 21, 23, 136, 146, 176, 207, 208).

[FL11] Feautrier, Paul and Lengauer, Christian. “Polyhedron Model”. In: *Encyclopedia of Parallel Computing*. 2011, pp. 1581–1592. url: https://doi.org/10.1007/978-0-387-09766-4_502 (cit. on pp. 2, 8, 206).

[Fea88b] Feautrier, Paul. “Parametric integer programming”. In: *RAIRO Recherche Opérationnelle* 22.3 (1988), pp. 243–268 (cit. on pp. 105, 106).

[FE02] Fernández, Manel and Espasa, Roger. “Speculative Alias Analysis for Executable Code”. In: *2002 International Conference on Parallel Architectures and Compilation Techniques (PACT 2002), 22-25 September 2002, Charlottesville, VA, USA*. 2002, pp. 222–231. url: https://doi.org/10.1109/PACT.2002.1106020 (cit. on pp. 102, 103).

[FG94] Fisher, Allan L. and Ghuloum, Anwar M. “Parallelizing Complex Scans and Reductions”. In: *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*. 1994, pp. 135–146. url: http://doi.acm.org/10.1145/178243.178255 (cit. on pp. 125, 134, 142–144).

[Fit+00] Fitzgerald, Robert P., Knoblock, Todd B., Ruf, Erik, Steensgaard, Bjarne, and Tarditi, David. “Marmot: an optimizing compiler for Java”. In: *Softw., Pract. Exper.* 30.3 (2000), pp. 199–232. url: https://doi.org/10.1002/(SICI)1097-024X(200003)30:3%3C199::AID-SPE296%3E3.0.CO;2-2 (cit. on p. 87).

[FM11] Fuller, Samuel H. and Millett, Lynette I. “Computing Performance: Game Over or Next Level?” In: *IEEE Computer* 44.1 (2011), pp. 31–38. url: https://doi.org/10.1109/MC.2011.15 (cit. on p. 1).

[Gal+94] Gallagher, David M., Chen, William Y., Mahlke, Scott A., Gyllenhaal, John C., and Hwu, Wen-mei W. “Dynamic Memory Disambiguation Using the Memory Conflict Buffer”. In: *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*. 1994, pp. 183–193. url: http://doi.acm.org/10.1145/195473.195534 (cit. on p. 103).

[Gam+08] Gampe, Andreas, Ronne, Jeffery von, Niedzielski, David, and Psarris, Kleanthis. “Speculative improvements to verifiable bounds check elimination”. In: *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, PPPJ 2008, Modena, Italy, September 9-11, 2008*. 2008, pp. 85–94. url: http://doi.acm.org/10.1145/1411732.1411745 (cit. on pp. 77, 88, 91).

[Gan+14] Gange, Graeme, Navas, Jorge A., Schachte, Peter, Søndergaard, Harald, and Stuckey, Peter J. “Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss”. In: *ACM Trans. Program. Lang. Syst.* 37.1 (2014), 1:1–1:35. url: http://doi.acm.org/10.1145/2651360 (cit. on p. 113).

[GGL99] Geigl, Max, Griebl, Martin, and Lengauer, Christian. “Termination detection in parallel loop nests with while loops”. In: *Parallel Computing* 25.12 (1999), pp. 1489–1510. url: https://doi.org/10.1016/S0167-8191(99)00063-0 (cit. on p. 62).

[GO17] Ginsbach, Philip and O’Boyle, Michael F. P. “Discovery and exploitation of general reductions: a constraint based approach”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. 2017, pp. 269–280. url: http://dl.acm.org/citation.cfm?id=3049862 (cit. on pp. 125, 142).

[Gir+06] Girbal, Sylvain, Vasilache, Nicolas, Bastoul, Cédric, Cohen, Albert, Parello, David, Sigler, Marc, and Temam, Olivier. “Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies”. In: *International Journal of Parallel Programming* 34.3 (2006), pp. 261–317. url: https://doi.org/10.1007/s10766-006-0012-3 (cit. on pp. 22, 63).

[GL94] Griebl, Martin and Lengauer, Christian. “On Scanning Space-Time Mapped While Loops”. In: *Parallel Processing: CONPAR 94 - VAPP VI, Third Joint International Conference on Vector and Parallel Processing, Linz, Austria, September 6-8, 1994, Proceedings*. 1994, pp. 677–688. url: https://doi.org/10.1007/3-540-58430-7_59 (cit. on p. 62).

[GC95] Griebl, Martin and Collard, Jean-Francois. “Generation of Synchronous Code for Automatic Parallelization of while Loops”. In: *Euro-Par ’95 Parallel Processing, First International Euro-Par Conference, Stockholm, Sweden, August 29-31, 1995, Proceedings*. 1995, pp. 315–326. url: https://doi.org/10.1007/BFb0020474 (cit. on pp. 51, 62).

[GL96] Griebl, Martin and Lengauer, Christian. “The Loop Parallelizer LooPo”. In: *Proceedings of the 6th Workshop on Compilers for Parallel Computers (CPC’96)*. 1996, pp. 311–320 (cit. on pp. 8, 9, 208).

[GFL99] Griebl, Martin, Feautrier, Paul, and Lengauer, Christian. “On Index Set Splitting”. In: *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, Newport Beach, California, USA, October 12-16, 1999*. 1999, pp. 274–282. url: https://doi.org/10.1109/PACT.1999.807572 (cit. on p. 172).

[Gro11] Grosser, Tobias. “Enabling Polyhedral Optimizations in LLVM”. diploma thesis. 2011 (cit. on pp. 15, 146).

[GGL12] Grosser, Tobias, Größlinger, Armin, and Lengauer, Christian. "Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation". In: *Parallel Processing Letters* 22.4 (2012). url: https://doi.org/10.1142/S0129626412500107 (cit. on pp. 3, 8, 15, 21, 23, 63, 65, 162, 176, 206, 208).

[Gro+15] Grosser, Tobias, Ramanujam, Jagannathan, Pouchet, Louis-Noël, Sadayappan, P., and Pop, Sebastian. "Optimistic Delinearization of Parametrically Sized Arrays". In: *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*. 2015, pp. 351–360. url: http://doi.acm.org/10.1145/2751205.2751248 (cit. on pp. 9, 13, 15, 67, 68, 71, 75, 77, 91, 98).

[GVC15] Grosser, Tobias, Verdoolaege, Sven, and Cohen, Albert. "Polyhedral AST Generation Is More Than Scanning Polyhedra". In: *ACM Trans. Program. Lang. Syst.* 37.4 (2015), 12:1–12:50. url: http://doi.acm.org/10.1145/2743016 (cit. on pp. 14, 15, 35, 66, 98, 154).

[GH16] Grosser, Tobias and Hoefler, Torsten. "Polly-ACC Transparent compilation to heterogeneous hardware". In: *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3, 2016*. 2016, 1:1–1:13. url: http://doi.acm.org/10.1145/2925426.2926286 (cit. on pp. 2, 9, 15, 22, 23).

[Guo+06] Guo, Bolei, Wu, Youfeng, Wang, Cheng, Bridges, Matthew J., Ottoni, Guilherme, Vachharajani, Neil, Chang, Jonathan, and August, David I. "Selective Runtime Memory Disambiguation in a Dynamic Binary Translator". In: *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*. 2006, pp. 65–79. url: https://doi.org/10.1007/11688839_6 (cit. on pp. 96, 102, 103).

[GRQ02] Gupta, Gautam, Rajopadhye, Sanjay V., and Quinton, Patrice. "Scheduling reductions on realistic machines". In: *SPAA*. 2002, pp. 117–126. url: http://doi.acm.org/10.1145/564870.564888 (cit. on pp. 144, 145).

[GR06] Gupta, Gautam and Rajopadhye, Sanjay V. "Simplifying reductions". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 2006, pp. 30–41. url: http://doi.acm.org/10.1145/1111037.1111041 (cit. on pp. 125, 143, 145).

[HG06] Hack, Sebastian and Goos, Gerhard. "Optimal register allocation for SSA-form programs in polynomial time". In: *Inf. Process. Lett.* 98.4 (2006), pp. 150–155. url: https://doi.org/10.1016/j.ipl.2006.01.008 (cit. on p. 208).

[Ham+16] Hammacher, Clemens, Streit, Kevin, Zeller, Andreas, and Hack, Sebastian. "Thread-level speculation with kernel support". In: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. 2016, pp. 1–11. url: http://doi.acm.org/10.1145/2892208.2892221 (cit. on p. 103).

[Ham17] Hammacher, Clemens. "Efficient runtime systems for speculative parallelization". PhD thesis. Saarland University, Saarbrücken, Germany, 2017. url: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/26882 (cit. on p. 91).

[Ham+17] Hammer, Julian, Eitzinger, Jan, Hager, Georg, and Wellein, Gerhard. "Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels". In: *CoRR* abs/1702.04653 (2017). arXiv: 1702.04653. url: http://arxiv.org/abs/1702.04653 (cit. on p. 162).

[Ham+18] Hammer, Julian, Doerfert, Johannes, Hager, Georg, Wellein, Gerhard, and Hack, Sebastian. "Cache-aware Scheduling and Performance Modeling with LLVM-Polly and Kerncraft Assumption Tracking for Optimistic Optimizations". In: *LLVM Performance Workshop at CGO*. 2018. url: https://llvm.org/devmtg/2018-02-24/slides/hammer_2018_CGO_LLVM_perf.pdf (cit. on pp. 9, 162).

[Hav97] Havlak, Paul. "Nesting of Reducible and Irreducible Loops". In: *ACM Trans. Program. Lang. Syst.* 19.4 (1997), pp. 557–567. url: http://doi.acm.org/10.1145/262004.262005 (cit. on p. 50).

[HU74] Hecht, Matthew S. and Ullman, Jeffrey D. “Characterizations of Reducible Flow Graphs”. In: *J. ACM* 21.3 (1974), pp. 367–375. url: http://doi.acm.org/10.1145/321832.321835 (cit. on pp. 17, 202).

[Heg+14] Hegarty, James, Brunhaver, John, DeVito, Zachary, Ragan-Kelley, Jonathan, Cohen, Noy, Bell, Steven, Vasilyev, Artem, Horowitz, Mark, and Hanrahan, Pat. “Darkroom: compiling high-level image processing code into hardware pipelines”. In: *ACM Trans. Graph.* 33.4 (2014), 144:1–144:11. url: http://doi.acm.org/10.1145/2601097.2601174 (cit. on pp. 121, 173, 174).

[Hoe+09] Hoenicke, Jochen, Leino, K. Rustan M., Podelski, Andreas, Schäf, Martin, and Wies, Thomas. “It’s Doomed; We Can Prove It”. In: *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. 2009, pp. 338–353. url: https://doi.org/10.1007/978-3-642-05089-3_22 (cit. on pp. 77, 78).

[HSS94] Huang, Andrew S., Slavenburg, Gert, and Shen, John Paul. “Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation”. In: *Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, April 1994*. 1994, pp. 200–210. url: https://doi.org/10.1109/ISCA.1994.288149 (cit. on p. 103).

[IJT91] Irigoin, François, Jouvelot, Pierre, and Triolet, Rémi. “Semantical interprocedural parallelization: an overview of the PIPS project”. In: *Proceedings of the 5th international conference on Supercomputing, ICS 1991, Cologne, Germany, June 17-21, 1991*. 1991, pp. 244–251. url: http://doi.acm.org/10.1145/109025.109086 (cit. on p. 63).

[JB18] Jangda, Abhinav and Bondhugula, Uday. “An effective fusion and tile size model for optimizing image processing pipelines”. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. 2018, pp. 261–275. url: http://doi.acm.org/10.1145/3178487.3178507 (cit. on p. 3).

[Jeo+17] Jeong, Sehun, Jeon, Minseok, Cha, Sung Deok, and Oh, Hakjoo. “Data-driven context-sensitivity for points-to analysis”. In: *PACMPL* 1.OOPSLA (2017), 100:1–100:28. url: http://doi.acm.org/10.1145/3133924 (cit. on p. 102).

[Jim+12] Jimborean, Alexandra, Mastrangelo, Luis, Loechner, Vincent, and Clauss, Philippe. “VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework”. In: *Compiler Construction - 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 220–239. url: https://doi.org/10.1007/978-3-642-28652-0_12 (cit. on pp. 77, 91).

[Jim+13a] Jimborean, Alexandra, Clauss, Philippe, Dollinger, Jean-François, Loechner, Vincent, and Caamaño, Juan Manuel Martinez. “Dynamic and Speculative Polyhedral Parallelization of Loop Nests Using Binary Code Patterns”. In: *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*. 2013, pp. 2575–2578. url: https://doi.org/10.1016/j.procs.2013.05.443 (cit. on pp. 9, 62, 77).

[Jim+13b] Jimborean, Alexandra, Clauss, Philippe, Caamaño, Juan Manuel Martinez, and Sukumaran-Rajam, Aravind . “Online Dynamic Dependence Analysis for Speculative Polyhedral Parallelization”. In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. 2013, pp. 191–202. url: https://doi.org/10.1007/978-3-642-40047-6_21 (cit. on pp. 23, 91).

[Joh+12] Johnson, Nick P., Kim, Hanjun, Prabhu, Prakash, Zaks, Ayal, and August, David I. “Speculative separation for privatization and reductions”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12, Beijing, China - June 11 - 16, 2012*. 2012, pp. 359–370. doi: 10.1145/2254064.2254107. url: http://doi.acm.org/10.1145/2254064.2254107 (cit. on p. 125).

[Jou86] Jouvelot, Pierre. “Parallelization by Semantic Detection of Reductions”. In: *ESOP 86, European Symposium on Programming, Saarbrücken, Federal Republic of Germany, March 17-19, 1986, Proceedings*. 1986, pp. 223–236. url: https://doi.org/10.1007/3-540-16442-1_17 (cit. on p. 125).

[JD89] Jouvelot, Pierre and Dehbonei, Babak. “A unified semantic approach for the vectorization and parallelization of generalized reductions”. In: *Proceedings of the 3rd international conference on Supercomputing, ICS 1989, Heraklion, Crete, June 5-9, 1989*. 1989, pp. 186–194. url: http://doi.acm.org/10.1145/318789.318810 (cit. on pp. 125, 142, 143).

[Jun15] Jung, Tina. “A Hybrid Approach for Parametric Memory Dependence Analysis”. Bachelor Thesis. Saarland University, 2015 (cit. on pp. 24, 186).

[Kan12] Kannavara, Raghudeep. “Securing Opensource Code via Static Analysis”. In: *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. 2012, pp. 429–436. url: https://doi.org/10.1109/ICST.2012.123 (cit. on p. 113).

[KMW67] Karp, Richard M., Miller, Raymond E., and Winograd, Shmuel. “The Organization of Computations for Uniform Recurrence Equations”. In: *J. ACM* 14.3 (1967), pp. 563–590. url: http://doi.acm.org/10.1145/321406.321418 (cit. on p. 9).

[KP95] Kelly, Wayne and Pugh, William. “A Unifying Framework for Iteration Reordering Transformations”. In: *Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP., IEEE First International Conference on*. Vol. 1. IEEE. 1995, pp. 153–162 (cit. on p. 13).

[Kel+95] Kelly, Wayne, Maslov, Vadim, Pugh, William, Rosser, Evan, Shpeisman, Tatiana, and Wonnacott, David. “The omega library interface guide”. In: (1995) (cit. on p. 106).

[Kev90] Kevin O’Brien. *Predictive Commoning: A method of optimizing loops containing references to consecutive array elements*. Tech. rep. IBM Thomas J. Watson Research Center, 1990 (cit. on pp. 162, 172).

[KCB07] Khan, Minhaj Ahmad, Charles, Henri-Pierre, and Barthou, Denis. “An Effective Automated Approach to Specialization of Code”. In: *Languages and Compilers for Parallel Computing, 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007, Revised Selected Papers*. 2007, pp. 308–322. url: https://doi.org/10.1007/978-3-540-85261-2_21 (cit. on p. 91).

[KS10] King, Andy and Søndergaard, Harald. “Automatic Abstraction for Congruences”. In: *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*. 2010, pp. 197–213. url: https://doi.org/10.1007/978-3-642-11319-2_16 (cit. on p. 106).

[KRS94] Knoop, Jens, Rüthing, Oliver, and Steffen, Bernhard. “Partial Dead Code Elimination”. In: *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*. 1994, pp. 147–158. url: http://doi.acm.org/10.1145/178243.178256 (cit. on p. 173).

[KRS98] Knoop, Jens, Rüthing, Oliver, and Steffen, Bernhard. “Code Motion and Code Placement: Just Synonyms?” In: *Programming Languages and Systems - ESOP’98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. 1998, pp. 154–169. url: https://doi.org/10.1007/BFb0053569 (cit. on p. 87).

[KRS99] Knoop, Jens, Rüthing, Oliver, and Steffen, Bernhard. “Expansion-Based Removal of Semantic Partial Redundancies”. In: *Compiler Construction, 8th International Conference, CC’99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*. 1999, pp. 91–106. url: https://doi.org/10.1007/978-3-540-49051-7_7 (cit. on p. 87).

[KS73] Kogge, Peter M. and Stone, Harold S. “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations”. In: *IEEE Trans. Computers* 22.8 (1973), pp. 786–793. url: https://doi.org/10.1109/TC.1973.5009159 (cit. on p. 143).

[Kol+13] Koliai, Souad, Bendifallah, Zakaria, Tribalat, Mathieu, Valensi, Cédric, Acquaviva, Jean-Thomas, and Jalby, William. "Quantifying performance bottleneck cost through differential analysis". In: *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013*. 2013, pp. 263–272. url: `http://doi.acm.org/10.1145/2464996.2465440` (cit. on p. 2).

[Kon+13] Kong, Martin, Veras, Richard, Stock, Kevin, Franchetti, Franz, Pouchet, Louis-Noël, and Sadayappan, P. "When polyhedral transformations meet SIMD code generation". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 127–138. url: `http://doi.acm.org/10.1145/2462156.2462187` (cit. on pp. 14, 140, 207).

[Kös+14] Köster, Marcel, Leißa, Roland, Hack, Sebastian, Membarth, Richard, and Slusallek, Philipp. "Code Refinement of Stencil Codes". In: *Parallel Processing Letters* 24.3 (2014). url: `https://doi.org/10.1142/S0129626414410035` (cit. on p. 121).

[KG18] Kruse, Michael and Grosser, Tobias. "DeLICM: scalar dependence removal at zero memory cost". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*. 2018, pp. 241–253. url: `http://doi.acm.org/10.1145/3168815` (cit. on pp. 9, 15, 23, 31, 146, 174).

[KP16] Kumar, Aditya and Pop, Sebastian. "SCoP Detection: A Fast Algorithm for Industrial Compilers". In: *International Workshop on Polyhedral Compilation Techniques*. IMPACT'16. 2016 (cit. on pp. 31, 51).

[Kur17] Kurtenacker, Matthias. "On Synchronization in the Polyhedral Model". Bachelor Thesis. Saarland University, 2017 (cit. on pp. 9, 120, 123).

[LA04] Lattner, Chris and Adve, Vikram S. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 2004, pp. 75–88. url: `https://doi.org/10.1109/CGO.2004.1281665` (cit. on pp. 3, 15, 78, 205).

[LLA07] Lattner, Chris, Lenharth, Andrew, and Adve, Vikram S. "Making context-sensitive points-to analysis with heap cloning practical for the real world". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 2007, pp. 278–289. url: `http://doi.acm.org/10.1145/1250734.1250766` (cit. on pp. 97, 102).

[LF98] Lefebvre, Vincent and Feautrier, Paul. "Automatic Storage Management for Parallel Programs". In: *Parallel Computing* 24.3-4 (1998), pp. 649–671. url: `https://doi.org/10.1016/S0167-8191(98)00029-5` (cit. on pp. 146, 173).

[LUH18] Lehner, Wolfgang, Ungethüm, Annett, and Habich, Dirk. "Diversity of Processing Units - An Attempt to Classify the Plethora of Modern Processing Units". In: *Datenbank-Spektrum* 18.1 (2018), pp. 57–62. url: `https://doi.org/10.1007/s13222-018-0276-y` (cit. on p. 2).

[Len93] Lengauer, Christian. "Loop Parallelization in the Polytope Model". In: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*. 1993, pp. 398–416. url: `https://doi.org/10.1007/3-540-57208-2_28` (cit. on pp. 8, 21, 105, 146).

[LCL99] Lim, Amy W., Cheong, Gerald I., and Lam, Monica S. "An affine partitioning algorithm to maximize parallelism and minimize communication". In: *Proceedings of the 13th international conference on Supercomputing, ICS 1999, Rhodes, Greece, June 20-25, 1999*. 1999, pp. 228–237. url: `http://doi.acm.org/10.1145/305138.305197` (cit. on p. 146).

[LGS09] Liu, Yanhong A., Gorbovitski, Michael, and Stoller, Scott D. "A language and framework for invariant-driven transformations". In: *Generative Programming and Component Engineering, 8th International Conference, GPCE 2009, Denver, Colorado, USA, October 4-5, 2009, Proceedings*. 2009, pp. 55–64. url: `http://doi.acm.org/10.1145/1621607.1621617` (cit. on p. 103).

[Lon+14] Long, Fan, Sidiroglou-Douskos, Stelios, Kim, Deokhwan, and Rinard, Martin C. "Sound input filter generation for integer overflow errors". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 2014, pp. 439–452. url: `http://doi.acm.org/10.1145/2535838.2535888` (cit. on p. 105).

[LM69] Lowry, Edward S. and Medlock, C. W. "Object code optimization". In: *Commun. ACM* 12.1 (1969), pp. 13–22. url: `http://doi.acm.org/10.1145/362835.362838` (cit. on p. 204).

[Luc+14] Lucas, Robert, Ang, James, Bergman, Keren, Borkar, Shekhar, Carlson, William, Carrington, Laura, Chiu, George, Colwell, Robert, Dally, William, Dongarra, Jack, et al. *DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges*. Tech. rep. US Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, 2014 (cit. on p. 1).

[MHT06] Matsuzaki, Kiminori, Hu, Zhenjiang, and Takeichi, Masato. "Towards automatic parallelization of tree reductions in dynamic programming". In: *SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Cambridge, Massachusetts, USA, July 30 - August 2, 2006*. 2006, pp. 39–48. url: `http://doi.acm.org/10.1145/1148109.1148116` (cit. on p. 143).

[MF18] Matz, Alexander and Fröning, Holger. "Enabling Automatic Partitioning of Data-Parallel Kernels with Polyhedral Compilation". In: *LLVM Performance Workshop at CGO*. 2018. url: `https://llvm.org/devmtg/2018-02-24/slides/matz_2018_CGO_LLVM_perf.pdf` (cit. on pp. 2, 9).

[MAL93] Maydan, Dror E., Amarasinghe, Saman P., and Lam, Monica S. "Array Data-Flow Analysis and its Use in Array Privatization". In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, SC, USA, January 1993*. 1993, pp. 2–15. url: `http://doi.acm.org/10.1145/158511.158515` (cit. on pp. 57, 136).

[MY15] Mehta, Sanyam and Yew, Pen-Chung. "Improving compiler scalability: optimizing large programs at small price". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 2015, pp. 143–152. url: `http://doi.acm.org/10.1145/2737924.2737954` (cit. on pp. 8, 23, 176, 208).

[MY16] Mehta, Sanyam and Yew, Pen-Chung. "Variable Liberalization". In: *TACO* 13.3 (2016), 23:1–23:25. url: `http://doi.acm.org/10.1145/2963101` (cit. on p. 173).

[Mid12] Midkiff, Samuel P. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2012. url: `https://doi.org/10.2200/S00340ED1V01Y201201CAC019` (cit. on p. 125).

[Mik+14] Mikushin, Dmitry, Likhogrud, Nikolay, Zhang, Eddy Z., and Bergstrom, Christopher. "KernelGen - The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs". In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*. 2014, pp. 1011–1020. doi: `10.1109/IPDPSW.2014.115`. url: `https://doi.org/10.1109/IPDPSW.2014.115` (cit. on p. 9).

[MDH16] Moll, Simon, Doerfert, Johannes, and Hack, Sebastian. "Input Space Splitting for OpenCL". In: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. 2016, pp. 251–260. url: `http://doi.acm.org/10.1145/2892208.2892217` (cit. on pp. 6, 8, 9, 14, 24, 51, 56, 57, 77, 93, 120, 123, 146, 172, 176, 208).

[MR79] Morel, Etienne and Renvoise, Claude. "Global Optimization by Suppression of Partial Redundancies". In: *Commun. ACM* 22.2 (1979), pp. 96–103. url: `http://doi.acm.org/10.1145/359060.359069` (cit. on pp. 79, 87).

[MVB15] Mullapudi, Ravi Teja, Vasista, Vinay, and Bondhugula, Uday. "PolyMage: Automatic Optimization for Image Processing Pipelines". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 2015, pp. 429–443. url: http://doi.acm.org/10.1145/2694344.2694364 (cit. on pp. 56, 121, 122, 162, 164, 170, 174, 184, 191).

[Mul+16] Mullapudi, Ravi Teja, Adams, Andrew, Sharlet, Dillon, Ragan-Kelley, Jonathan, and Fatahalian, Kayvon. "Automatically scheduling halide image processing pipelines". In: *ACM Trans. Graph.* 35.4 (2016), 83:1–83:11. url: http://doi.acm.org/10.1145/2897824.2925952 (cit. on pp. 173, 174).

[MVB17] Mullapudi, Ravi Teja, Vasista, Vinay, and Bondhugula, Uday. *PolyMage benchmarks*. http://mcl.csa.iisc.ac.in/polymage.html. 2017 (cit. on pp. 162, 164, 170).

[MS04] Müller-Olm, Markus and Seidl, Helmut. "Precise interprocedural analysis through linear algebra". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. 2004, pp. 330–341. url: http://doi.acm.org/10.1145/964001.964029 (cit. on p. 106).

[MWD00] Muth, Robert, Watterson, Scott A., and Debray, Saumya K. "Code Specialization Based on Value Profiles". In: *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings*. 2000, pp. 340–359. url: https://doi.org/10.1007/978-3-540-45099-3_18 (cit. on p. 91).

[Nie+09] Niedzielski, David, Ronne, Jeffery von, Gampe, Andreas, and Psarris, Kleanthis. "A Verifiable, Control Flow Aware Constraint Analyzer for Bounds Check Elimination". In: *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*. 2009, pp. 137–153. url: https://doi.org/10.1007/978-3-642-03237-0_11 (cit. on pp. 77, 88, 91).

[Nou+17] Nourian, Marziyeh, Wang, Xiang, Yu, Xiaodong, Feng, Wu-chun, and Becchi, Michela. "Demystifying automata processing: GPUs, FPGAs or Micron's AP?" In: *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*. 2017, 1:1–1:11. url: http://doi.acm.org/10.1145/3079079.3079100 (cit. on p. 2).

[NC12] Nugteren, Cedric and Corporaal, Henk. "Introducing 'Bones': a parallelizing source-to-source compiler based on algorithmic skeletons". In: *The 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, London, United Kingdom, March 3, 2012*. 2012, pp. 1–10. url: http://doi.acm.org/10.1145/2159430.2159431 (cit. on p. 94).

[NZ08] Nuzman, Dorit and Zaks, Ayal. "Outer-loop vectorization: revisited for short SIMD architectures". In: *17th International Conference on Parallel Architecture and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*. 2008, pp. 2–11. url: http://doi.acm.org/10.1145/1454115.1454119 (cit. on p. 30).

[OH05] Odaira, Rei and Hiraki, Kei. "Sentinel PRE: Hoisting beyond Exception Dependency with Dynamic Deoptimization". In: *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA*. 2005, pp. 328–338. url: https://doi.org/10.1109/CGO.2005.32 (cit. on p. 87).

[Oh+13] Oh, Taewook, Kim, Hanjun, Johnson, Nick P., Lee, Jae W., and August, David I. "Practical automatic loop specialization". In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*. 2013, pp. 419–430. url: http://doi.acm.org/10.1145/2451116.2451161 (cit. on p. 91).

[Pan+15] Pananilath, Irshad, Acharya, Aravind, Vasista, Vinay, and Bondhugula, Uday. "An Optimizing Code Generator for a Class of Lattice-Boltzmann Computations". In: *TACO* 12.2 (2015), 14:1–14:23. url: http://doi.acm.org/10.1145/2739047 (cit. on p. 22).

[PP91] Pinter, Shlomit S. and Pinter, Ron Y. "Program Optimization and Parallelization Using Idioms". In: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*. 1991, pp. 79–92. url: http://doi.acm.org/10.1145/99583.99597 (cit. on pp. 125, 142, 143).

[Pom+14] Pomonis, Marios, Petsios, Theofilos, Jee, Kangkook, Polychronakis, Michalis, and Keromytis, Angelos D. "IntFlow: improving the accuracy of arithmetic error detection using information flow tracking". In: *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*. 2014, pp. 416–425. url: http://doi.acm.org/10.1145/2664243.2664282 (cit. on p. 112).

[PC10] Pop, Antoniu and Cohen, Albert. "Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers". In: *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*. Vienna, Austria, 2010. url: https://hal.inria.fr/inria-00551518 (cit. on pp. 56, 62).

[PCS05] Pop, Sebastian, Cohen, Albert, and Silber, Georges-André. "Induction Variable Analysis with Delayed Abstractions". In: *High Performance Embedded Architectures and Compilers, First International Conference, HiPEAC 2005, Barcelona, Spain, November 17-18, 2005, Proceedings*. 2005, pp. 218–232. url: https://doi.org/10.1007/11587514_15 (cit. on pp. 17, 50, 57, 110, 124, 186).

[Pop+06] Pop, Sebastian, Cohen, Albert, Bastoul, Cédric, Girbal, Sylvain, Silber, Georges-andré, and Vasilache, Nicolas. "GRAPHITE: Polyhedral Analyses and Optimizations for GCC". In: *In Proceedings of the 2006 GCC Developers Summit*. 2006, p. 2006 (cit. on pp. 3, 8, 15, 21, 49, 65, 208).

[PE95] Pottenger, William M. and Eigenmann, Rudolf. "Idiom Recognition in the Polaris Parallelizing Compiler". In: *Proceedings of the 9th international conference on Supercomputing, ICS 1995, Barcelona, Spain, July 3-7, 1995*. 1995, pp. 444–448. url: http://doi.acm.org/10.1145/224538.224655 (cit. on pp. 125, 126, 142–145).

[Pou10] Pouchet, Louis-Noël. "Interactive Optimization in the Polyhedral Model". PhD thesis. Orsay, France: University of Paris-Sud 11, 2010 (cit. on p. 14).

[Pra+11] Prabhu, Prakash, Jablin, Thomas B., Raman, Arun, Zhang, Yun, Huang, Jialu, Kim, Hanjun, Johnson, Nick P., Liu, Feng, Ghosh, Soumyadeep, Beard, Stephen R., Oh, Taewook, Zoufaly, Matthew, Walker, David, and August, David I. "A survey of the practice of computational science". In: *Conference on High Performance Computing Networking, Storage and Analysis - State of the Practice Reports, SC 2011, Seattle, Washington, USA, November 12-18, 2011*. 2011, 19:1–19:12. url: http://doi.acm.org/10.1145/2063348.2063374 (cit. on p. 2).

[PCL11] Pradelle, Benoît, Clauss, Philippe, and Loechner, Vincent. "Adaptive runtime selection of parallel schedules in the polytope model". In: *2011 Spring Simulation Multi-conference, SpringSim '11, Boston, MA, USA, April 03-07, 2011. Volume 6: Proceedings of the 19th High Performance Computing Symposia (HPC)*. 2011, pp. 81–88. url: http://dl.acm.org/citation.cfm?id=2048588 (cit. on pp. 49, 77).

[Pra11] Pradelle, Benoît. "Static and Dynamic Methods of Polyhedral Compilation for an Efficient Execution in Multicore Environments. (Méthodes Statiques et Dynamiques de Compilation Polyédrique pour l'Exécution en Environnement Multi-Cœurs)". PhD thesis. University of Strasbourg, France, 2011. url: https://tel.archives-ouvertes.fr/tel-00733856 (cit. on pp. 62, 91).

[PKC12] Pradelle, Benoît, Ketterlin, Alain, and Clauss, Philippe. "Polyhedral parallelization of binary code". In: *TACO* 8.4 (2012), 39:1–39:21. url: http://doi.acm.org/10.1145/2086696.2086718 (cit. on p. 23).

[Pra+17] Pradelle, Benoît, Meister, Benoît, Baskaran, Muthu, Springer, Jonathan, and Lethin, Richard. "Polyhedral Optimization of TensorFlow Computation Graphs". In: *The 6th Workshop on Extreme-scale Programming Tools (ESPT-2017) at The International Conference for High Performance Computing, Networking, Storage and Analysis (SC17)*. ESPT'17. 2017 (cit. on pp. 9, 122, 191).

[Pra18] Prajapati, Nirmal. “Scheduling and Tiling Reductions on Realistic Machines”. In: *CoRR* abs/1801.05909 (2018). arXiv: 1801.05909. url: http://arxiv.org/abs/1801.05909 (cit. on p. 145).

[Pre31] Presburger, M. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*. 1931. url: https://books.google.de/books?id=7agKHQAACAAJ (cit. on pp. 8, 9, 206).

[Pre12] Preshing, Jeff. *A Look Back at Single-Threaded CPU Performance*. https://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance. 2012 (cit. on pp. 1, 2).

[Pro59] Prosser, Reese T. “Applications of Boolean Matrices to the Analysis of Flow Diagrams”. In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM ’59 (Eastern). Boston, Massachusetts: ACM, 1959, pp. 133–138. url: http://doi.acm.org/10.1145/1460299.1460314 (cit. on p. 204).

[Pug91a] Pugh, William. “The Omega test: a fast and practical integer programming algorithm for dependence analysis”. In: *Proceedings Supercomputing ’91, Albuquerque, NM, USA, November 18-22, 1991*. 1991, pp. 4–13. url: http://doi.acm.org/10.1145/125826.125848 (cit. on pp. 8, 105, 106).

[Pug91b] Pugh, William. “Uniform techniques for loop optimization”. In: *Proceedings of the 5th international conference on Supercomputing, ICS 1991, Cologne, Germany, June 17-21, 1991*. 1991, pp. 341–352. url: http://doi.acm.org/10.1145/109025.109108 (cit. on p. 139).

[PW93] Pugh, William and Wonnacott, David. “An Exact Method for Analysis of Value-based Array Data Dependences”. In: *Languages and Compilers for Parallel Computing, 6th International Workshop, Portland, Oregon, USA, August 12-14, 1993, Proceedings*. 1993, pp. 546–566. url: https://doi.org/10.1007/3-540-57659-2_31 (cit. on pp. 57, 136).

[PW94] Pugh, William and Wonnacott, David. “Static Analysis of Upper and Lower Bounds on Dependences and Parallelism”. In: *ACM Trans. Program. Lang. Syst.* 16.4 (1994), pp. 1248–1278. url: http://doi.acm.org/10.1145/183432.183525 (cit. on pp. 125, 128, 136, 139, 143).

[QHV02] Qian, Feng, Hendren, Laurie J., and Verbrugge, Clark. “A Comprehensive Approach to Array Bounds Check Elimination for Java”. In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. 2002, pp. 325–342. url: https://doi.org/10.1007/3-540-45937-5_23 (cit. on pp. 77, 88, 91).

[QRW00] Quilleré, Fabien, Rajopadhye, Sanjay V., and Wilde, Doran. “Generation of Efficient Nested Loops from Polyhedra”. In: *International Journal of Parallel Programming* 28.5 (2000), pp. 469–498. url: https://doi.org/10.1023/A:1007554627716 (cit. on p. 14).

[QR00] Quilleré, Fabien and Rajopadhye, Sanjay V. “Optimizing memory usage in the polyhedral model”. In: *ACM Trans. Program. Lang. Syst.* 22.5 (2000), pp. 773–815. url: http://doi.acm.org/10.1145/365151.365152 (cit. on pp. 9, 23, 146).

[Qui84] Quinton, Patrice. “Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations”. In: *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*. 1984, pp. 208–214. url: http://doi.acm.org/10.1145/800015.808184 (cit. on p. 207).

[Rac16] Rachid, Wiam. “Polyhedral Task Parallelization”. Bachelor Thesis. Saarland University, 2016 (cit. on p. 173).

[Rag+12] Ragan-Kelley, Jonathan, Adams, Andrew, Paris, Sylvain, Levoy, Marc, Amarasinghe, Saman P., and Durand, Frédo. “Decoupling algorithms from schedules for easy optimization of image processing pipelines”. In: *ACM Trans. Graph.* 31.4 (2012), 32:1–32:12. url: http://doi.acm.org/10.1145/2185520.2185528 (cit. on pp. 121, 173).

[Rag+13] Ragan-Kelley, Jonathan, Barnes, Connelly, Adams, Andrew, Paris, Sylvain, Durand, Frédo, and Amarasinghe, Saman P. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 519–530. url: http://doi.acm.org/10.1145/2462156.2462176 (cit. on pp. 121, 164, 172, 174).

[Rag11] Raghesh, A. "A Framework for Automatic OpenMP Code Generation". master thesis. 2011 (cit. on pp. 15, 161).

[Ram94] Ramalingam, G. "The Undecidability of Aliasing". In: *ACM Trans. Program. Lang. Syst.* 16.5 (1994), pp. 1467–1471. url: http://doi.acm.org/10.1145/186025.186041 (cit. on p. 102).

[RP94] Rauchwerger, Lawrence and Padua, David A. "The privatizing DOALL test: a run-time technique for DOALL loop identification and array privatization". In: *Proceedings of the 8th international conference on Supercomputing, ICS 1994, Manchester, UK, July 11-15, 1994*. 1994, pp. 33–43. url: http://doi.acm.org/10.1145/181181.181254 (cit. on p. 101).

[RAP95] Rauchwerger, Lawrence, Amato, Nancy M., and Padua, David A. "A scalable method for run-time loop parallelization". In: *International Journal of Parallel Programming* 23.6 (1995), pp. 537–576. doi: 10.1007/BF02577866. url: https://doi.org/10.1007/BF02577866 (cit. on p. 125).

[RP95] Rauchwerger, Lawrence and Padua, David A. "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization". In: *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*. 1995, pp. 218–232. url: http://doi.acm.org/10.1145/207110.207148 (cit. on pp. 62, 125, 142, 143, 145).

[RKC16] Reddy, Chandan, Kruse, Michael, and Cohen, Albert. "Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU". In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*. 2016, pp. 87–97. url: http://doi.acm.org/10.1145/2967938.2967950 (cit. on p. 125).

[RF93] Redon, Xavier and Feautrier, Paul. "Detection of Recurrences in Sequential Programs with Loops". In: *PARLE '93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, Munich, Germany, June 14-17, 1993, Proceedings*. 1993, pp. 132–145. url: https://doi.org/10.1007/3-540-56891-3_11 (cit. on pp. 125, 142, 143).

[RF94] Redon, Xavier and Feautrier, Paul. "Scheduling reductions". In: *Proceedings of the 8th international conference on Supercomputing, ICS 1994, Manchester, UK, July 11-15, 1994*. 1994, pp. 117–125. url: http://doi.acm.org/10.1145/181181.181319 (cit. on pp. 125, 143, 144).

[RF00] Redon, Xavier and Feautrier, Paul. "Detection of Scans in the Polytope Model". In: *Parallel Algorithms Appl.* 15.3-4 (2000), pp. 229–263. url: https://doi.org/10.1080/01495730008947357 (cit. on pp. 142, 143).

[RCP13] Rodrigues, Raphael Ernani, Campos, Victor Hugo Sperle, and Pereira, Fernando Magno Quintão. "A fast and low-overhead technique to secure programs against integer overflows". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. 2013, 33:1–33:11. url: https://doi.org/10.1109/CGO.2013.6494996 (cit. on p. 105).

[RM88] Ruggieri, Cristina and Murtagh, Thomas P. "Lifetime Analysis of Dynamically Allocated Objects". In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. 1988, pp. 285–293. url: http://doi.acm.org/10.1145/73560.73585 (cit. on p. 163).

[RR00] Rugina, Radu and Rinard, Martin C. "Symbolic bounds analysis of pointers, array indices, and accessed memory regions". In: *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. 2000, pp. 182–195. url: http://doi.acm.org/10.1145/349299.349325 (cit. on p. 57).

[RRH02] Rus, Silvius, Rauchwerger, Lawrence, and Hoeflinger, Jay. "Hybrid analysis: static & dynamic memory reference analysis". In: *Proceedings of the 16th international conference on Supercomputing, ICS 2002, New York City, NY, USA, June 22-26, 2002*. 2002, pp. 274–284. url: http://doi.acm.org/10.1145/514191.514229 (cit. on p. 62).

[RHR05] Rus, Silvius, He, Guobin, and Rauchwerger, Lawrence. "Scalable Array SSA and Array Data Flow Analysis". In: *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers*. 2005, pp. 397–412. url: https://doi.org/10.1007/978-3-540-69330-7_27 (cit. on p. 174).

[SK98] Sarkar, Vivek and Knobe, Kathleen. "Enabling Sparse Constant Propagation of Array Elements via Array SSA Form". In: *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings*. 1998, pp. 33–56. url: https://doi.org/10.1007/3-540-49727-7_3 (cit. on p. 174).

[SI11] Sato, Shigeyuki and Iwasaki, Hideya. "Automatic parallelization via matrix multiplication". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 2011, pp. 470–479. url: http://doi.acm.org/10.1145/1993498.1993554 (cit. on pp. 143, 144).

[SJL11] Seo, Sangmin, Jo, Gangwon, and Lee, Jaejin. "Performance characterization of the NAS Parallel Benchmarks in OpenCL". In: *Proceedings of the 2011 IEEE International Symposium on Workload Characterization, IISWC 2011, Austin, TX, USA, November 6-8, 2011*. 2011, pp. 137–148. url: https://doi.org/10.1109/IISWC.2011.6114174 (cit. on pp. 65, 74, 88, 148).

[Ser+12] Serebryany, Konstantin, Bruening, Derek, Potapenko, Alexander, and Vyukov, Dmitriy. "AddressSanitizer: A Fast Address Sanity Checker". In: *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. 2012, pp. 309–318. url: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany (cit. on p. 88).

[SH97] Shapiro, Marc and Horwitz, Susan. "Fast and Accurate Flow-Insensitive Points-To Analysis". In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*. 1997, pp. 1–14. url: http://doi.acm.org/10.1145/263699.263703 (cit. on p. 102).

[SHS17] Shirako, Jun, Hayashi, Akihiro, and Sarkar, Vivek. "Optimized two-level parallelization for GPU accelerators using the polyhedral model". In: *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*. 2017, pp. 22–33. url: http://dl.acm.org/citation.cfm?id=3033022 (cit. on pp. 9, 23).

[Sim+13] Simbürger, Andreas, Apel, Sven, Größlinger, Armin, and Lengauer, Christian. "The Potential of Polyhedral Optimization: An Empirical Study". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 2013, pp. 508–518. url: https://doi.org/10.1109/ASE.2013.6693108 (cit. on pp. 21, 22, 39, 41, 93).

[SK07] Simon, Axel and King, Andy. "Taming the Wrapping of Integer Arithmetic". In: *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*. 2007, pp. 121–136. url: https://doi.org/10.1007/978-3-540-74061-2_8 (cit. on p. 105).

[Sta84] Stansifer, Ryan. *Presburger's Article on Integer Airthmetic: Remarks and Translation*. Tech. rep. TR84-639. Cornell University, Computer Science Department, 1984. url: http://cs.fit.edu/~ryan/papers/presburger.pdf (cit. on pp. 8, 206).

[Ste96] Steensgaard, Bjarne. “Points-to Analysis in Almost Linear Time”. In: *Conference Record of POPL’96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. 1996, pp. 32–41. url: http://doi.acm.org/10.1145/237721.237727 (cit. on pp. 97, 102).

[Sto+14] Stock, Kevin, Kong, Martin, Grosser, Tobias, Pouchet, Louis-Noël, Rastello, Fabrice, Ramanujam, J., and Sadayappan, P. “A framework for enhancing data reuse via associative reordering”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 2014, pp. 65–76. url: http://doi.acm.org/10.1145/2594291.2594342 (cit. on pp. 8, 144, 176, 208).

[Str+12] Streit, Kevin, Hammacher, Clemens, Zeller, Andreas, and Hack, Sebastian. “Sambamba: A Runtime System for Online Adaptive Parallelization”. In: *Compiler Construction - 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 240–243. url: https://doi.org/10.1007/978-3-642-28652-0_13 (cit. on pp. 40, 65, 69).

[Str+15] Streit, Kevin, Doerfert, Johannes, Hammacher, Clemens, Zeller, Andreas, and Hack, Sebastian. “Generalized Task Parallelism”. In: *TACO* 12.1 (2015), 8:1–8:25. url: http://doi.acm.org/10.1145/2723164 (cit. on pp. 6, 24, 65, 69, 125, 142).

[Str17] Streit, Kevin. “Runtime-adaptive generalized task parallelism”. PhD thesis. Saarland University, Saarbrücken, Germany, 2017. url: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/26872 (cit. on pp. 2, 125, 142).

[Str+98] Strout, Michelle Mills, Carter, Larry, Ferrante, Jeanne, and Simon, Beth. “Schedule-Independent Storage Mapping for Loops”. In: *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998*. 1998, pp. 24–33. url: http://doi.acm.org/10.1145/291069.291015 (cit. on p. 146).

[Sud+14] Suda, Akihiro, Takase, Hideki, Takagi, Kazuyoshi, and Takagi, Naofumi. “Nested Loop Parallelization Using Polyhedral Optimization in High-Level Synthesis”. In: *IEICE Transactions* 97-A.12 (2014), pp. 2498–2506. url: http://search.ieice.org/bin/summary.php?id=e97-a_12_2498 (cit. on p. 23).

[SKN96] Suganuma, Toshio, Komatsu, Hideaki, and Nakatani, Toshio. “Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines”. In: *Proceedings of the 10th international conference on Supercomputing, ICS 1996, Philadelphia, PA, USA, May 25-28, 1996*. 1996, pp. 18–25. url: http://doi.acm.org/10.1145/237578.237581 (cit. on pp. 125, 142, 143, 145).

[Suk+14] Sukumaran-Rajam, Aravind, Caamaño, Juan Manuel Martinez, Wolff, Willy, Jimborean, Alexandra, and Clauss, Philippe. “Speculative Program Parallelization with Scalable and Decentralized Runtime Verification”. In: *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*. 2014, pp. 124–139. url: https://doi.org/10.1007/978-3-319-11164-3_11 (cit. on pp. 9, 62, 91).

[SC16] Sukumaran-Rajam, Aravind and Clauss, Philippe. “The Polyhedral Model of Nonlinear Loops”. In: *TACO* 12.4 (2016), 48:1–48:27. url: http://doi.acm.org/10.1145/2838734 (cit. on pp. 9, 62).

[Sun+15] Sun, Hao, Su, Chao, Wang, Yue, and Zeng, Qingkai. “Improving the Accuracy of Integer Signedness Error Detection Using Data Flow Analysis”. In: *The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015*. 2015, pp. 601–606. url: https://doi.org/10.18293/SEKE2015-123 (cit. on p. 112).

[SKF18] Süß, Tim, Kaya, Tunahan, and Feld, Dustin. “Extending PluTo for Multiple Devices by Integrating OpenACC”. In: *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*. 2018, pp. 288–291. url: https://doi.org/10.1109/PDP2018.2018.00049 (cit. on p. 9).

[Sut05] Sutter, Herb. “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”. In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210 (cit. on p. 1).

[TIF86] Triolet, Rémi, Irigoin, François, and Feautrier, Paul. “Direct parallelization of call statements”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*. 1986, pp. 176–185. url: http://doi.acm.org/10.1145/12276.13329 (cit. on p. 63).

[Upa13] Upadrasta, Ramakrishna. “Sub-Polyhedral Compilation using (Unit-)Two-Variables-Per-Inequality Polyhedra. (Compilation sous-polyédrique reposant sur des systèmes à deux variables par inégalité)”. PhD thesis. University of Paris-Sud, Orsay, France, 2013. url: https://tel.archives-ouvertes.fr/tel-00818764 (cit. on pp. 3, 8).

[VCC03] Vanbroekhoven, Peter, Corporaal, Henk, and Catthoor, Francky. “Advanced copy propagation for arrays”. In: *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03). San Diego, California, USA, June 11-13, 2003*. 2003, pp. 24–33. url: http://doi.acm.org/10.1145/780732.780736 (cit. on pp. 172–174).

[Van+07] Vanbroekhoven, Peter, Janssens, Gerda, Bruynooghe, Maurice, and Catthoor, Francky. “A practical dynamic single assignment transformation”. In: *ACM Trans. Design Autom. Electr. Syst.* 12.4 (2007), p. 40. url: http://doi.acm.org/10.1145/1278349.1278353 (cit. on pp. 173, 205).

[Vas+12] Vasilache, Nicolas, Meister, Benoit, Baskaran, Muthu, and Lethin, Richard. “Joint Scheduling and Layout Optimization to Enable Multi-Level Vectorization”. In: *IMPACT 2012* (2012) (cit. on p. 9).

[Vas+18] Vasilache, Nicolas, Zinenko, Oleksandr, Theodoridis, Theodoros, Goyal, Priya, DeVito, Zachary, Moses, William S., Verdoolaege, Sven, Adams, Andrew, and Cohen, Albert. “Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions”. In: *CoRR* abs/1802.04730 (2018). arXiv: 1802.04730. url: http://arxiv.org/abs/1802.04730 (cit. on pp. 3, 122, 191).

[Ven+14] Venkat, Anand, Shantharam, Manu, Hall, Mary W., and Strout, Michelle Mills. “Non-affine Extensions to Polyhedral Code Generation”. In: *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*. 2014, p. 185. url: http://doi.acm.org/10.1145/2544137.2544141 (cit. on pp. 22, 143, 144).

[Ver10] Verdoolaege, Sven. “*isl*: An Integer Set Library for the Polyhedral Model”. In: *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*. 2010, pp. 299–302. url: https://doi.org/10.1007/978-3-642-15582-6_49 (cit. on pp. 10, 15, 42, 57, 70, 71, 98, 105, 106).

[VG12] Verdoolaege, Sven and Grosser, Tobias. “Polyhedral extraction tool”. In: *International Workshop on Polyhedral Compilation Techniques*. IMPACT'12. Paris, France, 2012 (cit. on pp. 8, 49, 51, 53, 106, 107, 124).

[Ver+13] Verdoolaege, Sven, Juega, Juan Carlos, Cohen, Albert, Gómez, José Ignacio, Tenllado, Christian, and Catthoor, Francky. “Polyhedral parallel code generation for CUDA”. In: *TACO* 9.4 (2013), 54:1–54:23. url: http://doi.acm.org/10.1145/2400682.2400713 (cit. on pp. 9, 65, 93).

[Ver+14] Verdoolaege, Sven, Guelton, Serge, Grosser, Tobias, and Cohen, Albert. “Schedule Trees”. In: *International Workshop on Polyhedral Compilation Techniques*. IMPACT'14. 2014 (cit. on p. 35).

[Ver15a] Verdoolaege, Sven. “Integer set coalescing”. In: IMPACT'15. 2015 (cit. on p. 70).

[Ver15b] Verdoolaege, Sven. *PENCIL support in pet and PPCG*. Technical Report RT-0457. INRIA Paris-Rocquencourt ; INRIA, 2015. url: https://hal.inria.fr/hal-01133962 (cit. on pp. 173, 208).

[VC16] Verdoolaege, Sven and Cohen, Albert. "Live Range Reordering". In: *6th Workshop on Polyhedral Compilation Techniques (IMPACT, associated with HiPEAC)*. Prag, Czech Republic, 2016. url: https://hal.archives-ouvertes.fr/hal-01257224 (cit. on pp. 30, 31).

[Ver16] Verdoolaege, Sven. *Presburger Formulas and Polyhedral Compilation*. eng. Tech. rep. 2016. url: https://lirias.kuleuven.be/retrieve/361209 (cit. on pp. 9, 11, 178).

[VOE11] Villars, Richard L., Olofson, Carl W., and Eastwood, Matthew. "Big Data: What It Is and Why You Should Care". In: *White Paper, International Data Corporation (IDC)* 14 (2011), pp. 1–14 (cit. on p. 1).

[WK00] Walter, Joerg and Koch, Mathias. *uBLAS: The Boost Basic Linear Algebra Library*. https://www.boost.org/doc/libs/1_67_0/libs/numeric/ublas/doc/index.html. 2000 (cit. on p. 88).

[Wan+09] Wang, Tielei, Wei, Tao, Lin, Zhiqiang, and Zou, Wei. "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. 2009. url: http://www.isoc.org/isoc/conferences/ndss/09/pdf/17.pdf (cit. on p. 105).

[WR96] Wilde, Doran and Rajopadhye, Sanjay V. "Memory Reuse Analysis in the Polyhedral Model". In: *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume I*. 1996, pp. 389–397. url: https://doi.org/10.1007/3-540-61626-8_51 (cit. on pp. 146, 173).

[Won99] Wonnacott, David. *Constant Propagation Through Array Variables*. 1999 (cit. on p. 174).

[Won00] Wonnacott, David. "Extending Scalar Optimizations for Arrays". In: *Languages and Compilers for Parallel Computing, 13th International Workshop, LCPC 2000, Yorktown Heights, NY, USA, August 10-12, 2000, Revised Papers*. 2000, pp. 97–111. url: https://doi.org/10.1007/3-540-45574-4_7 (cit. on pp. 172, 174).

[Wu+13] Wu, Jingyue, Hu, Gang, Tang, Yang, and Yang, Junfeng. "Effective dynamic detection of alias analysis errors". In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 2013, pp. 279–289. url: http://doi.acm.org/10.1145/2491411.2491439 (cit. on p. 103).

[WWM07] Würthinger, Thomas, Wimmer, Christian, and Mössenböck, Hanspeter. "Array bounds check elimination for the Java HotSpot&trade; client compiler". In: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, PPPJ 2007, Lisboa, Portugal, September 5-7, 2007*. 2007, pp. 125–133. url: http://doi.acm.org/10.1145/1294325.1294343 (cit. on pp. 77, 88, 91).

[XKH04] Xu, Dana N., Khoo, Siau-Cheng, and Hu, Zhenjiang. "PType System: A Featherweight Parallelizability Detector". In: *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*. 2004, pp. 197–212. doi: 10.1007/978-3-540-30477-7_14. url: https://doi.org/10.1007/978-3-540-30477-7%5C_14 (cit. on pp. 125, 143, 144).

[XYR12] Xu, Guoqing (Harry), Yan, Dacong, and Rountev, Atanas. "Static Detection of Loop-Invariant Data Structures". In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. 2012, pp. 738–763. url: https://doi.org/10.1007/978-3-642-31057-7_32 (cit. on p. 87).

[YR06] Yu, Hao and Rauchwerger, Lawrence. "An Adaptive Algorithm Selection Framework for Reduction Parallelization". In: *IEEE Trans. Parallel Distrib. Syst.* 17.10 (2006), pp. 1084–1096. doi: 10.1109/TPDS.2006.131. url: https://doi.org/10.1109/TPDS.2006.131 (cit. on pp. 125, 126, 145).

[Yuk+10] Yuki, Tomofumi, Renganarayanan, Lakshminarayanan, Rajopadhye, Sanjay V., Anderson, Charles, Eichenberger, Alexandre E., and O'Brien, Kevin. "Automatic creation of tile size selection models". In: *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*. 2010, pp. 190–199. url: http://doi.acm.org/10.1145/1772954.1772982 (cit. on p. 162).

[YR13] Yuki, Tomofumi and Rajopadhye, Sanjay. "Memory Allocations for Tiled Uniform Dependence Programs". In: *IMPACT 2013* (2013), p. 13 (cit. on p. 9).

[Zha+15] Zhang, Yang, Sun, Xiaoshan, Deng, Yi, Cheng, Liang, Zeng, Shuke, Fu, Yu, and Feng, Dengguo. "Improving Accuracy of Static Integer Overflow Detection in Binary". In: *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*. 2015, pp. 247–269. url: https://doi.org/10.1007/978-3-319-26362-5_12 (cit. on p. 105).

[ZKC18] Zhao, Jie, Kruse, Michael, and Cohen, Albert. "A polyhedral compilation framework for loops with dynamic data-dependent bounds". In: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. 2018, pp. 14–24. url: http://doi.acm.org/10.1145/3178372.3179509 (cit. on pp. 9, 30, 51, 56, 62, 93).

[ZHB18] Zinenko, Oleksandr, Huot, Stéphane, and Bastoul, Cédric. "Visual Program Manipulation in the Polyhedral Model". In: *TACO* 15.1 (2018), 16:1–16:25. url: http://doi.acm.org/10.1145/3177961 (cit. on p. 9).

[ZR12] Zou, Yun and Rajopadhye, Sanjay V. "Scan detection and parallelization in "inherently sequential" nested loop programs". In: *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*. 2012, pp. 74–83. url: http://doi.acm.org/10.1145/2259016.2259027 (cit. on p. 143).

[Zuo+13] Zuo, Wei, Li, Peng, Chen, Deming, Pouchet, Louis-Noël, Zhong, Shunan, and Cong, Jason. "Improving polyhedral code generation for high-level synthesis". In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013, Montreal, QC, Canada, September 29 - October 4, 2013*. 2013, 15:1–15:10. url: https://doi.org/10.1109/CODES-ISSS.2013.6659002 (cit. on pp. 22, 23).