SPECULATIVE LOOP PARALLELIZATION

Johannes Doerfert

Bachelor Thesis

Compiler Design Lab
Faculty of Natural Sciences and Technology I
Department of Computer Science
Saarland University

Supervisor: Prof. Dr. Sebastian Hack

Advisors: Clemens Hammacher and Kevin Streit

Reviewers: Prof. Dr. Sebastion Hack
Prof. Dr. Andreas Zeller

Submitted: April 2012

UNIVERSITÄT
DES
SAARLANDES

Declaration of Authorship

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Datum/Date: ______________________________

Unterschrift/Signature: ______________________________

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Brian Kernighan, professor at Princeton University

SAARLAND UNIVERSITY

Abstract

Compiler Design Lab
Department of Computer Science

Bachelor of Science

by Johannes Doerfert

SPolly, short for speculative Polly, is an attempt to combine two recent research projects in the context of compilers. There is Polly, an LLVM project to increase data-locality and parallelism of regular loop nests and Sambamba, which pursues a new, adaptive way of compiling and offers features like method versioning, speculation and runtime adaption. As an extension of the former one and with the capabilities offered by the later one, SPolly can perform state-of-the-art loop optimizations on a wide range of loops, even in general purpose benchmarks as the SPEC 2000 benchmark suite. It is also capable of detection promising loops for parallel execution even if they contain irreversibel function calls or non computable dependencies. This thesis will explain under what circumstances speculation is possible, how static and dynamic information are used to minimize the amount of misspeculation and how this is integrated into the existing environment of Polly and Sambamba. To substantiate the improvements of this work an evaluation on SPEC 2000 benchmarks and a case study on different versions of the matrix multiplication benchmark is presented at the end.

Acknowledgements

Although I have no “Loving Wife” I could thank for keeping the children away there were many friends which helped me to bear the brunt of the work. Especially during the last, indeed frantic days they did their utmost. When the only concern is the verging deadline you may consider yourself lucky if you can pass on all other matters. As the daily living is important but not everything when you write your first academic contribution I have to extend special thanks to my advisors. Both of them did a heck of a job and endured all my moods and quirks. Not only that they were always accessible for questions but they also encouraged me to structure my work, something I am admittedly not that good at. From my present view I know it was the best supervision possible. I also owe special thanks to my supervising professor for the freedom of this thesis and the content.

Content

Abbreviations

AA	**A**lias **A**nalysis
CFG	**C**ontrol **F**low **G**raph
cloog	**C**hunky **Loo**p **G**enerator
isl	**i**nteger **s**et **l**ibrary
LOC	**l**ines **o**f **c**ode
LLVM	**L**ow **L**evel **V**irtual **M**achine
LLVM-IR	LLVM **I**ntermediate **R**epresentation
OpenMP	**Open** **M**ulti-**P**rocessing
Polly	**Poly**hedral **LL**VM
SCoP	**S**tatic **Co**ntrol **P**art
sSCoP	**s**peculative **S**tatic **Co**ntrol **P**art
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SPolly	**S**peculative **P**olly
ParCFG	**Par**allel **CFG**

Dedicated to my father whose never ending patience is priceless.

Chapter 1

Introduction

Nowadays multiprocessors became ubiquitous even in the area of personal and mobile computing; however, automatic parallelization did not. Programmers still write sequential code which will be translated to sequential binaries and executed by a single thread using only one of many cores. Benefits of modern multiprocessors are still unused because neither programmers nor compilers may utilize their potential to the full. Even if legacy applications would be amenable to parallelization, it is unclear how to find and exploit theirs potential automatically. Apart from the retrieval, parallelism faces the same problems as sequential code does. Cache invalidation and subsequently cache misses caused by poor data-locality is a well known one. Heavy research is going on to improve parallelism as well as data-locality but the results may differ. As there are promising approaches suffering from poor applicability on general purpose code, the real problem becomes more and more applying optimizations, not developing them.

Techniques using the so called polyhedral model become increasingly popular. The underlying model is a mathematical description of loop nests with their data dependencies. Optimal solutions in terms of e.g., locality or parallelism can be derived using this model while it implicitly applies traditional optimizations as loop blocking and unrolling. Various preliminary results reveal the potential but also the limits of this technique. Enormous speedups are possible, but only for very restricted and therefor few locations.

1.1 Related Work

Research on parallelism and data locality is very popular nowadays, as is the polytope model to tackle these problems. With or without speculation, there are promising attempts all using the polytope model, but the wide range impact on general purpose code is still missing.

Tobias Grosser describes in his thesis[2] a speedup of up to 8 for the matrix multiplication benchmark, archived by his polyhedral optimizer Polly[3]. He also produced similar results for other benchmarks of the Polybench[4] benchmark suite. Other publications on this topic[5–7] show similar results, but they are also limited to the Polybench benchmark suite. Admittedly, Polybench is well suited for comparative studies, between these approaches, but it has less significance for general applicability. Baghdadi et. al.[6] revealed a huge potential for speculative loop optimizations. They state that aggressive loop nest optimizations (including parallel execution) are profitable and possible, even though data and flow dependencies would statically prevent them. Theirs hand made tests also showed the impact when different kinds of conflict management are used. The results differ from loop to loop as the availability of such conflict management systems does. But even if the choice is restricted to one or two, their results indicate a general speedup.

1.2 Overview

SPolly, short for speculative Polly, is an attempt to combine two recent research projects in the context of compilers. There is Polly, an LLVM project to increase data-locality and parallelism of regular loop nests and Sambamba, which pursues a new, adaptive way of compiling and offers features like method versioning, speculation and runtime adaption. As an extension of the former one and with the capabilities offered by the later one, SPolly can perform state-of-the-art loop optimizations on a wide range of loops, even in general purpose benchmarks as the SPEC 2000 benchmark suite. It is also capable of detection promising loops for parallel execution even if they contain irreversibel function calls or non computable dependencies. This thesis will explain under what circumstances speculation is possible, how static and dynamic information are used to minimize the amount of misspeculation and how this is integrated into the existing environment of Polly and Sambamba. To substantiate the improvements of this work an evaluation on SPEC 2000 benchmarks and a case study on different versions of the matrix multiplication benchmark is presented at the end.

The rest of the thesis will be organised as follows. First Chapter 2 will provide information on the used tools and techniques, especially Polly and Sambamba. Afterwards the concept as well as the key ideas are stated in Chapter 3. Technical details about SPolly are given in Chapter 4, followed by an evaluation on the SPEC 2000 and Polybench 3.2 benchmark suites (Chapter 5). While Chapter 7 concludes the thesis and provides ideas for future work, a detailed case study on different versions of the matrix multiplication example is presented in Chapter 6.

Note

For simplicity source code is presented in a C like language only.

Chapter 2

Background

This work makes heavy use of different techniques and tools mostly in the context of compiler construction. To simplify the rest of the thesis this chapter explains them as far as necessary, thus we may take them for granted afterwards. As most of the key ideas will suffice, some details will be omitted. Interested readers have to fall back on the further readings instead.

2.1 LLVM - The LLVM Compiler Infrastructure

The Low Level Virtual Machine is a compiler infrastructure designed to optimize during compile time, link time and runtime. Originally designed for C and C++, many other frontends for a variety of languages exist by now. The source is translated into an intermediate representation (LLVM-IR), which is available in three different, but equivalent forms. There is the in-memory compiler IR, the on-disk bitcode representation and human readable assembly language. The LLVM-IR is a type-safe, static single assignment based language, designed for low-level operations. It is capable of representing high-level structures in a flexible way. Due to the fact that LLVM is built in a modular way and can be extended easily, most of the state of the art analysis and optimization techniques are implemented and shipped with LLVM. Plenty of other extensions, e.g., Polly, can be added by hand.

Further Reading

- A Compilation Framework for Lifelong Program Analysis & Transformation [8]
- `http://www.llvm.org`

2.2 The Polyhedral Model

The polyhedral model is a mathematical way to describe the iteration space of loop nests, or more specifically a very restricted subset of them. It became popular as it abstracts from the given source and applies loop optimizations in a pure mathematical way. Formulating optimization problems in terms of linear equations yields an optimal solution with regard to a given property, e.g., data-locality or parallelism.

Within the polyhedral model the iteration space is represented as a $\mathbb{Z}$-polyhedron, simply spoken a geometric object with flat surfaces existing in a space of any dimensionality. To do so, it is necessary that the iteration space can be described as solution set of affine inequalities (equation 2.1), where the loop bounds are encoded in a translation (b) and the steppings in a matrix (A).

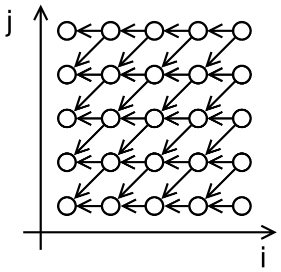
$$\text{Iteration Space IS } := \{x \in \mathbb{Z}^n \mid Ax \leq b \text{ with } A \in \mathbb{Z}^{m*n},\, b \in \mathbb{Z}^m\ \} \tag{2.1}$$

In addition to the points of the iteration space, the polyhedral model is also capable of representing loop carried dependencies between two iterations. Figure 2.1 illustrates this by relating a simple loop nest(A) to its representation in the polyhedral model(B). Note that we do not distinguish between the base pointers (here `A` and `B`) when we draw dependencies. This is feasible because we have to respect all dependencies in order to perform loop transformations.

An optimization within the model (which is in fact a composed affine transformation) could yield a loop nest as presented in listing 2.2. Data-locality has been increased by introducing the two inner most loops which also remove the data dependency from the outer most one, thus the whole loope nest may be executed in parallel now.

```
for (i = 1; i < 100; i++)
  for (j = 1; j < 100; j++)
    A[i][j] = A[i-1][j-1] * B[i-1][j];
```

(A) Simple loop nest



(B) Polyhedral representation for listing 2.1a

FIGURE 2.1: An example loop nest with its polyhedral representation

```
for (c1=-128;c1<=98;c1+=32) {
  for (c2=max(0,-32*floord(c1,32)-32);c2<=min(98,-c1+98);c2+=32) {
    for (c3=max(max(-98,c1),-c2-31);c3<=min(c1+31,-c2+98);c3++) {
      for (c4=max(c2,-c3);c4<=min(min(98,c2+31),-c3+98);c4++) {
        A[c3+c4+1][c4+1] = A[c3+c4][c4] * B[c3+c4][c4+1];
      }
    }
  }
}
```

Figure 2.2: Optimized version of listing 2.1a

Further Reading

- The Polyhedral Model is More Widely Applicable Than Yoy Think [10]
- Loop Parallelization in the Polytope Model [11]
- A practical automatic polyhedral parallelizer and locality optimizer [5]
- PoCC - The Polyhedral Compiler Collection [12]
- Polyhedral parallelization of binary code [7]
- Putting Polyhedral Loop Transformations to Work [13]

2.3 Polly - A Polyhedral Optimizer For LLVM

Exploiting parallelism and data-locality in order to balance the workload and to improve cache locality are the main goals of the Polly research project. The polyhedral model is used as abstract mathematical representation to get optimal results for a particular objective. The three-step approach of Polly first detects maximal loop nests suitable for polyhedral representation. These representations are analyzed and transformed before they are converted to code (here LLVM-IR) again. In the case of Polly the code generation is capable of generating thread level parallelism and vector instructions. The code regions Polly is interested in, are called static control parts, or short SCoPs. Such a SCoP is the most central entity within Polly and crucial for any kind of argumentation.

Definition 2.1 (Affine Transformation)**.**
Affine transformations are linear transformations followed by a translation, thus they can be written as:

$$f(x_1, \ldots, x_n) := A_1\, x_1 + \cdots + A_n\, x_n + b$$

Definition 2.2 (Canonical Induction Variable)**.**
An induction variable is canonical if it starts at zero and steps by one.

2.3.1 Static Control Parts

A static control part is a region with statically known control flow and memory accesses. As part of the control flow graph it is restricted to have one entry edge and one exit edge while loops and conditionals are allowed inside. To predict all accesses, it is necessary to restrict loop bounds and branch conditions to be affine with respect to invariant parameters and surrounding iteration variables, but only if they are canonical. Because dependency information are crucial for data flow information, SCoPs are not allowed to contain aliasing instructions at all. Furthermore only affine access functions will yield a precise result as non affine ones are overestimated (but not forbidden per se). The last point concerns called function as only side effect free functions ones are allowed here.

While the code can be transformed to fulfill some of these conditions (e.g., the canonical induction variables), the remaining ones are still quite restrictive. The desired results, namely the SCoPs, are valuable because they can be represented within the polyhedral model, thus further analyses and transformations do not need to deal with the details of the represented code.

Although all regions fulfilling these requirements are technically SCoPs, we will restrict ourself to those containing at least one loop.

2.3.2 SCoP Detection

Pollys SCoP detection is the gateway to all further analyses and transformations. All regions fulfilling the properties described in the last section, or in short, all (valid) SCoPs are detected here. The special interest of this part arises from the fact that all regions declared as valid SCoPs will be considered for polyhedral optimizations. Almost regardless of their content, any region could be given to Polly if the SCoP detection is properly instrumented. Two important consequences can be derived and summarized as: Utilizing the strength of Polly is possible in much more situations as intentionally implemented but coupled with responsibility of the outcome.

2.3.3 Loop Optimizations

Polly uses the integer set library (isl) to compute the scheduling and tiling for a SCoP. Once the polyhedral representation is computed, an optimized version of the algorithm proposed by Bondhugula et al.[5] will compute a new scheduling and tiling scheme. Traditional loop optimizations such as blocking, interchange, splitting, unrolling or unswitching are implicitly applied during this step.

2.3.3.1 Parallel Code Generation

While cache locality is implicitly improved by rescheduling and tiling of the loop nest, parallel code needs to be generated explicitly afterwards. Polly is capable of generating thread level parallelism using OpenMP annotations and data level parallelism using SIMD instructions. If the former one is desired, the first loop without any loop carried dependencies will be rewritten as if there were OpenMP annotations in the first place. Enabling the later one, namely vector code generation, will not only try to vectorize the innermost loop but also the scheduling in order to allow this in more cases.

Further Reading

- Polly - Polyhedral optimization in LLVM [3]
- Enabling Polyhedral Optimizations in LLVM [2]
- Base algorithm of isl [5]
- `http://polly.llvm.org`
- `http://www.kotnet.org/~skimo/isl/`

2.4 Sambamba A Framework For Adaptive Program Optimization

As an extension to LLVM the Sambamba compiler framework is designed to allow runtime analyses and (speculative) optimization. Furthermore these optimization can create and refine runtime profiles which are used to recalibrate and specialize the (speculative) execution. Method versioning allows conservative and speculative versions of a method to be stored and switched during runtime. Written in a completely modular way, Sambamba extensions consist of a static part (compile time) and a dynamic one (runtime). Both extension parts can use Sambamba to store information, collected at the corresponding time, accessible for the dynamic part at runtime. Profiling combined with the method versioning system allows runtime interactions to explore more parallelism or minimize the overhead in case of misspeculation.

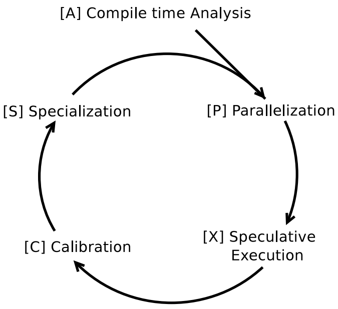


FIGURE 2.3: Different stages in the Sambamba framework

Stages in the Sambamba framework

As illustrated in Figure 2.3 we can differentiate five stages embedded in the Sambamba framework.

- **[A]** Execute static whole-program **analyses** with the possibility to save their results.
- **[P]** Speculatively **parallelize** parts of the program based on the results of [**A**] and [**S**].
- **[X]** Detect and recover from conflicts by speculative **execution** using a software transactional memory system.
- **[C]** Use the profiles and misspeculation information gained during the execution to **calibrate** future optimization steps.
- **[S]** Generate **specialized** variants of a function, one for each special input profile detect by [**C**].

2.4.1 Sambambas Parallelizer

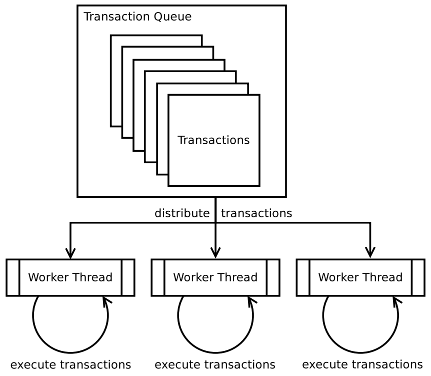


FIGURE 2.4: Symboliced transaction queue with three worker threads

The Sambamba parallelizer will become the main interface for any kind of parallelization in the framework. At the moment it is in need of parallel control flow graphs to explicitly state section which should be executed in parallel, but in the future an automatic loop parallelization component will be implemented too. The runtime part of the parallelizer instantiates as many worker threads as there are cores on the executing machine. Each worker thread will later on execute one transaction at a time. In contrast to the OpenMP approach no external libraries are needed to exploit parallelism.

2.4.2 Parallel Control Flow Graphs

A parallel control flow graph (ParCFG) is a data structure used by the parallelizer to express parallel sections within an ordinary CFG. Each parallel sections consist of an entry block called `piStart` and an exit block called `piEnd`. The `piStart` is terminated by a symbolic switch statement which may have an arbitrary number of successors. Every successor denotes a, so called, transaction which ends in the `piEnd` block after some arbitrary computation. To put it in another way, parallel sections correspond to fork-join parallelism where the `piStart` blocks forks transactions which are synchronized at the `piEnd` block. In the case of Sambamba this synchronization is done by a software transaction memory as described in the next section.

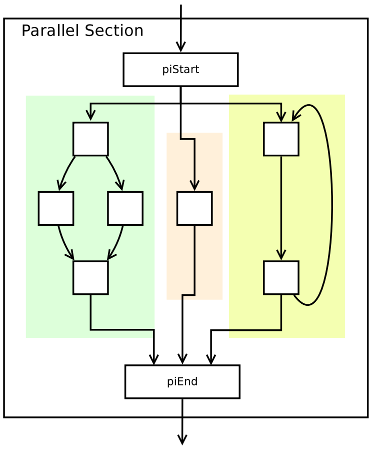


FIGURE 2.5: A parallel section with three transactions

Figure 2.5 shows such a parallel section with three highlighted transactions. At runtime they will be executed concurrently, each by one worker thread at a time.

2.4.3 Software Transactional Memory

A software transactional memories (STM) is a conflict management system used to detect memory conflicts in speculatively parallelized programs and closely related to the transaction system in databases. When in use the STM logs all accesses to the memory from the beginning of a transaction up to the synchronization. If, during this time, a memory address has been written and read by different threads, the reading ones might have worked with corrupted data, thus all resulting computations might be wrong. In such a case the STM will force the threads to recompute theirs results. There are different implementations which will, for example, allow at least one thread finish its computation before all other might have to recompute theirs. Such implementations will always preserve the liveness of a system as there is progress, even though the required time might be quadratic in the number of threads.

As the described implementation only preserves liveness but no ordering between the different computations, you might want to use, so called, commit orders. A commit is the final synchronization step of a thread in an STM environment and a commit order will enforce the threads to synchronize, thus store their result permanently, in the exact same order as they have been created. Just like the unordered version this one guarantees liveness.

Another possible extension for an STM system is a wait construct which is build on top of the commit order. Such a wait will cause the thread to pause until all other threads, in front in the commit order, have passed a corresponding barrier, intentionally placed only a few instructions behind the wait. The enclosed instructions would form a mutual exclusive area which can only be entered by one thread at a time.

The current STM implementation in the Sambamba framework does neither provide a commit order nor the wait construct, but this will change in the future.

Further Reading

- Sambamba: A Runtime System for Online Adaptive Parallelization [17]
- `http://www.sambamba.org`

Chapter 3

Concept

From a high point of view SPolly is divided into a speculative loop parallelizer and a non speculative extension to Polly. Even if the objectives for both parts are the same, namely to improve the performance of loops, the approaches to accomplish them are different. While the former one will introduce speculative parallelism for promising loops at runtime, the later one tries to weaken the harsh requirements on SCoPs in order to make Pollys loop optimizations applicable on a wider range of loop nests. In the presented setting both approaches may benefit from the polyhedral optimizations and also from parallel execution, so it is hardly surprising that the polyhedral analyses play a decisive role. On the one hand they reveal loop nests which may be optimized by Polly, with or without the extensions of SPolly, on the other hand they are used to detect promising loops to speculate on. Apart from the implementation work, which will be described in the next chapter, this thesis presents the concepts and key ideas behind. We believe that these ideas and the gained knowledge is very valuable not only for future work on SPolly or one of its bases but also for other approaches facing similar situations.

3.1 Speculative Parallel Execution

Speculatively executing loops in parallel is one of the two major purposes of SPolly. The challenge was to exploit as much parallelism as possible without restricting the applicability to much.

3.1.1 s

Even if the execution is secured by a (not extended) STM it may not ne sound to execute a loop nest in parallel. If there is a conflict between two iterations and the later one (in terms of the original ordering) commits its result before the former one could, the STM will force the former one to recompute, but now based on corrupted data already committed and therefore permanently written. From this fact the need of an commit order arose, as it would in any case preserve the initial ordering when applying permanent changes.

As one great feature of Polly and the underlying polyhedral model is to detect and model loop carried data dependencies, it sounds plausible to use this for speculative execution purposes as well. Unfortunately speculative extensions will compromise this ability because they will speculatively assume less dependencies in order to weaken the requirements. To handle such situations SPolly tries to unify both of its parts. At first, non computable memory accesses, as they arise from aliases we cannot check in beforehand or from function calls within a SCoP, are overestimated. The resulting polyhedral model is complete with regards to possible data dependencies, but could anyway reveal transformations for a better data-locality or possible vectorization. Afterwards speculation is applied, thus the optimized loop nest is speculatively parallelized. Note here, that only Polly could change the iteration order but because this would be based on a complete and overestimated polyhedral model, it would still yield a sound ordering.

In the worst case there are dependencies between all parallel executed iterations and each thread has to recompute its part once another one committed its results. As countermeasure to such time consuming misspeculations SPolly monitors the runtime of all speculatively parallelized SCoPs and reverts the speculation for poorly performing ones.

In contrast to the speculation free optimization approach, this one is also capable of parallelizing sSCoPs with almost arbitrary function calls. To do so the STM has to provide the wait construct as described earlier. Even if this is not yet the case, SPolly could introducing such wait with no further effort. This will certainly cause additional overhead during the parallel execution, but it does not preclude a performance gain per se.

In the special case of sSCoPs, we know certain things about the control flow and the memory accesses within a parallelized region. With the assumptions we may derive from the requirements on sSCoPs it is possible to secure the speculative parallel execution by introducing wait constructs right before each possibly violating function call. Executing those function calls in the correct ordering will enforce all their possible effects to be applied in the correct order too. If arbitrary control flow and conditionals are allowed, this conclusion does not hold anymore. In fact every memory access could change the taken execution path of another thread, thus every branch would need annotation.

3.2 Speculation Free Optimizations

As second objective SPolly may enable sound polyhedral optimizations, even parallelization, on sSCoPs with no need for speculation at all. This is the case when the tests we introduce in the following can rule out all violations of a sSCoP.

3.3 SPolly In A Nutshell

During **compile time** the main goal of SPolly is to simplify the runtime part, thus to reduce runtime overhead through preprocessing and even static method versioning. First the SCoP detection tries to find valid regions within the given LLVM-IR, but instead of rejecting a region once a restriction is violated, the region speculation is asked how to proceed. Restrictions we want to speculate on are gathered by the region speculation, but ignored by the SCoP detection. This proceeding allows to find all violations with a region and to treat valid and speculatively valid ones nearly the same. After all speculative valid SCoPs (or short sSCoPs) are encountered the Sambamba compile time part takes action. It separates the sSCoPs as some of them do not need speculation at all. Those sSCoPs are optimized and exchanged, while the others are currently only extracted. This means they are replaced by calls to functions only containing the speculative valid region. At the moment, no pre computation takes place and the onliest violations not dependent on speculation are special kinds of aliasing instructions, namely those which can be ruled out by tests in beforehand.

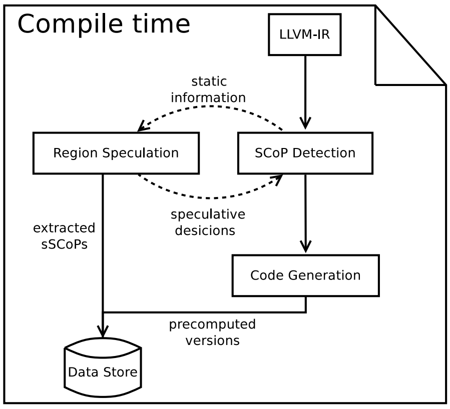


FIGURE 3.1: Draft paper: SPolly at compile time

The **runtime** part of SPolly first retrieves the extracted sSCoPs and precomputed versions from the data store. If not already done during compile time, profiling versions will be created now. It would be possible to restrict this to the best rated sSCoPs only, but as the creation is very cheap and the execution overhead for most of them is non-existent it is feasible to do so for rather bad ranked sSCoPs as well.

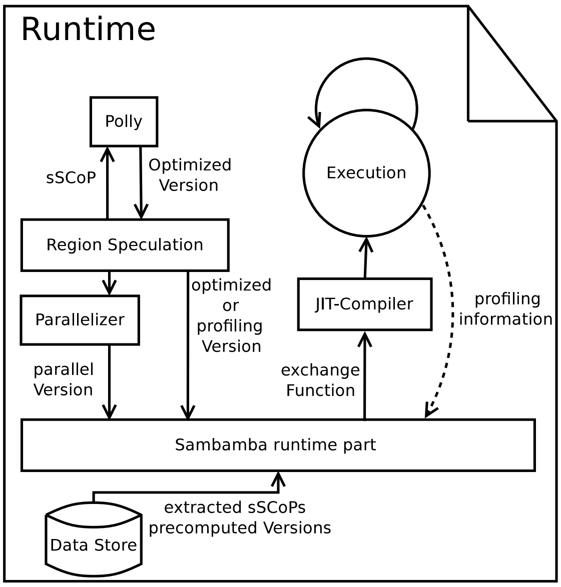


FIGURE 3.2: Draft paper:

Those profiling versions will now collect information not only about the time consumption of the sSCoP, but also about loop bounds, branch probabilities and the results of introduced checks. Except of the time consumption these values will affect the rating of the sSCoP which again is used to identify promising sSCoPs.

The next section will explain this rating and the effects of profiling in more detail, while section 3.6 covers the test creation and theirs use. As explained above, the combined static and dynamic information determine which region is promising, thus which region will be speculatively optimized and in the end executed in parallel. Because the impact on the performance might be worse than expected, maybe even worse than the sequential execution, the runtime part will continually monitor all exchanged functions and intervene if necessary.

3.4 Region Scores

```
// A, B, C may alias
for (i = 0; i < 1024; i++) {
  A[i] = B[i] * C[i];
}
```

(A) Complete static sSCoP

```
// f is not declared as readnone
for (i = 0; i < N; i ++) {
  for (j = 0; j < M; j++) {
    A[i][j] = f(i,j);
  }
}
```

(B) sSCoP with variable loop bounds and a function call

```
for (i = 2; i < 1024; i++) {
  if (...)
    A[i] += A[i-1] * A[i-2];
  else
    A[i] -= A[i-1];
}
```

(C) Branch within a sSCoP

```
for (i = 0; i < N; i ++) {
  if (A[i] == 0)
    printf("Unlikely error!");
  else
    A[i] = 1024 / A[i] ;
}
```

(D) irreversible call within a sSCoP

Figure 3.3: example sSCoPs

Region scores are used as a heuristic to decide whether or not a sSCoP is worth to speculate on, thus for which regions profiling and optimized versions should be created and as a result executed. As the former ones may change the score again it is reasonable to create optimized versions only if the profiling results suggest to do so. It is obvious that we want to consider only loops and loop nests of a certain size, thus profiling the trip count may have a enormous impact on the actual region score. Additionally we are interested in execution paths within a sSCoP in order to predict how often e.g., irreversible instructions, may be executed. While such instructions, like calls of `printf`, may cause STM rollbacks during the parallel executions, branch probabilities may also have a huge impact on the actual sSCoP size and only rarely occurring dependencies. To clarify the idea, the regions scores for the listings 3.3a to 3.3d as well as some other listings contained in this thesis is listed in table 3.1. For a detailed explanation on the implementation and meaning see the corresponding section in the next chapter.

Table 3.1: Scores for the sSCoPs presented in various listings

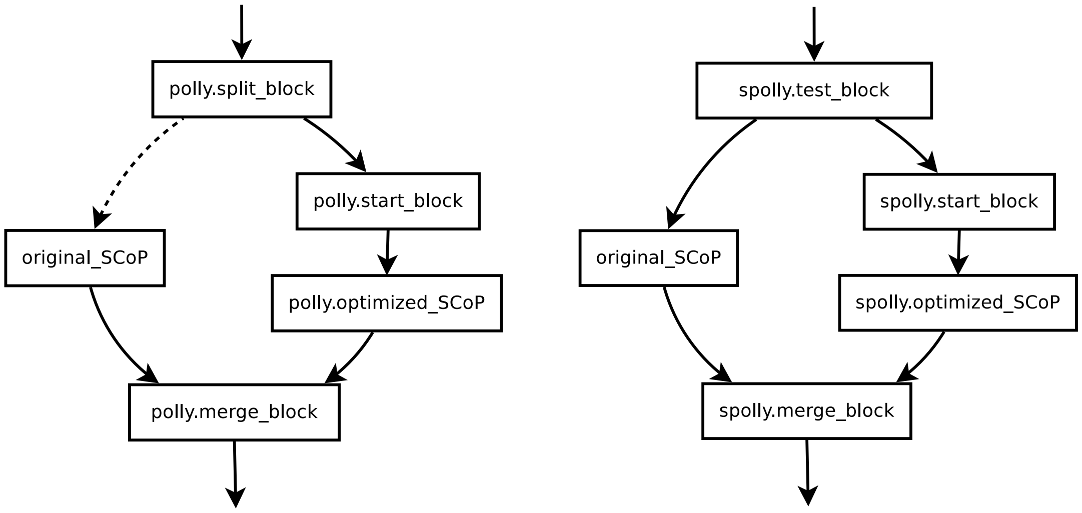
listing	score
2.1a	408
3.3a	576
3.3c	63 ∗ (11 + ((7 ∗ @if.then_ex_prob)/100) + ((5 ∗ @if.else_ex_prob)/100))
3.3b	((0 smax %N)/16) ∗ (7 + (10 ∗ ((0 smax %M)/16)))
3.3d	((0 smax %N)/16) ∗ (6 + (−1000 ∗ @if.then_ex_prob/100)
4.3b	(7 + ((8 + (8 ∗ (%N/16))) ∗ (%N/16))) ∗ ((0 smax %N)/16)

3.5 Method Versioning

Even if method versioning in Sambamba is not fully implement yet, SPolly is already capable of generating a profiling and an optimized version. Both just transform the sSCoP, thus only the loop or loop nest has to be cloned and stored. Further work on SPolly will include the creation of more different optimized versions as there are plenty of optional parameters which could be adjusted as needed. The impact of tiling size, loop fusion and the amount of actual introduced tests for the optimized version may have significant impact, but in the short time of this work it was not feasible to investigate them fully. Future, work as described last chapter, may use the already present infrastructure to adapt and specialize sSCoPs further.

3.6 Introduced Tests

The mentioned tests are one attempt to keep down the rate of misspeculations. First of all they are used to refine region scores in the context of profiling but they can be of great use in optimized versions too. At the moment SPolly is capable of creating two different kind of checks, which may partially rule out SCoP violations completely. If so we may call the tests complete and we apply them on the optimized version with no need of speculate at all. As such cases can improve the applicability of Polly even without an STM, they could find their way into the main branch one day. At this point we take advantage of Pollys default behaviour which is to copy the optimized SCoP as an alternative to the original one. Figure 4.2a shows the CFG after Polly optimized a given SCoP. The dotted edge is not taken since the guard of the conditional is constant true, but in the SPolly version it is replaced by the actual test result (see figure **??**).



(A) CFG after optimization with Polly

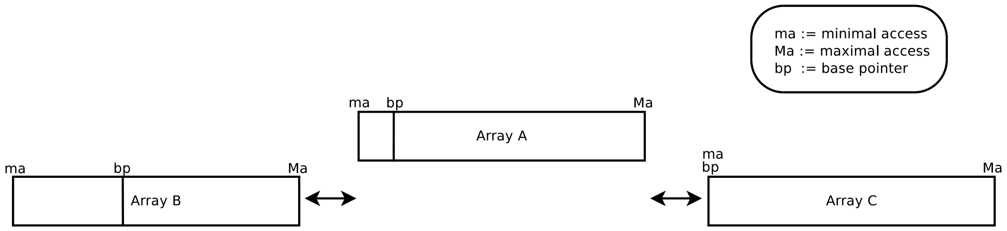
(B) CFG after optimization with SPolly

Figure 3.4: CFG produced by Polly and SPolly, respectively

3.6.1 Alias Tests

Testing for aliasing pointers in general would not be feasible so another way was chosen. Only sSCoP invariant pointers are tested once before the sSCoP is entered. If the test succeeds, thus no aliases are found, the optimized version is executed. At compile time the accesses for each base pointer are collected and marked either as possible minimal, possible maximal or not interesting access. At runtime all possible minimal and maximal accesses are respectively compared until, in the end, the minimal and the maximal access for each base pointer is computed. The alias test as such compares again the minimal access for a base pointer with the maximal accesses for each other base pointer and vice versa. After this comparison chain the result may indicate or rule out aliasing between them. As Polly introduces two versions of a SCoP by default, we may just replaces the constant guard in the split block in order to choose the executed version based on the test result. If all base pointers are invariant in the SCoP the test is complete, thus aliasing can be ruled out for the sSCoP at runtime. However, non invariant pointers are not tested at all, as it would imply to perform all computation and testing within the loop. Figure 4.3a illustrates the concept of the alias tests while listing 4.3b and figure

4.3c provide an example loop nest and the corresponding minimal and maximal memory accesses. The alias test for this example would look like listing 4.3d.



(A) Alias test from a birds eye view

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
//  I1
    C[i][j] = 0;
    for (k = 0; k < N; k++) {
//    I2          I3        I4
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

(B) Aliasing accesses

Acc	bp	ma	Ma
I1	C	0	N∗N−1
I2	C	0	N∗N−1
I3	A	0	N∗N−1
I4	B	0	N∗N−1

(C) Statically derived min/maximal accesses

```
bool ab = B[N*N-1] < A[0] || B[0] > A[N*N-1];
bool ac = C[N*N-1] < A[0] || C[0] > A[N*N-1];
bool bc = B[N*N-1] < C[0] || B[0] > C[N*N-1];
bool result = ab && ac && bc;
```

(D) Introduced compare chain

Figure 3.5: Alias tests concept and example

3.6.2 Invariant Tests

Apart from alias tests, SPolly may introduce invariants tests if there are possibly invariant variables and a function call within a sSCoP. The key idea is to monitor possible changes in such variables during the execution of the profiling version. As the results may introduce new dependencies between loop iterations, the sSCoP could be discarded. If it does not, the sSCoP may be optimized, depending on its new region score. Even this is a disqualification test in the first place, the information gathered about the variables could be used to create specialized sSCoP versions too. Listing 4.4a gives an example of an sSCoP for which invariant tests can be introduced and 4.4b shows the modified source.

```
int c;

void f() {
  int i;
  for (i = 0; i < 1024; i++) {
    // function g may change c
    A[i] = g() + c;
  }
}
```

(A) Loop nest with possible invariant variables

```
int c;

void f() {
  int i;
  int c_tmp = c;
  for (i = 0; i < 1024; i++) {
    if (c != c_tmp)
      signalNonInvariancy();
    // function g may change c
    A[i] = g() + c;
  }
}
```

(B) Loop nest with invariant tests

FIGURE 3.6: Invariant test introduced by SPolly

3.7 Non Computable Dependencies

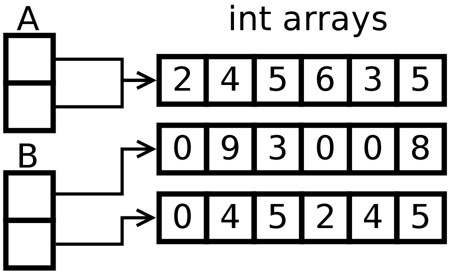
Ruling out may aliases due to checks as described earlier is not feasible in every situation. Assuming the example in listing 3.7a we may check if `A` and `B` alias in front of the loop nest, but every array `A[i]` and `B[i]` may alias also. Introducing tests within the loop nest is possible, but it would reduce the possible parallelism to outermost loop not containing the aliases. As in the presented example type based alias analysis could lift the computation of `A[i]` and `B[i]` to the outer loop, the innermost one would become a valid SCoP after checks are introduced. At the moment SPolly is not capable of detection and deciding if such an approach could be profitable, but instead SPolly tries to optimize and speculate on the whole loop nest. As described earlier SPolly will overestimate the data dependencies in order to apply sound transformations before it will speculatively execute the splitted loop nest in parrallel. This example is well suited to illustrade why even an STM based approach cannot detect all conflicts when no commit order is available. Assuming we would translate the loop to the pseudo parallelized version in listing 3.7c with the corresponding ParCFG as illustrated by figure 3.7d. An input such as the one in figure 3.7b seems very unlikely but nevertheless it could produce wrong results without causing an syncronizing conflict. This will happen everytime the second transaction is commited and therefor written first. This will cause the first one to recompute its results, because they worked on the same memory locations, namely the array `A[0] = A[1]`. During this recomputation a wrong value for `A[0][j]` will be used without beeing reported as a conflict on the syncronization.

3.7.1 Overestimating Dependencies

To handle loop nests with unknown dependencies, a conservative approach which pretends dependencies between all possibly aliasing instructions is already implemented. The strategy is quite similar to the one Polly uses for non affine memory accesses as

```
void f(int **A, int **B, unsigned b) {
  int i,j;
  for (i = 0; i < 1024; i++) {
    for (j = 0; j < 1024; j++) {
      A[i][j] = A[i][j] * B[i][j];
    }
  }
}
```

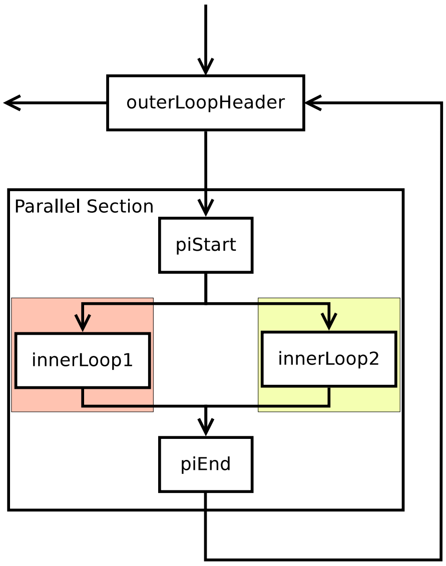
(A) Loop nest with non computable dependencies



(B) Possible input situation for listing 3.7a

```
void f(int **A, int **B) {
  int i,j;
  for (i = 0; i < 1024; i += 2) {
    // Begin parallel section
    // Transaction 1
    for (j = 0; j < 1024; j++) {
      A[i][j] = A[i][j] * B[i][j];
    }
    // Transaction 2
    for (j = 0; j < 1024; j++) {
      A[i+1][j] = A[i+1][j] * B[i+1][j];
    }
    // End parallel section
  }
}
```

(C) Loop nest 3.7a pseudo parallelized



(D) ParCFG for listing 3.7c

Figure 3.7: Example for non computable dependencies and a violating input

both overestimate the access until it is sound to proceed. This technique is quite restrictive but even though it may allow loop invariant code motion or even vectorization.

Considering listing 3.7a again this method will hoist the conditional out of the loop and at the same time reveal that the innermost one has no loop carried dependencies.

3.8 What Is Missing

Chapter 4

Implementation

SPolly in its entirety is a compound of three parts. The region speculation, embedded into Polly, and the two Sambamba modules for compile time and runtime respectively. The region speculation part is the interface to all discovered sSCoPs, thus it contains most of the transformation code, while the Sambamba passes concentrate on the program logic. At the moment the runtime part is far more evolved and the compile time component plays only a minor role. Apart from SPolly itself, a basic profiler and a statistic module for Sambamba arose during this work. Both have been very helpful during the development and may become permanent features of Sambamba. During the implementation various bugs occurred and even if most of them arose through my own fault I triggered some in the code base of Polly and Sambamba too. A full table of reported bugs is listed in table **??**.

4.1 Speculative Polly

It would be feasible to look at SPolly as extension to Polly, especially designed to interact with Sambamba. As such it was crucial to preserve all functionality of Polly and supplement it with new ones. Most of them are implemented in the region speculation, but there are some new options in the code generation too. The bridge between the region speculation and Polly is is ideally suited to serve as the bridge between Polly and the speculative part as speculative valid regions would be rejected here. The information currently needed for region speculation is also available at this point and can be directly reused.

As the architecture of Polly is nicely illustrated by figure **??**, it has been extended in figure 4.1 to capture the changes introduced by SPolly. In comparison the region speculation, the fork join backend and the sSCoP backend have been added as they can be used without Sambamba.

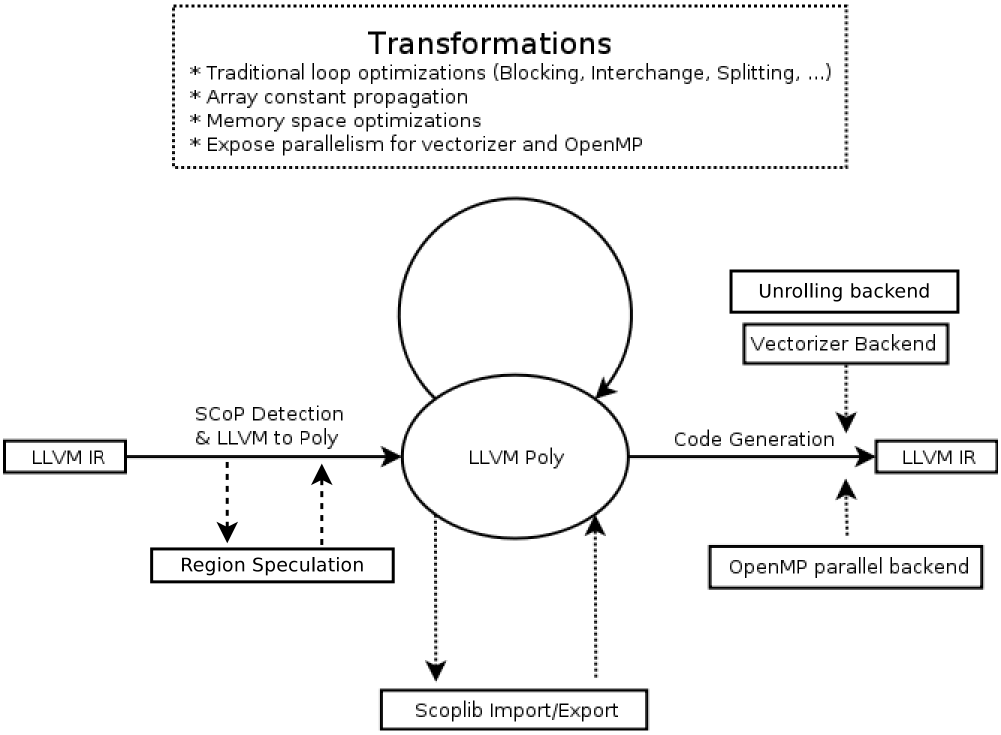


FIGURE 4.1: SPolly architecture

4.1.1 Region Speculation

The region speculation (RS) of SPolly has several tasks to fulfill. The first one includes the communication with Polly or more precise, with the SCoP detection. Each region analyzed by the SCoP detection needs to be considered as possibly speculative valid SCoPs, thus all information exposed during the detection are stored. If the region contains a validation not listed in 4.1 or if it is without any validation the information are discarded. Valid SCoPs are handled by Polly while for others a new sSCoP is created which initially validates itself. These validation mainly computes information needed for later transformations but it maybe discards the sSCoP too. This is the case if violating function calls, e.g., a “printf”, occur on every execution path. In the following these computations as well as the creation of profiling and parallel versions for a sSCoP are explained briefly.

4.1.1.1 sSCoP extraction

The sSCoP extraction was designed to simplify the method versioning for functions containing several speculative valid SCoPs. It creates a new sub function for every sSCoP and inserts a call in the former place. Later on this call could be inlined again but this is not implemented yet. On the one hand the creation of profiling and parallel versions as well the later function exchanging becomes a lot easier and cheaper this way.

Table 4.1: Restrictions on sSCoPs

- Only perfectly nested loops and conditionals
- No unsigned iteration variables [1]
- Only canonical PHI nodes and induction variables
- Instructions may not be used in PHI nodes outside the region[2]
- Only speculatively “non violating” function calls
- No PHI nodes in the region exit block
- Only simple regions not containing the entry block of the function

[1] open for further work of Polly

[2] open for further work of SPolly

On the other hand it is the first step to multiple specialized versions, e.g., with constant instead of variable loop bounds. Section **??** will cover this in more detail.

4.1.1.2 sSCoP Versions

Method versioning is one of the great benefits of the Sambamba system and allows to adapt the execution at runtime. In the scope of this work two different versions are created, a profiling one and a optimized one. As these two are explained in the following.

The Profiling Version

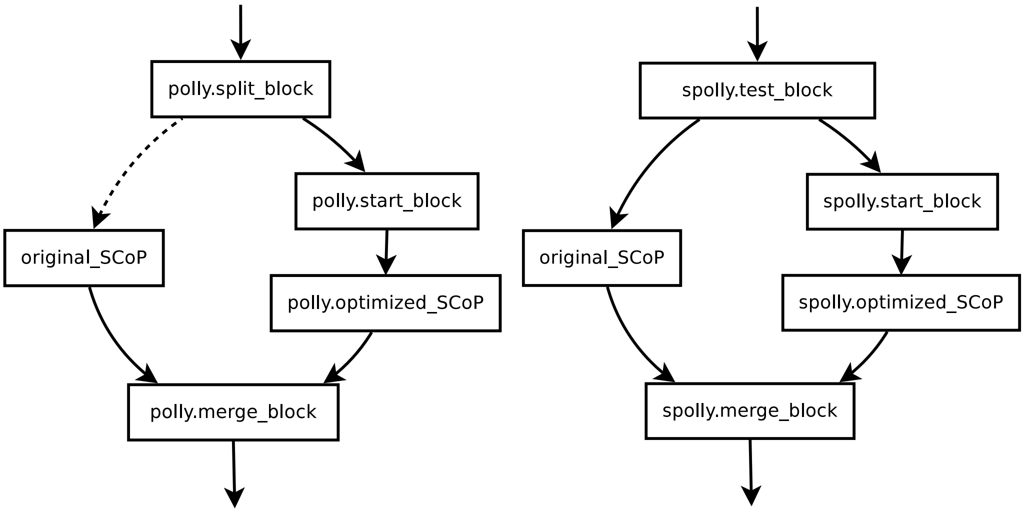
Profiling is a powerful ability for just in time executed programs, like the ones produced by Sambamba. SPolly benefits not only from the introduced tests, but also from the monitored branch probabilities and loop trip counts. The retrieved information are in the first place used to compute the region scores, thus to improve the heuristic which chooses sSCoPs for optimization. Later on, specialized versions may arise based on them.

The Optimized Version

Just as valid SCoPs may valid sSCoPs be optimized by Polly, but there are also differences. As Polly normally takes care of possible dependencies it cannot do the same for all kinds of sSCoPs. As stated in section 3.7 there is a safe way to overestimate all possible depenencies, thus to use Polly for optimizations. This way is very restrictive and therefor it does not yield a desired speedup that often. If SPolly can introduce complete checks, the situation changes and

4.1.1.3 Introduced tests

To reduce the overhead of misspeculation tests are introduced in front of each sSCoP. At the moment there are two different kinds available, invariants and alias tests. Only the later one is used in optimized versions because the former one needs to check the invariant every iteration, thus it produces a huge overhead. In contrast to profiling versions which uses both tests to refine the score of a sSCoP. Placing the test section was quite easy, since Polly itself introduces a (constant) branch before the SCoP anyway. Figure 4.2a and 4.2b shows the (simplified) CFG introduced by Polly and SPolly, respectively. The dotted edge in the CFG produced by Polly is not taken (constant false), but remains in the CFG. As far as I know, SPolly is the first extension to Polly which uses the untouched original SCoP.

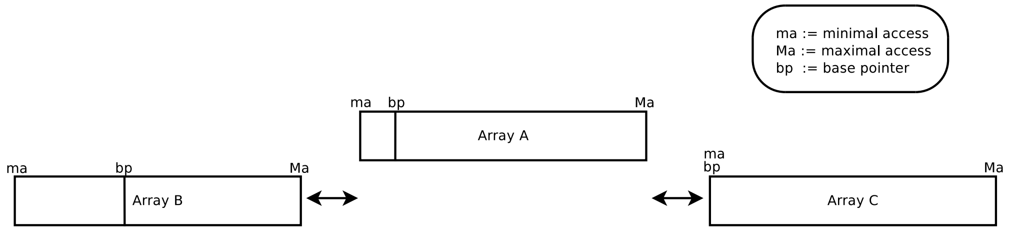


(A) Polly CFG (B) SPolly optimized CFG

FIGURE 4.2: CFG produced by Polly and SPolly, respectively

Alias tests

Testing for aliasing pointers in general would not be feasible so another way was chosen. Only sSCoP invariant pointers are tested once before the sSCoP is entered. If the test succeeds, thus no aliases are found, the optimized version is executed. At compile time the accesses for each base pointer are collected and marked either as possible minimal, possible maximal or not interesting access. At runtime all possible minimal and maximal, respectively, accesses are compared and the minimal and maximal access for each base pointer is computed. The alias test as such compares again the minimal access for a base pointer with the maximal accesses for the others and vice versa. At the end of this comparison chain the result replaces the constant guard in the split block right before the original SCoP and the speculative optimized one. If all base pointers are invariant in the SCoP the test is complete, thus aliasing can be ruled out for the sSCoP at runtime. However, non invariant pointers are not tested at all, as it would imply to perform all computation and testing within the loop. Figure 4.3a illustrates the concept of the alias tests while listing 4.3b and figure 4.3c provide an example with the derived accesses. The alias test for this example would look like listing 4.3d.



(A) Alias test from a birds eye view

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
//  I1
    C[i][j] = 0;
    for (k = 0; k < N; k++) {
//    I2         I3         I4
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

(B) Aliasing accesses

Acc	bp	ma	Ma
I1	C	0	N∗N−1
I2	C	0	N∗N−1
I3	A	0	N∗N−1
I4	B	0	N∗N−1

(C) Statically derived min/maximal accesses

```
bool ab = B[N*N-1] < A[0] || B[0] > A[N*N-1];
bool ac = C[N*N-1] < A[0] || C[0] > A[N*N-1];
bool bc = B[N*N-1] < C[0] || B[0] > C[N*N-1];
bool result = ab && ac && bc;
```

(D) Introduced compare chain

Figure 4.3: Alias tests concept

Invariants tests

Apart from alias tests, SPolly may introduce invariants tests if there are possibly invariant variables and a function call within a sSCoP. The key idea is to monitor possible changes in such variables during the execution of the profiling version. As the results may introduce new dependencies between loop iterations, the sSCoP could be discarded. If it does not, the sSCoP may be optimized, depending on its new region score. As this is a disqualification test in the first place, the information gathered about the variables could be used to create specialized sSCoP versions. As mention earlier, the last chapter will discuss specialized versions in more detail. Listing 4.4a gives an example of an sSCoP for which invariant tests can be introduced and 4.4b shows the modified source.

```
int c;

void f() {
  int i;
  for (i = 0; i < 1024; i++) {
    // function g may change c
    A[i] = g() + c;
  }
}
```

(A) source

```
int c;

void f() {
  int i;
  int c_tmp = c;
  for (i = 0; i < 1024; i++) {
    if (c != c_tmp)
      signalNonInvariancy();
    // function g may change c
    A[i] = g() + c;
  }
}
```

(B) source with invariant tests

FIGURE 4.4: Invariant test introduced by SPolly

Complete Checks

Alias tests may rule out aliasing in sSCoPs completely, thus some sSCoPs become valid SCoPs after this tests are introduced. Such non speculative optimizations are done by the compile time part of the Sambamba module and may be included in Polly too. An example for such an sSCoP is given in figure 4.3b. The presented code is the well known matrix multiplication example which is a valid SCoP if the arrays do not alias. For the sake of completeness it is to mention that this code could be rewritten as array of pointers which would also lead to a sSCoP but without complete checks. Chapter 6 will discuss this example and the various implementations in detail.

4.1.2 Code Generation

As part of the extension of Polly a new code generation type was added. Apart from sequential, vectorized and OpenMP annotated code generation SPolly is capable of creating a unrolled and blocked loop, which can be easily translated into an ParCFG, thus parallelized by a Sambamba module. Listing 4.5b presents this transformation. The special case where lower and upper bound as well as the stride are statically known constants, the second loop, which computes remaining iterations, is completely unrolled. This kind of loop unrolling and blocking may find its way into Polly in the near future.

Parallelization

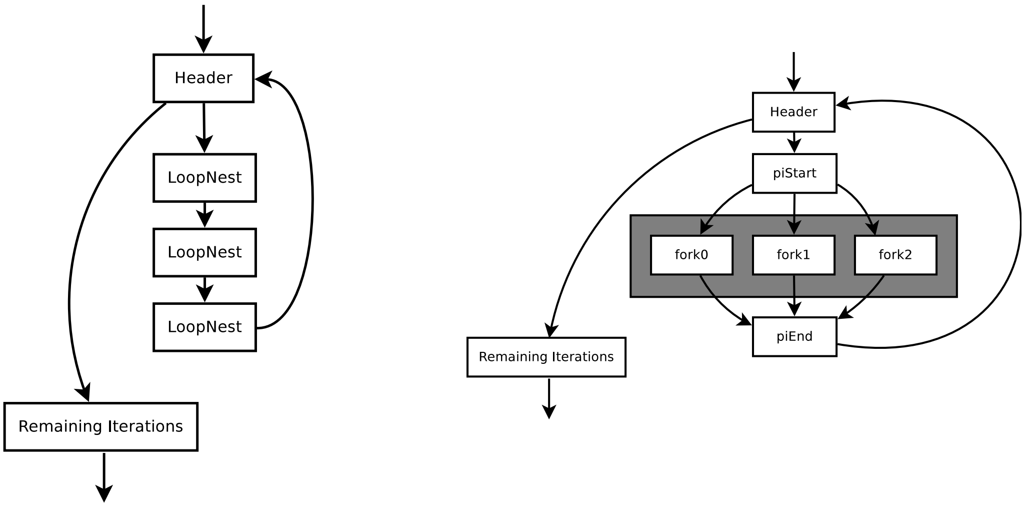
In order to secure the speculative executions with Sambambas STM (see **??**) the Sambamba parallelizer needs to be used. As this parallelizer does not yet support loop parallelization per se, some transformation needed to be done first. The new code generation type was created to do the difficult one without recomputing information provided during by the polytope model (during the code generation) anyway. As these transformation yields code as in listing 4.5b the creation of a ParCFG (see **??**) remains. Figures 4.5c and 4.5d visualize these changes.

```
void f() {
  int i;
  for (i = B; i < E; i += S) {
    LoopNest(i);
  }
}
```

(A) Initial loop nest

```
void f() {
  int i;
  for (i = B; i < E - N + 1; i += S * N) {
    LoopNest(i + 0 * S);
    ...
    LoopNest(i + N * S);
  }
  // Remaining iterations
  for (int j = i; j < E; j += S) {
    LoopNest(j);
  }
}
```

(B) Listing 4.5a after N-fork creation



(C) SPolly forked CFG

(D) SPolly ParCFG

Figure 4.5: Forked CFG produced by SPolly and resulting ParCFG

4.2 Sambamba Compile Time Module

The compile time part of the Sambamba module locates all sSCoPs within the given input module and transfers each one afterwards in a separated function. These extracted sSCoPs are stored within the created Sambamba bitcode or respectively executable file. Extracting every single sSCoP decreases the performance but allows to easily change and combine different optimized sSCoPs, even if they originated from the same function. At the moment there is one exception implemented which will be applied on sSCoPs with complete checks, thus valid SCoPs after the tests are passed. All of those are optimized in place without consulting the region scores or any other heuristic. The functionality of the compile time part is minimal but it helps to focus on profiling and execution during runtime, without analysis overhead. Further work could heavily improve this part, beginning by compile time preparation of the sSCoPs, but imaginable is more than just a precomputed profiling or optimized version of a sSCoP. As there are a lot of parameters which could have significant impact on the performance, several optimized versions of an sSCoP could be created and stored, in order to choose the best depending on the system and the actual run, thus depending on runtime information. While table B.2 gives a brief overview of available options for Polly, it is not clear which ones will fit best for a particular environment and sSCoP. As the method versioning system of Sambamba evolves, the compile time part should, in order to reduce the workload at runtime and increase the ability to adapt.

4.3 Sambamba Runtime Module

In addition to the compile time parts, which only rely on static analyses, the runtime part uses different kinds of runtime information to decide. To take advantage of this extra knowledge, most of the decisions, thus most of the program logic, is implemented in the runtime module.

Table **??**[TODO] gives an overview of the functionalities.

Table 4.2: Functionality of the SPolly Sambamba module

4.3.1 Profiling

4.4 Profiling For Sambamba

Sambamba, as heavily developed research project, was not capable of any kind of profiling when I started my work. By now, there are two profilers available. The first one, implemented by the authors of Sambamba, is used for exact time measuring, while I created the second one to profiles executions and data. [TODO if first is used later on, write it here] Both were developed during the same time to fulfill different needs and could be [TODO will be ?] merged anytime soon. As most of SPollys parts are unrelated, to both of them, the profiling versions, as their name indicates, would become useless without. This will definitely increase the number of unnecessary created (parallel) functions, but it would not render SPolly redundant. There a sSCoPs which a guarded by sound checks, thus they can be used with the overhead of only the check. As the use of other sSCoPs could increase the performance too, a heuristic could look for promising candidates, even without any runtime information. This heuristic could be part of future work since even with a profiler by hand, there are cases where the gathered information (see [TODO]) are not helpful at all.

4.5 Region Scores

Initial efforts to create these scores did not use any kind of memory, thus the whole region was analysed every time new information was available. To avoid this unnecessary computations the current score is a symbolic value which may contain variables for values not known statically, e.g., branch probabilities or loop bounds. Evaluation of these symbolic values will take all profiling information into account and yield a comparable integer value. Only during the initial score creation the region will be traversed to find parameters and branches for later annotation.

All instructions will be scored and the violating ones will be checked for their speculative potential. As memory instructions are guarded by the STM for the case the speculation failed, calls may not be reversible, thus without any speculative potential at all. Such function calls are not checked earlier since the region speculation needs the information about possible other branches within this region. Listing 3.3d provides such an example but these cases will be revisited in the next two chapters too. Table 3.1 lists the scores for the examples in figure 3.3.

Chapter 5

Evaluation

The evaluation, as essential part of this thesis, allows to compare the this work with others which share the same goals. Since it is hard to put effort in figures, the evaluation may provide information about the results of them. Correctness, applicability and speedup have been the stated goals which were tested using the SPEC 2000 and Polybench 3.2 benchmark suites. While the former one contains general benchmarks used to evaluate most new optimizations, the later one is especially designed for polyhedral optimizations as done by SPolly. These benchmarks might not reflect the reality in everyday optimization but they might be used to compare SPolly and Polly (as presented in the diploma thesis of Tobias Grosser [2]).

5.1 The Environment

(more details in table 5.1). running Arch linux with an *Intel(R) Core(TM) i5 CPU M 560 @ 2.67GHz* and 6GB RAM. Parallel versions could use up to four simultaneous running threads. The second one ... TODO ... As the compile time evaluation is machine independent, it was performed on the general purpose machine only. Contrary, the runtime evaluation has been performed twice, once on each machine.

The work and thus the evaluation is based on an LLVM 3.0 build with enabled assertions and disabled optimization. All source files have been converted by clang to LLVM-IR files, optimized by opt and ... TODO linked TODO

Table 5.1: The evaluation environment

	A	B
CPU	i5 M560	X5570
clock speed	2.67GHz	2.93GHz
smart cache	3MB	8MB
#cores	2	8
#threads	4	16
RAM	6GB	24 GB
LLVM	3.0 debug	3.0
OS	Arch	Gentoo R7

TODO picture of the chain

5.2 Compile Time Evaluation

The main part of the compile time evaluation aims to get quantitative results about valid and invalid sSCoPs. These results correspond with the applicability of this work, as they both outline how many regions can be taken into account now and which work is needed to increase this number. As mentioned earlier this part is mainly machine independent, since the quantitative results are. There is one case where compile time transformation can improve the program with no need of speculation at all. These cases are explained and evaluated separately in section 5.2.3.

5.2.1 Preperation

TODO -basicaa -indvars -mem2reg -polly-independent -polly-region-simplify -polly-prepare

5.2.2 Quantitative Results

5.2.2.1 SPEC2000

TODO why not all spec2000 benchmarks ?
TODO

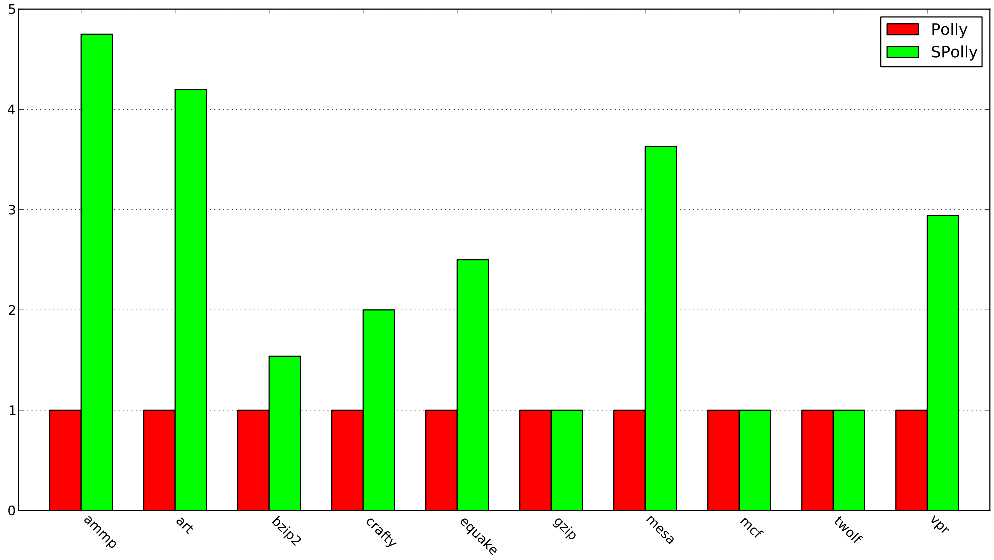


FIGURE 5.1: Numbers of valid and speculative valid SCoPs

5.2.2.2 Polybench 3.2

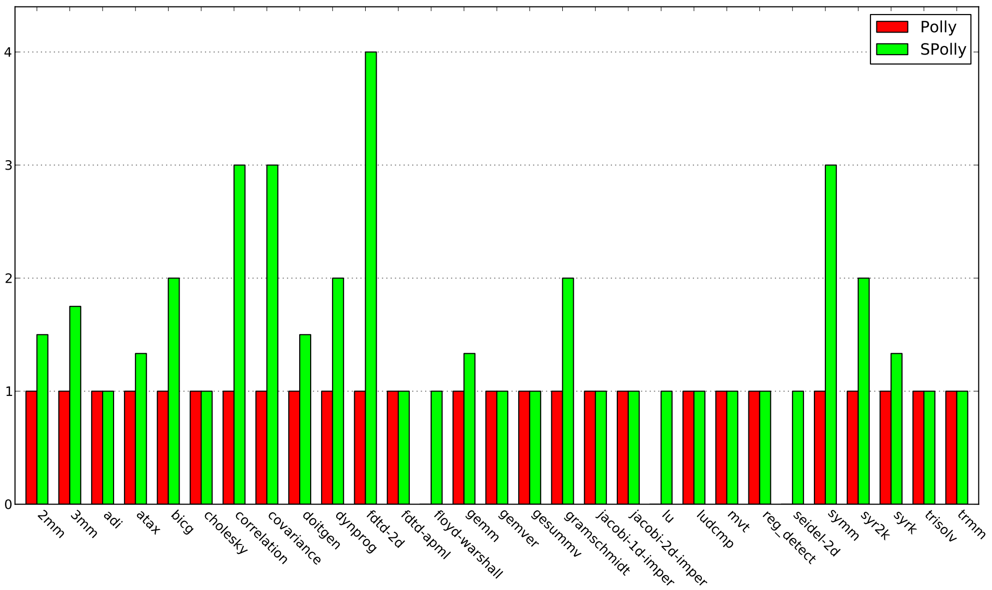


FIGURE 5.2: Numbers of valid and speculative valid SCoPs

TABLE 5.2: Results of running Polly and SPolly on SPEC 2000 benchmarks

Benchmark	**#instr**	**#simple regions**		**valid SCoPs**		**Avg detec. time**	
		initial	**prepared**	**Polly**	**SPolly**	**Polly**	**SPolly**
188.ammp	19824	205	208	12	45		
179.art	1667	66	66	5	16		
256.bzip2	3585	114	116	13	7		
186.crafty	25541	305	310	23	23		
183.equake	2585	70	71	10	15		
164.gzip	4773	92	95	6	0		
181.mcf	1663	33	33	0	0		
177.mesa	80952	816	832	94	247		
300.twolf	35796	679	716	6	0		
175.vpr	19547	319	329	17	33		

TODO numbers can be produced via /home/johannes/git/sambamba/testdata/spec2000/timeAnalysis.

5.2.2.3 Available tests

Alias tests

Invariant tests

5.2.3 Sound Transformations

As described earlier, region speculation collects violations within a SCoP and can introduce tests for some of them. There are cases when these tests will suffice to get a sound result, thus there is no need for a runtime system at all. Although this hold in respect to the soundness of a program, this does not mean performance will rise when these transformations are used.
TODO scores – heuristic / statistics

5.3 Runtime Evaluation

5.4 Problems

During the work with LLVM 3.0 and a corresponding version of Polly a few problems occurred. Some of them could not be reproduced in newer versions they were just be tackled with tentative fixes, as they will be resolved as soon as Sambamba and SPolly will be ported to a newer version. Others, which could be reproduced in the current trunk versions, have been reported and listed in figure 5.3. All bugs were reported with a minimal test case and a detailed description why they occur.

Table 5.3: Reported bugs

ID	Description	Status	Patch provided	Component
12426	Wrong argument mapping in OpenMP subfunctions	RESOLVED FIXED	yes	Polly
12427	Invariant instruction use in OpenMP subfunctions	NEW	yes	Polly
12428	PHInode use in OpenMP subfunctions	NEW	no	Polly
12489	Speed up SCoP detection	NEW	yes	Polly
TODO	add the others to bugzilla			
TODO	PollybenchC2 gemver , 2mm			

Chapter 6

Case Study

6.1 Matrix Multiplication

Matrix multiplication is a well known computational problem and part of many algorithms and programs, e.g. ammp or mesa. If the data size grows, the runtime may have crucial impact on the overall performance. Tiling, vectorization and parallel execution yield an enormous speedups as different approaches already showed[2, 16], still the question about applicability remains. A slightly modified source code would not be optimized at all, even if the computation has not been changed.

Measurement

This section will compare different implementations of a simple 2d matrix multiplication for a sample size of $1024 * 1024$ floats. Each example is executed 10 times and the geometric mean of the results (without the best and worst one) is computed. With this strategy it was possible to evaluate sequential executions with a maximal deviation of 5%. As this did not hold for parallel executions we decided to use the geometric mean of 100 executions here. All numbers are generated on the server described in table **??**. If not mentioned explicitly, the base algorithm stays the same for each case, so there is no hand made optimization involved. Furthermore no hand made optimizations are applied on intermediate results, thus the outcome is only dependent on the input and the presented options. To prevent false optimizations the result is dumped after each matrix multiplication.

Notes

Even if the STM embedded into the Sambamba framework does not provide a commit order yet, SPolly will speculatively execution loops in parallel. For the matrix multiplication example with proper inputs this is sound but it will obviously not reflect the overhead introduced by a commit order.

Case A

Listing 6.1 shows the matrix multiplication as used in many presentations and benchmarks. This case is quite grateful because the global arrays are distinct and fixed in size. Furthermore the loop nest is perfectly nested and all memory accesses can be computed statically. With this in mind the popularity of this case is hardly surprising, just as the outstanding results are.

```
float A[N][N], B[N][N], C[N][N];

void matmul() {
 int i, j, k;

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

Figure 6.1: Matmul case A

Case B

```
void matmul(float A[N][N],
            float B[N][N],
            float C[N][N]) {
  int i, j, k;

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

Figure 6.2: Matmul case B

Case B (see listing 6.2) is very similar to the previous one. The arrays are still fixed in size but now given as arguments. Even if the declaration is still global, common alias analysis can not prove the independence of each array anymore. Summarized aliasing between `A,B` and `C` is possible.

Case C and D

Cases C and D as presented in listings 6.3a and 6.3b are using pointers instead of fixed size arrays. This common practice to generalize the computation suffers from the same disadvantages as the second case. Common alias analyses will provide insufficient information to optimize the loop nest. In the context of this work case D is of special interest as it does not allow conclusive alias tests in front of the loop nest.

```
void matmul(float *A,
            float *B,
            float *C) {
 int i, j, k;

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

(A) Matmul case C

```
void matmul(float **A,
            float **B,
            float **C) {
 int i, j, k;

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

(B) Matmul case D

Figure 6.3: Matmul case C and D

Case E

The last case we will look at is the matrix multiplication within the ammp benchmark. It is similar to case D but with hand made optimizations which could be performed by clang (and therefor by Polly and SPolly) too, if type based alias analysis is available. The particular differences between case D and E are the lifted index computations and the strength reduction on the innermost loop. A stripped version of the original source code is given in listing 6.4.

```
void matmul( a,b,c,n,m)
float a[],b[],c[];
int n,m; {
  int i,j,k,ioff,koff;

  for( i=0; i< n*n; i++)
    c[i] = 0.;
  for( i=0; i< n; i++) {
    ioff = i*n;
    for( j=0; j< n; j++) {
      koff = 0.;
      for( k=0; k<m; k++) {
        c[ ioff +j] += a[ioff + k]
                      *b[ j +koff];
        koff += m;
      }
    }
  }
}
```

Figure 6.4: Matmul case E, extracted from the ammp benchmark (SPEC 2000)

Results and Discussion

Regarding the results in table 6.1 both Polly and SPolly achieved speedups up to 79 and , respectively. Even if smaller input data will not reach such dimensions, the results attest SPolly to be nearly as effective as Polly but with much greater applicability. All

Table 6.1: Execution times for the different matrix multiplication examples. All data is provided in milliseconds for an input size of $1024 * 1024$ floats.

Optimizer	Case A	Case B	Case C	Case D	Case E	Options
gcc	9305	9130	9154			best of O1, O2, O3
clang	9268	9289	9180			best of O1, O2, O3
Polly	3718	”	”			isl, ts[1] 256
Polly	”	”	”			isl, ts 256, vectorized
Polly	1072	”	”			OpenMP
Polly	”	1080	1077			OpenMP, ia[2]
Polly	114	115	115			isl, ts 256, OpenMP, ia
Polly	331	328	334			isl, ts 32, OpenMP, ia
SPolly	114	114	115			isl, ts 256, OpenMP, rs[3]
SPolly	397	330	333			isl, ts 32, OpenMP, rs
SPolly	763	618	664			isl, ts 256, sp[4]
SPolly	313	316	312			isl, ts 32, sp

[1] tile size

[2] ignore aliasing

[3] replaced with parallel OpenMP version if checks were sound

[4] speculatively parallelized with Sambamba

Chapter 7

Conclusion

List of Figures

List of Tables

Appendix A

Case Study Source Code

All presented cases of the example matrix multiplication (see section 6.1) are embedded into a small template. Listing A.1 shows the template instanciated with the first matrix multiplication case (A). Apart from the main method and the matumul function time measurement and a correctness check are included.

```
#define N 1024

float A[N][N], B[N][N], C[N][N];

void matmul() {
  int i, j, k;

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        C[i][j] += A[k][i] * B[j][k];
}

void init_arrays() {
    int i, j;

    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            A[i][j] = (1+(i*j)%1024)/2.0;
            B[i][j] = (1+(i*j)%1024)/2.0;
            C[i][j] = 0;
        }
    }
}

double sum_array() {
    double sum = 0.0;
    int i, j;

    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
          sum += C[i][j];
        }
    }

    return sum;
}

int main() {
    double sum;

    int i;
    for (i = 0; i < 10; i++) {
      init_arrays();
      matmul();
      sum = sum_array();

      if (sum != 69920704991472.0)
        return 1;
    }

    return 0;
}
```

Figure A.1: Matmul case A (full)

Appendix B

SPolly As Static Optimizer Pass

Table B.1: Command line options to interact with SPolly

Command line option	Description
-enable-spolly	Enables SPolly during SCoP detection, (options containing spolly will not work without)
-spolly-replace	Replaces all sSCoPs by optimized versions (may not be sound)
-spolly-replace-sound	As spolly-replace, but sound due to runtime checks (iff sound checks can be introduced)
-spolly-extract-regions	Extracts all sSCoPs into their own sub function
-polly-forks=N	Set the block size which is used when polly-fork-join code generation is enabled
-enable-polly-fork-join	Extracts the body of the outermost, parallelizeable loop, performs loop blocking with block size N and unrolls the new introduced loop completely (one loop with N calls in the body remains)
-polly-inline-forks	Inline the call instruction in each fork

Table B.2: Brief overview of Polly's optimization options

Short or option name	Description
-polly-no-tiling	Disable tiling in the scheduler
-polly-tile-size=N[1]	Create tiles of size N
-polly-opt-optimize-only=STR	Only a certain kind of dependences (all/raw)
-polly-opt-simplify-deps	Simplify dependences within a SCoP
-polly-opt-max-constant-term	The maximal constant term allowed (in the scheduling)
-polly-opt-max-coefficient	The maximal coefficient allowed (in the scheduling)
-polly-opt-fusion	The fusion strategy to choose (min/max)
-polly-opt-maximize-bands	Maximize the band depth (yes/no)
-polly-vector-width=N[1]	Try to create vector loops with N iterations
-enable-polly-openmp	Enable OpenMP parallelized loop creation
-enable-polly-vector	Enable loop vectorization (SIMD)
-enable-polly-atLeastOnce	Indicates that every loop is at leas executed once
-enable-polly-aligned	Always assumed aligned memory accesses
-enable-polly-grouped-unroll	Perform grouped unrolling, but don't generate SIMD

[1] Not available from the command line

Bibliography

[1] Tobias Grosser. Polly - polyhedral optimization in llvm, 2011. URL http://polly.llvm.org.

[2] Tobias Grosser. *Enabling Polyhedral Optimizations in LLVM.* Diploma thesis, University of Passau, April 2011. URL http://polly.llvm.org/publications/grosser-diploma-thesis.pdf.

[3] Tobias Grosser, Hongbin Zheng, Raghesh A, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly - polyhedral optimization in llvm. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[4] Louis-Noël Pouchet. Polybench, the Polyhedral Benchmark suite, 2010. URL http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6): 101–113, June 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375595. URL http://doi.acm.org/10.1145/1379022.1375595.

[6] Riyadh Baghdadi, Albert Cohen, Cédric Bastoul, Louis-Noël Pouchet, and Lawrence Rauchwerger. The potential of synergistic static, dynamic and speculative loop nest optimizations for automatic parallelization. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA'10)*, Saint-Malo, France, June 2010.

[7] Benoit Pradelle, Alain Ketterlin, and Philippe Clauss. Polyhedral parallelization of binary code. *ACM Trans. Archit. Code Optim.*, 8(4):39:1–39:21, January 2012. ISSN 1544-3566. doi: 10.1145/2086696.2086718. URL http://doi.acm.org/10.1145/2086696.2086718.

[8] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[9] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, 2004. URL http://www.llvm.org.

[10] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, LNCS, Paphos, Cyprus, March 2010. Springer-Verlag. Classement CORE : A, nombre de papiers acceptés : 16, soumis : 56.

[11] Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR '93, Lecture Notes in Computer Science 715*, pages 398–416. Springer-Verlag, 1993.

[12] Louis-Noël Pouchet. PoCC - The Polyhedral Compiler Collection, 2009. URL http://www.cse.ohio-state.edu/~pouchet/software/pocc/.

[13] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, pages 23–30, College Station, Texas, October 2003. Springer-Verlag.

[14] A. RAGHESH. *A Framework for Automatic OpenMP Code Generation*. PhD thesis, INDIAN INSTITUTE OF TECHNOLOGY, MADRAS, 2011.

[15] Sven Verdoolaege. Integer Set Library, 2004. URL http://www.kotnet.org/~skimo/isl/.

[16] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. VMAD: an Advanced Dynamic Program Analysis & Instrumentation Framework. In M. O'Boyle, editor, *CC - 21st International Conference on Compiler Construction*, volume 7210 of *Lecture Notes in Computer Science*, pages 220–237, Tallinn, Estonia, March 2012. Springer. URL http://hal.inria.fr/hal-00664345.

[17] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. Sambamba: A runtime system for online adaptive parallelization. In Michael F. P. O'Boyle, editor, *CC*, volume 7210 of *Lecture Notes in Computer Science*, pages 240–243. Springer, 2012. ISBN 978-3-642-28651-3.

[18] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. Sambamba: A runtime system for online adaptive parallelization, 2012. URL http://www.sambamba.org.