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
Kevin Streit

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

Submitted: May 23, 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. It offers features like method versioning, speculation and runtime adaption. As an extension of the former one and with the capabilities offered by the latter one, SPolly can perform state-of-the-art loop optimizations and speculative but sound loop parallelization on a wide range of loops, even in general purpose benchmarks as the SPEC 2000 benchmark suite. In this context, candidate loops for speculative parallelization may contain non-computable data dependencies (including possible aliases) as well as observable side effects which prohibit parallelization at first. The speculative potential of such a candidate loop depends not only on the size and the trip count, but also on the execution probability of these dependencies and function calls. Summarized, SPolly detects promising candidate loops, speculatively parallelizes them and monitors the result to tackle possible misspeculation.

This thesis will explain under which circumstances speculation is possible, how static and dynamic information is 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 nearly all other matters. With this in mind it remains to say: Joshua, Ruben and Sarah; Thank you!

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. They were always accessible for questions and helped me in the best possible way during all stages of this work. Furthermore (and in my opinion even more important), they 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.

Last but not least, I owe special thanks to my supervising professor for the liberty he granted me. Even if it took some time to find (or to define) the topic, it was a great experience I would not like to miss.

Content

Dedicated to my father whose never ending patience is priceless.

Chapter 1

Introduction

Nowadays multi-core processors 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 multi-core processors are widely unused because neither programmers nor compilers may utilize their potential to the fullest. Even if legacy applications would be amenable to parallelization, it is unclear how to find and exploit their potential automatically. Apart from the retrieval, parallelism faces the same problems as sequential code does. Cache invalidation and subsequent 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 vary in their impact. 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.

Lately, techniques using the so called polyhedral model grow in popularity. The underlying model is a mathematical description of loop nests with their loop carried data dependencies. Optimal solutions in terms of, e.g., locality or parallelism can be derived using this model while it implicitly applies traditional optimizations such 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 therefore few cases.

1.1 Related Work

Research on parallelism and data locality is very popular nowadays, just like the polyhedral model to tackle these problems is. There are promising attempts, all using a non speculative polyhedral model. Yet, the wide range impact on general purpose code is still missing.

Tobias Grosser describes in his thesis[1] a speedup of up to 8x for the matrix multiplication benchmark, achieved by his polyhedral optimizer Polly[2]. He also produced similar results for other benchmarks of the Polybench[3] benchmark suite. Other publications on this topic[4–6] show similar results, but their evaluation is also limited to well suited benchmarks, e.g., linear algebra kernels as the ones in the Polybench benchmark suite. Admittedly, Polybench is well suited for comparative studies of these approaches, but it has less significance for general applicability.

Baghdadi et al.[5] revealed a huge potential for speculative loop optimizations as an extension to the former described techniques. They state that aggressive loop nest optimizations (including parallel execution) are profitable and possible, even though overestimated data and flow dependencies would statically prevent them. Their manually crafted tests also show the impact of different kinds of conflict management. Overhead and therefore speedup differs from loop to loop, as the applicability of such conflict management systems does, but a trend was observable. The best conflict management system has to be determined per loop and per input, but all can provide speedup, even if they are not that well suited for the situation.

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. It offers features like method versioning, speculation and runtime adaption. As an extension of the former one and with the capabilities offered by the latter one, SPolly can perform state-of-the-art loop optimizations and speculative but sound loop parallelization on a wide range of loops, even in general purpose benchmarks as the SPEC 2000 benchmark suite. In this context, candidate loops for speculative parallelization may contain non-computable data dependencies (including possible aliases) as well as observable side effects which prohibit parallelization at first. The speculative potential of such a candidate loop depends not only on the size and the trip count, but also on the execution probability of these dependencies and function calls. Summarized, SPolly detects promising candidate loops, speculatively parallelizes them and monitors the result to tackle possible misspeculation.

This thesis will explain under which circumstances speculation is possible, how static and dynamic information is 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 is organised as follows: 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 benchmark suite (Chapter 5). Chapter 6 presents a detailed case study on different versions of the matrix multiplication example and in the end a conlusion and ideas for future work are provided (Chapter 7).

Note

For reasons of 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

LLVM, previously known as the Low Level Virtual Machine, is a compiler infrastructure designed to optimize during compile time and link time. 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. As the LLVM framework is modular and can be extended easily, most of the state of the art analyses and optimization techniques are available from the beginning. Plenty of other extensions, e.g., Polly, can be added by hand.

Further Reading

- LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation (Lattner and Adve [7])
- `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 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.

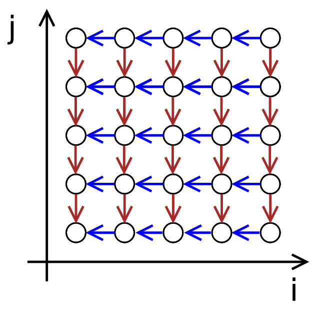
Within the polyhedral model the iteration space is represented as a $\mathbb{Z}$-polyhedron, in other words: a geometric object with flat surfaces existing in a space of any dimensionality. To do so, it is necessary that the iteration space can be described as solution set of affine inequalities (equation 2.1).

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

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 (graphical) representation in the polyhedral model(B). Both loops cannot be parallelized as there are dependencies, denoted by arrows, in the dimension they generate (labeled with their iteration variable). An optimization within the model could now transform the loops as indicated in Figure 2.2. The generated polyhedral model (2.2a) indicates that there are no dependencies in the `q`-dimension (dashed lines) which corresponds to the innermost loop in the new scheduling (2.2b). Thus, this transformation would allow parallel execution of the innermost loop but it would not increase data-locality. To do so, a second transformation could tile the inner loop as shown in Figure 2.2c. Afterwards vectorization could also be applied to create a version similar to the one in Figure 2.2d.

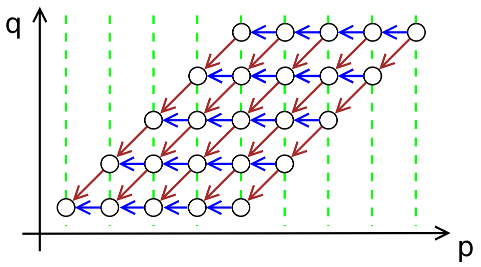
```
for (i = 1; i < ...; i++)
  for (j = 1; j < ...; j++)
    A[i][j] = A[i][j-1] * A[i-1][j];
```

(A) Simple loop nest



(B) Polyhedral representation of the loop nest presented in Figure 2.1a

Figure 2.1: An example loop nest with its polyhedral representation



(A) Polyhedral representation of the skewed loop nest presented in Figure 2.2b

```
for (p = 1; p < ...; p++) {
  parfor (q = ...; q < ...; q++)
    A[p][q] = A[p-1][q-1] * A[p-1][q];
}
```

(B) Skewed version of the loop nest presented in 2.1a (pseudo C code)

```
for (p = 1; p < ...; p++) {
  parfor (q = ...; q < ...; q += 256)
    for (q1 = q; q1 < ...; q1++)
      A[p][q1] =   A[p-1][q1-1]
                 * A[p-1][q1];
}
```

(C) Figure 2.2b after tiling

```
for (p = 1; p < ...; p++) {
  parfor (q = ...; q < ...; q += 256)
    for (q1 = q; q1 < ...; q1 += 16)
      A[p][q1:q1+16] =   A[p-1][q1-1:q1+16]
                       * A[p-1][q1:q1+16];
}
```

(D) Vectorized version of Figure 2.2c

FIGURE 2.2: Possible optimized versions of Figure 2.1a

Further Reading

- The Polyhedral Model is More Widely Applicable Than You Think (Benabderrahmane et al. [9])
- Loop Parallelization in the Polytope Model (Lengauer [10])
- A practical automatic polyhedral parallelizer and locality optimizer (Bondhugula et al. [4])
- PoCC - The Polyhedral Compiler Collection (Pouchet [11])
- Polyhedral parallelization of binary code (Pradelle et al. [6])
- Putting Polyhedral Loop Transformations to Work (Bastoul et al. [12])

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 representable in the polyhedral model. These representations are analyzed and transformed before they are again converted to code (here LLVM-IR). 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.

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 enable precise prediction of all memory accesses, it is necessary to restrict loop bounds and branch conditions to be affine expressions with respect to invariant parameters and surrounding iteration variables (see definition 2.2). The iteration variables themselves have to be canonical as stated in definition 2.3. Because dependencies are crucial for data flow information, SCoPs are not allowed to contain aliasing pointers at all. Furthermore, only affine memory access functions (as described above) will yield a precise result. Non-affine ones need to be overestimated when they are represented in the polyhedral model, but they are not forbidden per se. The last point concerns called function as only side effect free functions without any memory accesses 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. 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 ourselves to those containing at least one loop. The full list of restrictions on SCoPs is given in Table 2.1.

Table 2.1: Restrictions on SCoPs

- ▷ Only simple regions not containing the entry block of the function
- ▷ Only nested loops and conditionals
- ▷ Only branches with affine conditions
- ▷ Only affine accesses based on a constant (base) pointer
- ▷ No unsigned iteration variables [1]
- ▷ No aliasing pointers
- ▷ Only canonical PHI nodes and induction variables
- ▷ Instructions may not be used in PHI nodes outside the region[1]
- ▷ Only "readnone" function calls
- ▷ No alloca or cast instructions within the region
- ▷ Only affine trip counts for loops
- ▷ No PHI nodes in the region exit block
- ▷ *At least one loop contained*

[1] open for further work

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

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

Definition 2.2 (Affine Expressions).
An expression is called affine with respect to a set X, when the expression can be described as an affine transformation f (see definition 2.1) and the domain of f is restricted to X.

Definition 2.3 (Canonical Induction Variable).
An induction variable is canonical if it starts at zero and steps by one.

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. This part is of particular interest because only the accepted regions will be considered for polyhedral optimizations. Almost regardless of its content, any region could be given to Polly if the SCoP detection is properly instrumented, e.g., by disabling certain checks. An important consequence derived from this fact is: Utilizing the strength of Polly is possible in much more situations than intentionally implemented, provided that there is a mechanism to deal with possibly unsound results.

2.3.3 Loop Optimizations

By default, Polly consults the integer set library (isl) to compute the final schedule of a SCoP, but alternatively it is possible to use the more matured Pluto [13], too. Internally isl uses an optimized version of the algorithm proposed by Bondhugula et al. [4], which is in fact the base algorithm of Pluto. 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 of the loop nest, parallel code needs to be generated explicitly afterwards. Polly is capable of exploiting thread level parallelism using OpenMP [14] and data level parallelism using SIMD instructions (called vector instructions here). 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 latter one, namely vector code generation, will not only try to vectorize the innermost loop but also change the scheduling process in order to allow this in more cases.

Further Reading

- Enabling Polyhedral Optimizations in LLVM (Grosser [1])
- Polly - Polyhedral optimization in LLVM (Grosser et al. [2])
- Base algorithm of isl (Bondhugula et al. [4])
- `http://polly.llvm.org`
- `http://www.kotnet.org/~skimo/isl/`

2.4 Sambamba - A Framework for Adaptive Program Optimization

The Sambamba project is built on top of the LLVM compiler infrastructure and aims at adaptive and speculative runtime optimizations. It is easily extensible by modules consisting of a compile time and a runtime part. While they are conceptually independent, the compile time parts may store information which can be later on accessed by the runtime part. Apart from this "data store", Sambamba offers a way to maintain several (semantically equivalent) variants of a function. As there might be no order of any kind between those variants, it is admissible to think of them as somehow specialized versions. This method versioning allows to store conservatively and speculatively optimized versions in order to switched between them during runtime. Profiling combined with the method versioning system allows runtime interactions to explore more parallelism or minimize the overhead in case of misspeculation.

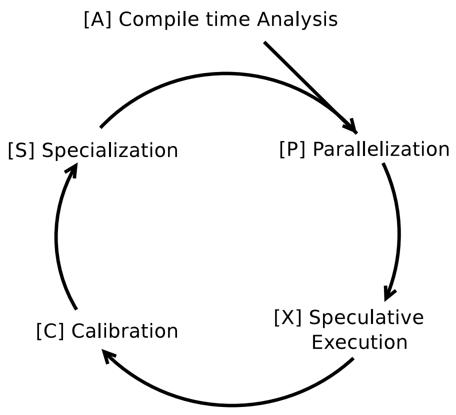


FIGURE 2.3: Conceptual stages in the Sambamba framework

Stages in the Sambamba framework

As illustrated in Figure 2.3 we can differentiate five conceptual 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 (see next Section).
- [**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 Software Transactional Memory

A software transactional memory system (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 within a certain code region. If, during the execution of this region, a memory address has been accessed by different threads and at least one of them was a writing one, all reading ones (during the concurrent execution and later on) may read illegal data. As a consequence, all resulting computations might be wrong. Such cases will be detected by the STM and conflicting threads might be forced to recompute their results. These recomputations are also called rollbacks. Different STM implementations will, for example, allow at least one thread to finish its computation before all the others might have to recompute theirs. Such implementations will maintain the liveness of a system.

As the described implementation only preserves liveness but no order between the different computations, there is an extension called commit order. 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 a determined order. For later argumentation we will assume the creation order of the threads. Just like the unordered version this one maintains liveness because there is always one thread at the front of the remaining ones.

Another possible extension for an STM system is a wait construct which is built on top of the commit order. Such a wait will cause the thread to pause until all preceding threads (in the commit order) have successfully committed their results. The section from the wait construct to the commit point forms a mutual exclusive area which can only be entered by one thread at a time and only if the preceding ones have been already successfully synchronized. Furthermore, the wait can only be passed by a thread if there is no conflict with all previous commits, otherwise the computation starts again at the beginning. Together, this guarantees that a thread passing the wait construct is able to successfully commit its results, in fact this thread will be the next committing one.

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

2.4.2 Parallel Control Flow Graphs

A parallel control flow graph (ParCFG) is a data structure used by the Sambamba parallelizer module to express parallel sections within an ordinary CFG. Each parallel section consists of an entry block called `piStart` and an exit block called `piEnd`. The `piStart` has an arbitrary number of successors, each denotes a (concurrent) transaction possibly guarded by the STM. All transactions end in the `piEnd` block after some arbitrary computation. To put it in another way, parallel sections represent fork-join parallelism where the `piStart` block forks transactions which are synchronized (or commited) at the corresponding `piEnd` block.

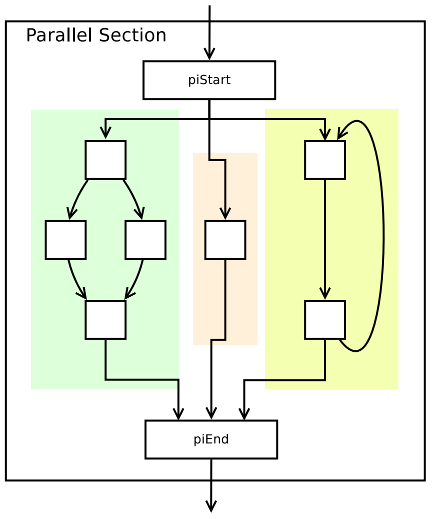


FIGURE 2.4: A parallel section with three transactions (colored regions)

Figure 2.4 shows such a parallel section with three highlighted transactions.

2.4.3 The Sambamba Parallelizer

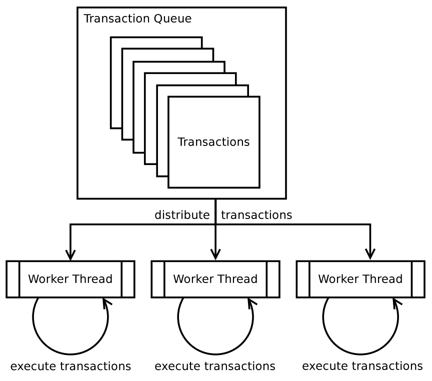


FIGURE 2.5: Illustration of the transaction queue with three worker threads

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.

Figure 2.5 presents a stylized transaction queue which is currently used to distribute the work among the worker threads.

Note that there is a new parallelizer module implemented by now which utilizes the "Intel(R) Threading Building Blocks" (TBB) framework [17]. As the evaluation of this work used the “old” implementation, the description given above might not fit the current state of Sambamba.

Further Reading

- Sambamba: A Runtime System for Online Adaptive Parallelization (Streit et al. [18])
- `http://www.sambamba.org`

Chapter 3

Concept

Conceptually, 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 latter 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 to accomplish both stated goals. We believe that these ideas and the gained knowledge is very valuable not only for future work on SPolly but also for other approaches facing similar situations.

3.1 Speculation in the Polytope Model

One great benefit of Polly and the underlying polyhedral model is to detect and model loop carried data dependencies. Regarding this fact it sounds plausible to use this ability in a speculative context as well. Unfortunately speculative extensions will compromise this ability because they will assume less dependencies when they weaken the requirements. As less dependencies may lead to more transformations, any speculative extension has to keep track of any change, e.g., to the iteration space, in order to preserve the semantics once the speculation fails. It is beyond the scope of this work to allow arbitrary polyhedral transformations with speculatively removed dependencies, so a more conservative way was chosen and implemented. At first, non-computable memory accesses, as they may arise from aliases or function calls, are overestimated.

The resulting polyhedral model is complete with regard to possible data dependencies, but could still reveal e.g., a scheduling for better data-locality or possible vectorization. Afterwards speculation is applied or in other words, the optimized loop nest is speculatively parallelized. Note here, that Polly can only change the scheduling based on the overestimated and therefore complete polyhedral representation of the loop nest, thus only in a sound way. A more detailed description on both steps is given in the following Sections, especially 3.3, 3.4 and 3.7.1.

3.2 Speculative Static Control Parts

As Polly is restricted to static control parts, there is a counterpart limiting the applicability of SPolly. This counterpart is called speculative static control part, or short sSCoP. Both are the central entities of the respective tool and they only differ in the restrictions on the underlying region. In contrast to Polly, SPolly allows arbitrary function calls, possible aliases and non-affine base pointers of memory accesses.

To illustrate the differences, Figure 3.1 presents three sSCoPs where each one validates one of the weakened restrictions, thus is a sSCoP but no SCoP. As a comparison to the restrictions on SCoPs (Table 2.1), a full list of sSCoP restrictions is given in Table 4.1 on page 30.

```
int g(int i) {
  return A[i];
}

void f() {
  int i;
  for (i=1; i<N; i++)
    A[i] = 3 * g(i-1);
}
```

(A) Speculative static control part violating the function call restriction on SCoPs

```
void f(int *A, int *B) {
  int i;
  for (i=1; i<N; i++)
    A[i] = 3 * B[i];
}
```

(B) Speculative static control part violating the aliasing restriction on SCoPs

```
void f(int **A) {
  int i, j;
  for (i=1; i<N; i++)
    for (j= 1; j<M; j++)
      A[i][j] *= 2;
}
```

(C) Speculative static control part violating the base pointer restriction on SCoPs

FIGURE 3.1: Speculative valid static control parts (sSCoPs), each violating one restriction of static control parts (see Section 2.3.1, especially Table 2.1)

3.3 Speculative Parallel Execution

Speculative loop parallelization is one of the two major purposes of SPolly. The challenge is to exploit as much parallelism as possible without restricting the applicability too much. In the context of this work, we limit ourselves to speculative valid SCoPs as described in the last Section. This restriction allows us to derive certain assumptions similar to the ones Polly requires. As a result, we are able to utilize main parts of Polly including the polyhedral representation. It is used to detect speculatively parallelizable loops without risking too much misspeculations. Before a sSCoP is parallelized the loop carried dependencies for each loop are computed as they would be in a non-speculative context. The used polyhedral model differs from the one created for an ordinary SCoP because it does not represent all possible memory accesses, thus it is incomplete. Even if aliasing pointers may access the same memory address, SPolly speculatively "removes" the dependencies between possibly aliasing base pointers when the polyhedral representation is created. Furthermore, all possible effects of a function call are ignored in this step. This includes memory effects as well as observable behaviour. Those differences prevent us from using the "normal" polyhedral transformations in order to increase data-locality, because they could alter the iteration order. Even if we use an extended STM to guard the execution, it is crucial to preserve the initial order for sound speculative parallelization. If not, a commit order could not prevent that the STM allows the wrong transaction to commit its results once a conflict is detected. Otherwise, if the initial order is maintained, hence all memory effects applied as in a sequential execution, only the observable behaviour of function calls can cause unsound results. To prevent this, the described wait construct is introduced right before the first call instruction on each execution path, thus functions will be executed only after all former changes have been synchronized. Because those wait constructs will create mutual exclusive (or non-parallel) sections, this speculative approach promises better results when the function calls are executed less frequently, e.g., like error handling code.

An example illustrating the need of a commit order and the preservation of the initial order is given in Section 3.7.

3.3.1 Sequential Consistency

As speculative execution implies the possibility of misspeculation, actions have been taken to preserve the semantic of each speculatively parallelized loop nest. As explained in the last Chapter, Sambamba uses an STM as conflict management system for speculative execution. In the current (not extended) implementation, the STM is not able to resolve conflicts between iterations executed in parallel in a proper way. If two transaction interfere the STM might allow the later one (in terms of the original iteration order) to commit its result before the former one could. As this one will be forced to roll back and recompute, it has to use the corrupted, but permanently written, data resulting from the committed computation. Because this does not preserve the semantics in each case, it is important to prevent such cases from happening. For most of the introduced speculations a commit order will suffice, but arbitrary function calls need the described wait construct too. It is also crucial that the initial order is only changed in the non-speculative setting, because otherwise the commit order would become futile and the described problem arises again. Later on (Section 3.4) we will describe how polyhedral transformations on sSCoPs can be applied in a sound way, even if they change the iteration order.

Assuming that all transactions are committed in a “proper order” and function calls are guarded by wait constructs, we state that the speculatively parallelized loop will always yields a sound result. To support this statement the following three paragraphs will present argumentations which successively back up the soundness of the described speculations. The first one also defines the “proper order” we assumed.

Aliasing pointers
Given a valid SCoP, but with possibly aliasing pointers (for example Figure 3.1b), SPolly may speculatively parallelize the loop and therefore execute iterations in parallel which access the same memory addresses. As the STM will by construction detect all memory access conflicts, it will also detect those emerging from aliases. We assume that such conflicts will be resolved based on a commit order which correlates with the iteration order of the parallelized loop. This is ensured, because the commit order (once it is available) is filled by unrolled iterations in their initial order (see Section 4.2.4 for details). We can conclude that in the case of a memory conflict, the transaction prior in the loop order is allowed to commit its changes, whereas the other conflicting (and later) ones are forced to roll back. Thus, the first transaction (corresponding to the first iteration) can always commit its results and the later ones will respect the iteration order of the loop in the case of a conflict. Inductively, the soundness of the speculative parallelization can be concluded if the iteration order of the loop nest was only transformed in a non-speculative context (e.g., not transformed at all).

Non-affine base pointers
Non-affine base pointers include loop dependent ones (as in Figure 3.1c), but also those returned by functions or computed in another non-affine way. In any case the soundness argument follows from the one given for aliasing instructions. Non-affine base pointers can be seen as start address of an unknown array. This array may alias with every other memory address, thus all accesses to this array may alias with all other memory accesses within the sSCoP. At this point the line of argument presented in the last paragraph fits again.

Arbitrary function calls
Speculative execution of arbitrary function calls causes three main problems you have to cope with. First there are memory accesses, but in our case the argumentation of the last two paragraphs holds here too. Apart from that, non-termination and other observable behaviour is possible. At the moment, we do not (seriously) analyze the called functions, thus we have to assume each of them may not terminate or cause observable effects. With this assumption speculative execution becomes impossible, we might want to allow such function calls. As speculative execution is not an option we have to stop it right before a call and ensure the whole program is in a state it would be in when executed non-speculative. To do so, the wait construct described in the context of the STM can be used. Placed right in front of the first call instruction on each path, the executing thread is only allowed to pass if all prior iterations (in terms of the non-speculatively changed iteration order) have been completed and the current state of the waiting thread is not conflicting with any of the previous commits. The wait construct will pause the thread until the first condition is fulfilled and then check if the second one holds too. If so, the thread might pass, otherwise the current transaction will be rolled back, but with the guarantee that this time no STM synchronization problem can occur.

3.3.2 Possible Misspeculations

Misspeculations as such occur when speculatively removed dependencies emerge during the execution. In the context of this work, also executed call instructions could be considered misspeculations. In both cases the current computation might be discarded, however, this is not always necessary. Even existing dependencies could allow speculative execution as long as the memory accesses follow the earlier stated "proper order". But since this depends only on the scheduling of the threads, we have to assume all misspeculations as fatal. The worst case would force each succeeding iteration, speculatively executed in parallel, to roll back and recompute, once another one committed its results. The processor time overhead would be quadratic in the number of parallel executed transactions. This includes the memory access logging, the synchronization and the performed rollbacks. As countermeasure, SPolly monitors the runtime of all speculatively parallelized sSCoPs and reverts the speculation for poorly performing ones. Future work could minimize the overhead of monitoring by instrumenting the parallelized sSCoP (or the STM) in order to notify SPolly if the number of rollbacks exceed a certain threshold.

3.4 Speculation Free Optimizations

As second objective, SPolly may enable sound polyhedral optimizations, even parallelization, on sSCoPs. Because there is no need for speculation at all, this kind of optimization could be applied even without the presence of Sambamba and the included STM. In the following we will introduce tests initially designed to lower the rate of misspeculations but sometimes powerful enough to completely rule out violations on sSCoPs. In such cases SPolly can use the code generation of Polly to exploit parallelism and the tests to preserve the semantics. Apart from these cases, it is possible to overestimate possible data dependencies in order to obtain a complete and sound polyhedral representation. Section 3.7.1 describes the latter approach while the next Section will cover the introduced checks.

3.5 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 = 0; 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) Observable call within a sSCoP

Figure 3.2: Examples for speculative valid static control parts (sSCoPs)

Region scores are used as a heuristic to decide whether or not an sSCoP is worth to speculate on, thus for which regions profiling and optimized versions should be created and as a result executed. As profiling 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 enormous impact on the actual region score. Additionally we are interested in execution paths within an sSCoP in order to predict how often e.g., irreversible instructions, may be executed. While such instructions, like calls to `printf`, may cause STM rollbacks during the parallel executions (after the wait construct failed synchronizing with previous commits), 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 Figure 3.2a to 3.2d are given in Table 3.1. Some of the constants in the presented scores are fixed, e.g., to weight trip counts, but others are dynamically computed from the given sSCoP size and the contained instructions. The variables can also be divided into those representing sSCoP invariant parameters (`N` and `M`) and symbolic values to take e.g., branch probabilities into account. Later on, both kinds will be instantiated with profiling data, e.g., the execution probability of a branches consequence or alternative, which results in an actual integer score.

Considering Figure 3.2c. Starting leftmost we can see the weighted loop trip count of 64 which is the quotient of the actual trip count 1024 and the constant 16. This ensures that loops with less then 16 iterations will not be scored positive. The outermost expression enclosed in parenthesis is the region score of the whole loop body. First, the 11 instructions of the always executed branch condition and afterwards the branch structure is taken into account. Two symbolic variables, namely "@if.then_ex_prob" and "@if.else_ex_prob", have been inserted and they will be instantiated with the profiled execution probabilities for the conditional. The instantiation for all symbolic variables will be done by the runtime part of the SPolly module, first with sample values, e.g., fifty percent execution probability for each branch, and later on with the real profiled execution probabilities and loop trip counts. Within the two branch subexpressions the numbers 5 and 7 denote the size and the 100 weights the execution probability which was given in per cent. For this example all instructions are of the same weight, hence the region score will yield the number of executed instructions once two probabilities are given. In fact it will only be one-sixteenth as the loop trip counts have been weighted.

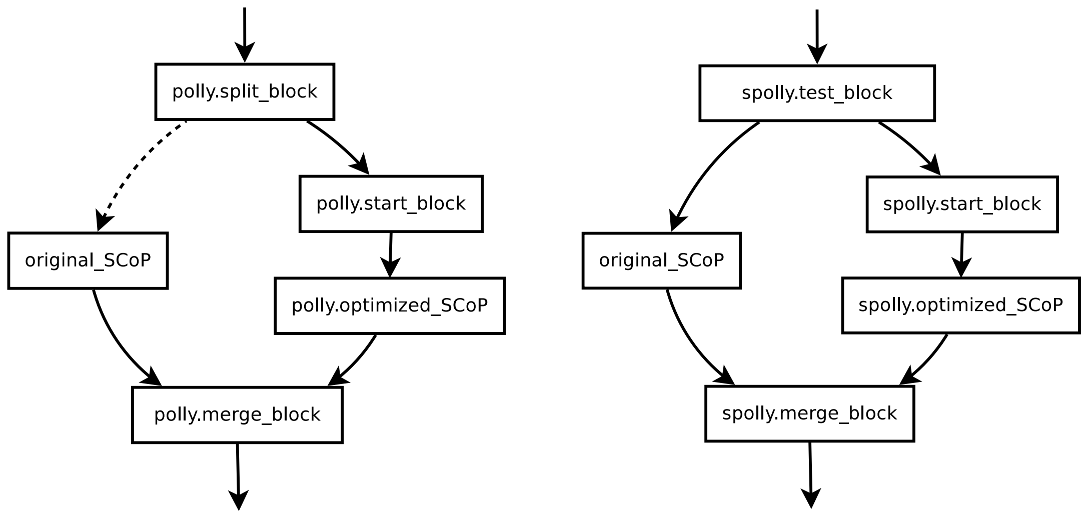
Table 3.1: Region scores for the sSCoPs presented in Figure 3.2

Figure	symbolic region score
3.2a	576
3.2b	$\%\mathtt{N}/16 * (7 + (10 * (\%\mathtt{M}/16)))$
3.2c	$64 * (11 + (7 * \text{@if.then\_ex\_prob}/100) + (5 * \text{@if.else\_ex\_prob}/100))$
3.2d	$\%\mathtt{N}/16 * ((6 * \text{@if.else\_ex\_prob}/100) + (-100 * \text{@if.then\_ex\_prob}/100))$

3.6 Introduced Tests

The already mentioned tests were initially designed as an attempt to keep the rate of misspeculations low. As such, they are used to refine region scores in the context of profiling, but it turned out that they can be of great use in optimized versions too. At the moment SPolly is capable of creating two different kinds of checks. If they rule out SCoP violations completely we may call them complete. For such cases the optimized versions with the introduced checks will always yield a sound result, hence there is no need for speculation at all.

To place the tests we take advantage of Pollys default behaviour which is to copy the optimized SCoP as an alternative to the original one. Figure 3.3a shows the CFG after Polly optimized a given SCoP. The dotted edge is not taken since the guard of the conditional in the split block is constant true. In the SPolly version this guard is replaced by the actual test result as indicated in Figure 3.3b.



(A) CFG after optimization with Polly (B) CFG after optimization with SPolly

Figure 3.3: CFG produced by Polly and SPolly, respectively

3.6.1 Alias Tests

Testing for aliasing pointers in general would be quite expensive because there could be aliasing between every possible pair. As this includes also the loop dependent pointers we would need to compute them beforehand in order to compare them pairwise. With this in mind, we decided to test only sSCoP invariant pointers, once before the sSCoP is entered. If the test succeeds, thus no aliases are found, the optimized version is executed, otherwise, the unoptimized sequential one. At compile time the accesses for each base pointer are collected and marked either as possibly minimal, possibly maximal or not interesting access. Accesses are called possibly maximal (minimal) if there is no other access to this base pointer which is statically known to be greater (less) in terms of integer arithmetic. At runtime all possibly 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. If all base pointers are invariant in the SCoP the test is complete, thus aliasing can be ruled out for the sSCoP at runtime.

Figure 3.4 shows three arrays with their base pointers (bp) and the minimal (maximal) access, denoted by ma and Ma, respectively. Two of the six comparisons needed to rule out aliasing for these arrays are indicated by arrows. The other four also compare a minimal and maximal accesses for different base pointers. As shown, the minimal accesses might be at the base pointer but also in front of it. This is the case if the base pointer is e.g., not the beginning of an array.

Part A of Figure 3.5 shows the matrix multiplication example which will be discussed further in Chapter 6. Assuming that the arrays are statically flattened, the loop nest contains four accesses. Such flattening is done by the compiler if the array is multidimensional but of a statically known size (see Section 6.1). Within Polly the accesses are analyzed and matched to their base pointers, here `A`, `B` and `C`. SPolly will statically analyse them too, in order to find all possible minimal and maximal accesses. In this case both types can be statically reduced to a unique offset as shown in part B. Acc denotes the access as named in the source code, bp the base pointer, ma the minimal access and Ma the maximal one. Comparing them will rule out aliasing for the loop nest completely because no base pointer is loop dependent. The last part (C) shows the actual alias test (or the comparison chain) SPolly introduces and the result which is used to choose whether or not to execute the speculatively optimized version.

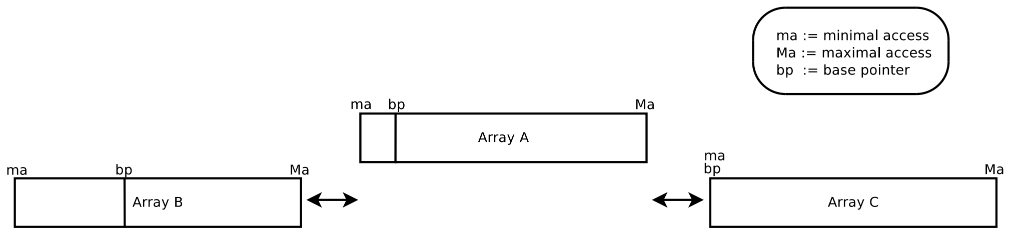


FIGURE 3.4: Alias test from a birds eye view

```
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];
    }
  }
}
```

(A) sSCoP with possibly aliasing accesses (I1 to I4)

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

(B) 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 no_alias_found = ab && ac && bc;
```

(C) Introduced alias test compare chain

Figure 3.5: Alias tests concept and example

3.6.2 Invariance Tests

Apart from alias tests, SPolly may introduce invariance tests if there are possibly invariant variables and function calls within an sSCoP. The key idea is is to monitor possible changes to the variables during the execution of the sSCoP. Even if these checks are not complete, they may give hints on dependencies between iterations. During the execution of the profiling version the test results are gathered and then used to influence the region score as the alias test results do. Initially designed as an exclusion criterion, the invariance tests also provide useful information about the variables. Such information could be used to create specialized sSCoP versions in the future.

Figure 3.6a gives an example of an sSCoP for which invariance tests can be introduced and 3.6b shows the modified source. Note here that changes to the variable `c` during the last iteration are not monitored.

```
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 possibly invariant variables

```
int c;

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

(B) Loop nest with invariance tests

Figure 3.6: Invariance test introduced by SPolly

3.7 Non-Computable Dependencies

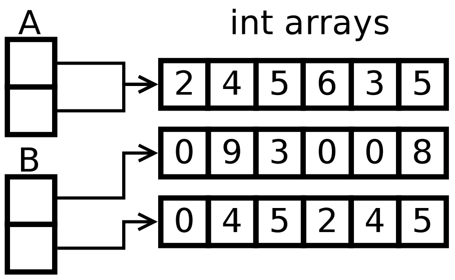
Ruling out may-aliases due to checks was already described earlier, but the approach is not feasible in every situation. Assuming the example in Figure 3.7a we may check if `A` and `B` alias in front of the loop nest, but every array `A[i]` and `B[i]` may also alias. Introducing tests within the loop nest would not only cause additional overhead every iteration but also prohibit the speculative optimization and parallelization of the outer loop. At the moment, SPolly is not capable of detection and deciding if such a proceeding could be profitable, but instead it tries to optimize and speculate on the whole loop nest or to be more precise, on the maximal valid sSCoP. As mentioned earlier, overestimating such data dependencies could allow sound polyhedral transformations before SPolly will speculatively execute the loop nest in parallel. While further details on the overestimation are given in the next Section, we will now look at the example in more detail and explain why even an STM based approach can produce wrong results when no commit order is available. Assuming we would translate the loop to the pseudo parallelized version in Figure 3.7c with the corresponding ParCFG as illustrated by Figure 3.7d. An input such as the one in Figure 3.7b could now produce wrong results even if an STM is used. In the speculatively parallelized versions two transactions (threads) would work on the same int array because `A[0]` and `A[1]` are equal. As both transactions read and write the same locations (`A[0][0]` to `A[0][1023]`), the STM will detect a conflict when the second transaction commits its results. As transaction 1 (see comment in Figure 3.7c) might commit after transaction 2 when no commit order is present, the STM will force a rollback of transaction 1 once the memory conflict is detected. Unfortunately, this will not yield a proper result as the second transaction already altered the state of `A[0] = A[1]` permanently. Introducing a commit order will always ensure the original iteration order for STM commits. In this example only transaction 2 could be forced to rollback and the overall result would always be valid.

3.7.1 Overestimating Dependencies

Enabling polyhedral optimizations for a wider range of programs is possible, even if no precise data dependency information can be computed statically. In the case of non-affine memory accesses Polly assumes all possible addresses, reachable from the base pointer, to be accessed. For non-computable dependencies arising from possible aliases (which includes loop dependent base pointers) SPolly uses a similar, conservative approach. When the polyhedral description is computed, dependencies between all possible aliasing base pointers are added. Even if this restricts transformations, there is the chance to allow some. Loop nests could be for example tiled, despite the fact that there are aliases

```
void f(int **A, int **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] + 1);
    }
  }
}
```

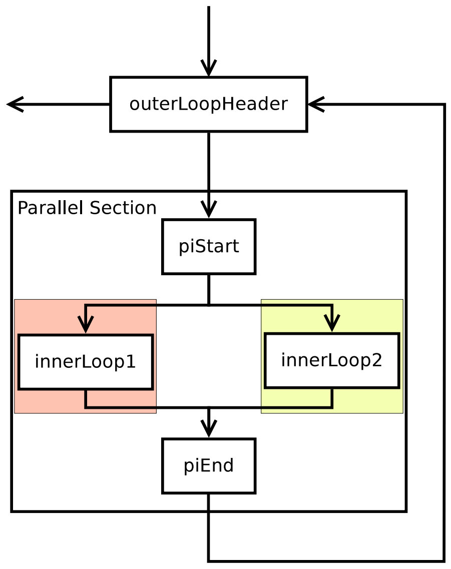
(A) Loop nest with non-computable dependencies



(B) Possible input situation for Figure 3.7c

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

(C) Loop nest 3.7a pseudo parallelized



(D) ParCFG for Figure 3.7c

FIGURE 3.7: Example for non-computable dependencies and a violating input

between used base pointers. It is worth to empathize that these transformations are completely speculation free because they are based on an overestimated, hence complete, polyhedral model.

3.8 Method Versioning

Method versioning is a key feature of the Sambamba framework, but not fully implement yet. However, SPolly is already capable of generating a profiling and an optimized sSCoP version. As Sambamba will take care of profiling and dispatching versions in the near future, SPolly can be extended to create multiple other sSCoP versions without much effort. In this context profiling data will become even more valuable. Specialized sSCoP versions e.g., where we assume constant instead of variable loop bounds, are one possible usage. This would not only speed up the execution, but also improve transformations as they would benefit from this additional knowledge. Other points of interest might be the various options which can be passed to Polly. It is not clear which e.g., tiling size or fusion strategy is the best, neither for different sSCoPs nor for different systems, hence SPolly could create more than one optimized version. Once the dispatcher system of Sambamba is fully implemented, the execution times of sample runs would provide enough information to choose a version and simultaneously report overhead through misspeculations. As most of this needs to be implemented in the scope of Sambamba first, SPolly is currently limited to the default configuration of Polly.

A full list of options which can be passed to Polly is given in the appendix B.

Chapter 4

Technical Details

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.

4.1 SPolly in a Nutshell

During **compile time** the main goal of SPolly is to simplify the runtime part, by reducing runtime overhead through preprocessing and even static method versioning. First the SCoP detection tries to find all valid regions within the given LLVM-IR (according to the SCoP definition; Table 2.1). But instead of rejecting a region once a restriction is violated, the region speculation is asked how to proceed. Rejection causes we want to speculate on namely aliasing, non-affine base pointers and function calls, are gathered by the region speculation, but fully transparent to the SCoP detection. This proceeding allows to find all violations within a region and to treat valid and speculatively valid ones nearly the same. After all speculatively valid SCoPs 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

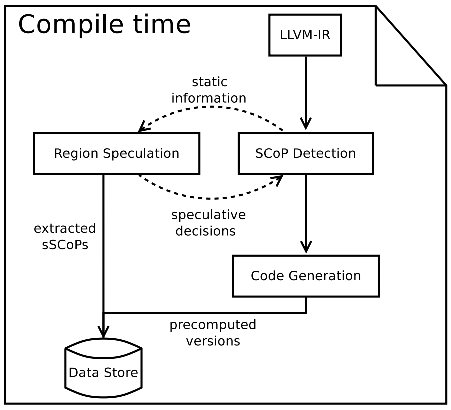


FIGURE 4.1: Draft paper: SPolly at compile time

others are currently only extracted. This means they are replaced by calls to functions only containing the speculatively valid region. At the moment, no pre computation takes place and only sSCoPs with complete alias tests are considered to be optimized and exchanged immediately. Future work on SPolly could change this in order to reduce the workload during runtime.

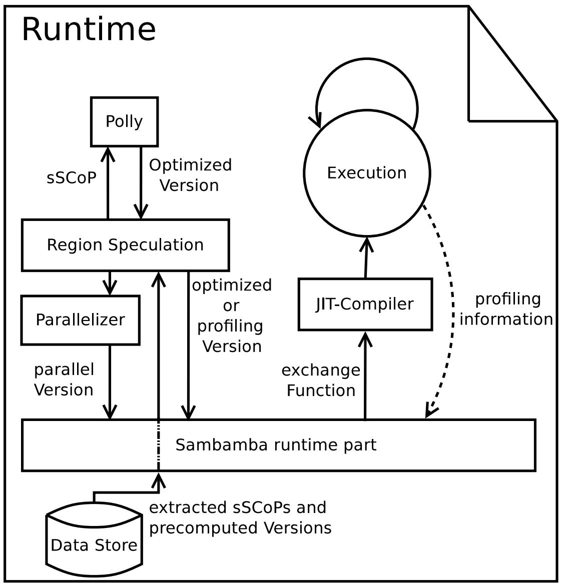


FIGURE 4.2: Draft paper: SPolly at runtime

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. It would be possible to restrict this to the highest rated sSCoPs only, but as the creation is cheap, only a few additional instructions without traversing the code again, and the execution overhead for most of them is nonexistent it is feasible to do so for rather bad ranked sSCoPs as well. 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 for the time consumption these values will affect the rating of the sSCoP which again is used to identify promising sSCoPs.

4.2 Speculative Polly

You could 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. Pollys SCoP detection was chosen to serve as a 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 was nicely illustrated in the thesis of Tobias Grosser[1], we extended the Figure to capture the changes introduced by SPolly, too (see Figure 4.3). In comparison to the original one, the region speculation, the unrolling backend and the splitting backend have been added.

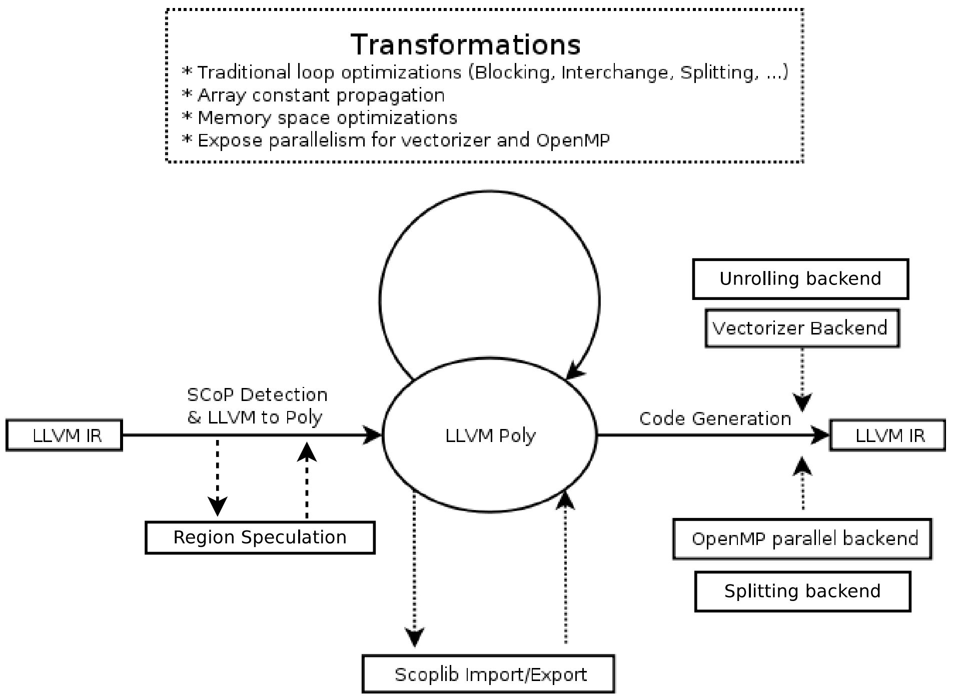

Transformations
* Traditional loop optimizations (Blocking, Interchange, Splitting, ...)
* Array constant propagation
* Memory space optimizations
* Expose parallelism for vectorizer and OpenMP
Unrolling backend
Vectorizer Backend
SCoP Detection & LLVM to Poly
Code Generation
LLVM IR
LLVM Poly
LLVM IR
Region Speculation
OpenMP parallel backend
Splitting backend
Scoplib Import/Export

Figure 4.3: SPolly - The Architecture
(Based on "Polly - The Architecture"[1])

4.2.1 Region Speculation

The region speculation of SPolly has several tasks to fulfill. The first one includes the communication with Polly or, to be more precise, with the SCoP detection. Each region analyzed by the SCoP detection needs to be considered as possibly speculatively valid SCoP, thus all information exposed during the detection is stored. If the region contains a sSCoP violation, listed in Table 4.1, or if it is without any validation, the information is discarded. In the latter case, the region is a valid SCoP and will be handled by Polly and in the former one it is neither a SCoP nor an sSCoP, hence we cannot optimize it any further. For all others regions a new sSCoP is created and initially validated. This validation mainly computes information needed for later transformations but it may discard the sSCoP too. This is the case if violating function calls, e.g., a `printf`, occur on every execution path. As the *Concept* Chapter described how to handle such cases in general, we may include even those in later work. For now, violating calls are only allowed in conditionals as such cases provide much more speculation potential.

Table 4.1: Restrictions on sSCoPs

- ▷ Only simple regions not containing the entry block of the function
- ▷ Only nested loops and conditionals
- ▷ Only branches with affine conditions
- ▷ No unsigned iteration variables
- ▷ Only canonical PHI nodes and induction variables
- ▷ No alloca or cast instructions within the region
- ▷ Only affine trip counts for loops
- ▷ Only guarded "violating" function calls
- ▷ No PHI nodes in the region exit block
- ▷ *At least one loop contained*

4.2.2 Region Scores

Initial efforts to create region scores in order to rank sSCoPs did not use any kind of memory, thus the whole region was analyzed every time new information was available. To avoid these 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 creation of the score expression the region needs to be traversed to find all used parameters and branches.

4.2.3 sSCoP Extraction

The sSCoP extraction was designed to simplify the method versioning for functions containing several speculatively valid SCoPs. It creates a new sub function for every sSCoP and inserts a call in the respective place. 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. On the other hand it is the first step to multiple specialized versions, e.g., with constant instead of variable loop bounds. Another possible benefit of this strategy is that the transformations applied to an sSCoP (or SCoP) would not affect other code regions anymore.

4.2.4 Code Generation

As part of the extension of Polly two new code generation backends were added. Apart from sequential, vectorized and OpenMP code generation, SPolly is capable of creating easily parallelizable loop shapes with regard to fork-join parallelism. The first one is an unrolled and blocked loop as denoted by the translation from Figure 4.4a to Figure 4.4b. A parallel version of this loop will execute the N blocked and unrolled iterations concurrently, while there might be remaining ones which are executed sequentially after the loop. Figure 4.4c shows the outcome if the other new backend is used. This time the whole loop nest was fragmented into N "identical" ones but with adjusted loop bounds. The main difference between these two approaches is the workload per transaction. While the former one will only execute one loop iteration at a time, the latter one will execute one N-th of the iteration space (for N $\ll$ loop trip count). The main similarity is the simple CFG structure, which is easily convertible to a ParCFG. Figure 4.5 and 4.6 illustrates such transformations for the blocking backend and the splitting backend respectively.

Since STM transactions will always cause synchronization overhead, the second approach has certain advantages over the first one; e.g., if N is chosen as the number of available threads, the overhead is minimal, compared to the achieved parallelism. This is the "best case" as long as no misspeculation takes place. If there are speculatively removed dependencies the former approach might perform better, but in the extrem case where all iterations depend on former ones, the second one has an advantage again. As this holds only in theory, Section 5.4 compares the maximal number of rollbacks for both new code generation backends in more detail.

4.3 Profiling for Sambamba

Sambamba, as research project under heavy development, was not capable of any kind of profiling at the beginning of this work. By now, there are two profilers available. The first one, implemented by the authors of Sambamba, is used for exact time measuring, while the other one was created considering the needs of SPolly. It is capable of collecting execution counts for arbitrary program locations and offers an interface to measure and compute branch probabilities as well as loop trip counts. Apart from profiling executions it can also be used to monitor variables at given program locations, useful in the context of specialization as it might provide hints on "quasi constant" parameters. If such parameters are encountered, specialized versions may yield additional speedups.

```
void f() {
  int i;
  for (i = Lb; i < Ub; i += Str) {
    LoopNest(i);
  }
}
```

(A) Initial loop nest

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

(B) Figure 4.4a after N iterations have been blocked and unrolled (Unrolling backend)

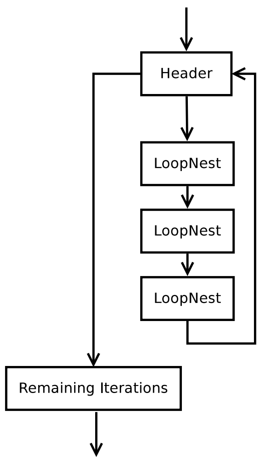
```
void f() {
  int i;
  Part  := (Ub - Lb) / N;
  Ch    := max(D, Str);
  Chunk := Ch - 1;

  // First chunk
  NUb   := min(Ub, Lb + Chunk);
  for (i = Lb; i < NUb; i += Str) {
    LoopNest(i);
  }

  ...

  // N-th chunk
  Lb  := Lb + Ch;
  NUb := min(Ub, Lb + Chunk);
  for (i = Lb; i < NUb; i += Str) {
    LoopNest(i);
  }
}
```

(C) Figure 4.4a after loop splitting (Splitting backend)

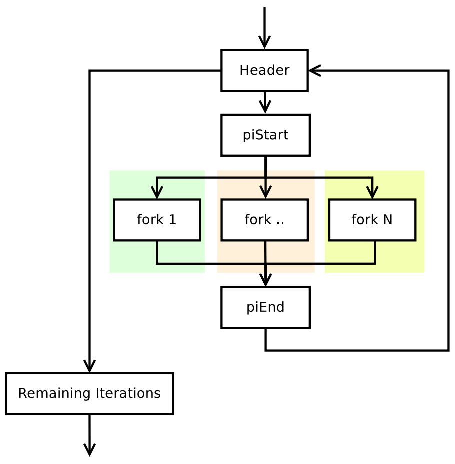
FIGURE 4.4: Outcome of the new code generation backends

TABLE 4.2: Reported bugs

Bug-ID	Description	Status	Patch provided	Component
12426	Wrong argument mapping in OpenMP subfunctions	FIXED	partially	Polly
12427	Invariant instruction use in OpenMP subfunctions	NEW	yes	Polly
12428	Wrong PHInode use in OpenMP subfunctions	NEW	no	Polly
N/A	Wrong computations of STM guarded code	FIXED	no	Sambamba

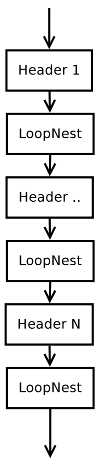


(A) CFG after blocking and unrolling

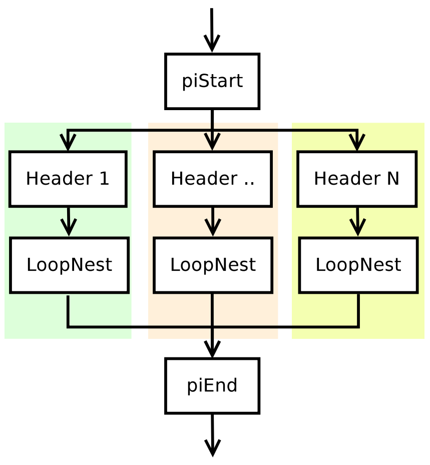


(B) ParCFG created from Figure 4.5a

FIGURE 4.5: Forked (Par)CFG after the blocking backend was used



(A) CFG after loop splitting



(B) ParCFG created from Figure 4.6a

FIGURE 4.6: Forked (Par)CFG after the splitting backend was used

Chapter 5

Evaluation

SPolly is based on Sambamba and Polly, both research tools under heavy development. It is hardly surprising that during the implementation of SPolly, but also during the evaluation, problems occurred. Even if some of them could be fixed without incurring too much delay, there are still open problems. On the one hand there are reported bugs (see Table 4.2) which will be resolved in future releases. And on the other hand there is the current STM implementation in the Sambamba framework which does neither provide a commit order nor a wait construct, as most general STM implementations do not. Because of those problems it is currently not possible to handle arbitrary general purpose code. Even non-speculative parallelization for SPEC benchmarks will certainly cause Polly to crash or to produce invalide code. Nevertheless, we provide quantitative results on the applicability of SPolly on SPEC 2000 benchmarks as well as a discussion on the shape and characteristics of the detected sSCoPs. For performance data we refere to the next Chapter as it contains a detailed evaluation for several versions of the matrix multiplication example.

TABLE 5.1: The evaluation environment

	Machine
CPU	X5570
clock speed	2.93GHz
L2 cache	8MB
#cores	8
#threads	16
RAM	24 GB
LLVM	3.0
OS	Gentoo R7

Information about the environment used for the evaluation but also for the case study is given in Table 5.1.

5.1 Quantitative Results

Speculative SCoPs are an extension to ordinary SCoPs, thus we may compare them in terms of quantity and the amount of parallelism we can exploit. Absolute numbers together with general information about the benchmarks are listed in Table 5.2, while a graphical representation is given by Figure 5.1. First note that the **SCoPs** (**sSCoPs**) column in Table 5.2 contains the number of detected SCoPs (sSCoPs) as well as the number of *parallelizable* SCoPs (sSCoPs). The other columns provide the number of instructions (**#instr**), functions (**#func**) and simple regions (**#sr**) for a particular benchmark.

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

Benchmark	**#instr**	**#func**	**#sr**	**SCoPs**		**sSCoPs**	
188.ammp	19824	161	208	16	*12*	73	*34*
179.art	1667	22	66	5	*0*	17	*5*
256.bzip2	3585	59	116	31	*22*	76	*46*
186.crafty	25541	105	310	42	*30*	67	*52*
183.equake	2585	24	71	10	*9*	40	*36*
164.gzip	4773	59	95	9	*5*	10	*6*
181.mcf	1663	24	33	2	*2*	2	*2*
177.mesa	80952	769	832	107	*79*	276	*208*
300.twolf	35796	166	716	8	*4*	39	*20*
175.vpr	19547	294	329	17	*8*	56	*31*

Regarding only the amount of detected SCoPs and sSCoPs respectively, SPolly is able to handle 1.5 times more regions than Polly. Even if this does not correspond to amount of exploitable parallelism in general, it indicates that the applicability of SPolly is very much bigger on general purpose code. For the particular benchmarks, the number of parallelizable sSCoPs is actually 1.5 times greater than the number of parallelizable SCoPs, too. In this context a SCoP (or sSCoP) is considered as parallelizable if it contains a loop without loop carried dependencies (modulo the applied speculations; Section 3.3).

5.2 Detected sSCoPs

The "quality" of the detected sSCoPs is unfortunately not as desired; except three of them, all contain only a single loop and in the majority less than 30 LLVM-IR instructions. Even if this is comparable to the size of the matrix multiplication example, the loop trip counts are much smaller. Due to these facts, parallelization causes more overhead than speedup for most of them. Unsound tests (without an STM) substantiate this statement as the performance worsens when each sSCoP is speculatively parallelized. Even though parallelizing all sSCoPs has negative effects on the overall performance, the highest ranked sSCoPs are promising candidates for speculation after all. Examples for such high ranked candidates are given in Figure 5.2. Parts A and B present loops with statically unknown trip counts, while parts C and D have a statically known iteration counts (64 and 1024, respectively). All four loops contain possibly aliasing pointers, but only the last one also function calls. Note that the function calls in this sSCoP are not guarded by a conditional, thus SPolly would currently discard this sSCoP during the initial validation. As future work could include (basic) analysis on the called functions, the `exp` function called here could be classified as non-violating, hence there would be no need to discard this sSCoP. With or without such an analysis, it is worth to emphasize that SPolly detects the presented loops automatically and ranks them far beyond the average.

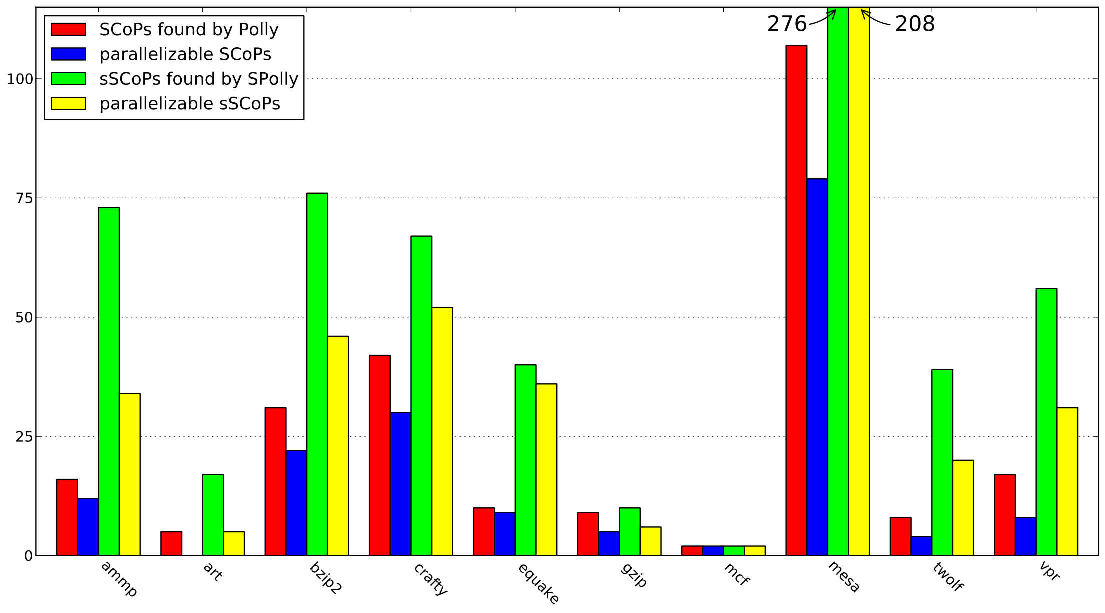


FIGURE 5.1: Quantity of detected and parallelizable SCoPs and sSCoPs

5.3 Sound Transformations

TABLE 5.3: Number of sSCoPs with complete checks

Benchmark	
188.ammp	6
183.equake	5
177.mesa	109
300.twolf	26
175.vpr	3

For sSCoPs with complete checks, as described earlier, there is no need for a runtime system, especially an STM, in order to allow sound optimization and parallelization. Within the SPEC 2000 benchmarks, SPolly detected 149 sSCoPs with complete checks (see Table 5.3), but attempts to compute performance data failed. In the case of e.g., 300.twolf, Polly created valid parallelized code for 3 of the 26 sSCoPs, hence the result is not very representative. As cases B and C of the matrix multiplication example are sSCoPs with complete checks it is worth to look at the case study results when the "-replace-sound" option is used. A discussion of these results is given in the third paragraph of Section 6.1.1. Also note here that the best result was achieved that way.

```
for (i=0;i<width;i++) {
  red[i]   = (GLubyte) (GLint) (red[i]   * rscale);
  green[i] = (GLubyte) (GLint) (green[i] * gscale);
  blue[i]  = (GLubyte) (GLint) (blue[i]  * bscale);
  alpha[i] = (GLubyte) (GLint) (alpha[i] * ascale);
}
```

(A) Benchmark: 177.mesa Function: copy_tex_sub_image

```
for (i=0;i<n;i++) {
  /* Cv = CfCt */
  red[i]   = PROD( red[i],   Rt[i] );
  green[i] = PROD( green[i], Gt[i] );
  blue[i]  = PROD( blue[i],  Bt[i] );
  /* Av = AfAt */
  alpha[i] = PROD( alpha[i], At[i] );
}
```

(B) Benchmark: 177.mesa Function: apply_texture

```
for (i=0;i<64;i++) {
  clear_mask[i]=Compl(Shiftr(mask_1,i));
  clear_mask_rl45[i]=Compl(Shiftr(mask_1,init_l45[i]));
  clear_mask_rr45[i]=Compl(Shiftr(mask_1,init_r45[i]));
  clear_mask_rl90[i]=Compl(Shiftr(mask_1,init_l90[i]));
  set_mask[i]=Shiftr(mask_1,i);
  set_mask_rl45[i]=Shiftr(mask_1,init_l45[i]);
  set_mask_rr45[i]=Shiftr(mask_1,init_r45[i]);
  set_mask_rl90[i]=Shiftr(mask_1,init_l90[i]);
}
```

(C) Benchmark: 186.crafty Function: InitializeMasks

```
for( i2 = 1 ; i2 <= 1023 ; i2++ ) {
    table1[ i2 ] = exp( -(double) i2 / 8.0 ) ;
    table2[ i2 ] = exp( -(double) i2 / 8192.0 ) ;
    table3[ i2 ] = exp( -(double) i2 / 8388608.0 ) ;
}
```

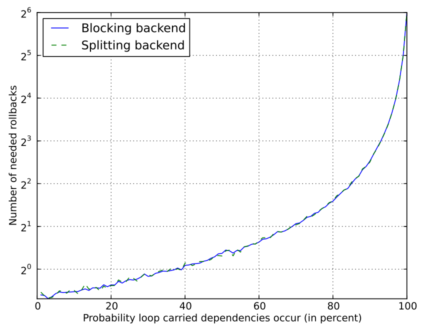
(D) Benchmark: 300.twolf Function: utemp

FIGURE 5.2: Selection of the highest ranked sSCoPs within the SPEC benchmarks

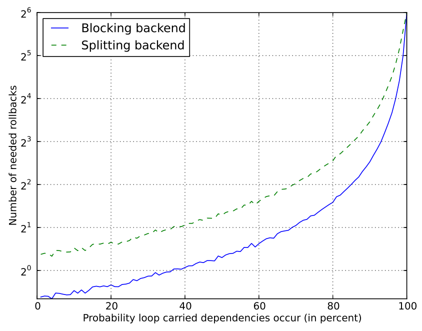
5.4 Backend Evaluation

In the last Chapter we described the two new backends added to Pollys code generation and we mentioned that both have theirs eligibility. To substantiate this statement we compared the number of rollbacks for different loop trip and thread counts when dependencies between iterations occur. For trip counts of 64 and 1024 we assumed 2 and 16 executing threads respectively. The dependencies were generated and distributed randomly with a probability from 0 to 100 percent and last over 1,2 or 16 iterations. Figures 5.3 and 5.4 provide the curves for the geometric mean of 1000 iterations without the best and worst 10%. The insight gained is the relation between workload per transaction (in terms of iterations) and possible amount of needed recomputations. The solid line denotes always the blocking backend where each transaction computes one iteration at a time and the dashed lines denote the splitting backend where a transaction computes $\frac{\#iterations}{\#threads}$ iterations. In the former case, only one iteration has to be recomputed once a dependency occurs, but in the latter case it would be $\frac{\#iterations}{\#threads}$. Regarding parts A and C of the Figures 5.3 and 5.4, we can see that dependencies over only one iteration do affect both versions nearly the same. This is hardly surprising as the probability to recompute a transaction in the splitted version is decreased by the factor $\frac{\#iterations}{\#threads}$ and once it is rolled back exactly the same number of iterations have to be recomputed. For such cases, the preferred version would be the splitted one, because the additional overhead due to e.g., synchronization, is far less (again by a factor of $\frac{\#iterations}{\#threads}$). The situation changes as the dependencies last over more iterations, as indicated by the parts B and D. The overhead for the blocked and unrolled version stays the same but the splitted version suffers from more rollbacks as now each dependency forces a transaction to recompute. The factor decreasing the probability was $\frac{\#iterations}{\#threads}$, hence the curves in the Figures 5.3b and 5.4b are, at the beginning, shifted up by 2 and the ones in Figures 5.3d and 5.4d by 16. As the asymptotic behaviour of both versions is the same, the graphs will always converge with increasing dependency probability. In the presented examples, this behaviour only matters after the probability for dependencies reached about 90%. Beyond this limit, the distance between the two graphs is rapidly decreasing.

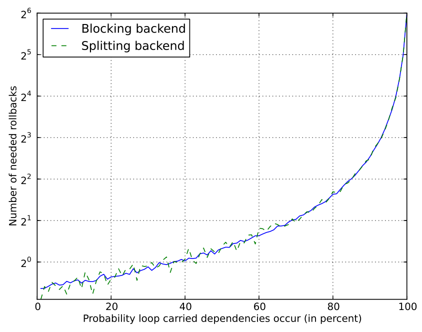
Summarized, both versions have advantages and drawbacks which needed to be considered for speculative execution. A blocked and unrolled version might not provide the desired speedup but nevertheless hints on the probability of occurring dependencies and even their “size”. Such information could suggest a second parallelized version e.g., with more workload per transaction, if it would exceed the first one in terms of speedup but with a acceptable probability of overhead.



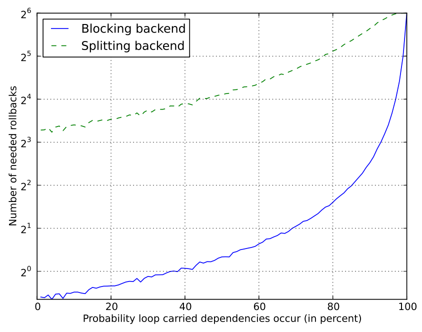
(A) 64 iterations, 2 threads, dependencies over 1 iteration



(B) 64 iterations, 2 threads, dependencies over 2 iteration

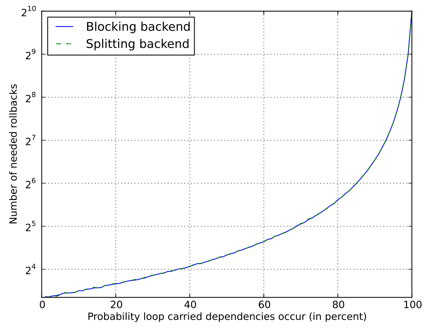


(C) 64 iterations, 16 threads, dependencies over 1 iteration

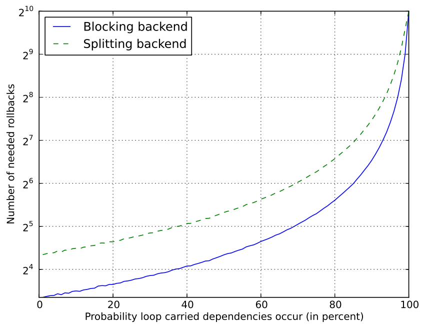


(D) 64 iterations, 16 threads, dependencies over 16 iteration

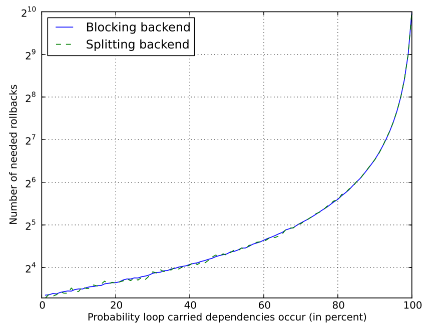
Figure 5.3: Backend evaluation with 64 iterations



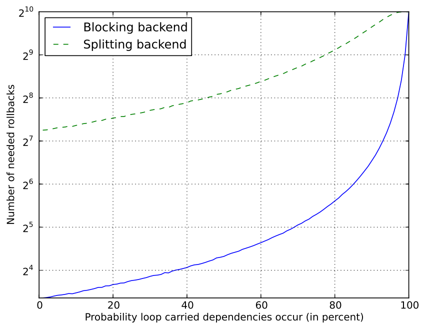
(A) 1024 iterations, 2 threads, dependencies over 1 iteration



(B) 1024 iterations, 2 threads, dependencies over 2 iteration



(C) 1024 iterations, 16 threads, dependencies over 1 iteration



(D) 1024 iterations, 16 threads, dependencies over 16 iteration

Figure 5.4: Backend evaluation with 1024 iterations

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 within the SPEC 2000 benchmark suite. If the data size grows, the runtime may have crucial impact on the overall performance. Tiling, vectorization and parallel execution yield an enormous speedup as different approaches already showed[1, 19], still the question of applicability remains. A slightly modified source code may not be optimized at all, even if the computation has not been changed.

Measurement

This Chapter will compare different implementations of a simple 2d matrix multiplication for a sample size of $1024 * 1024$ floats. Each example is executed 100 times and the geometric mean of the results (without the best and worst 10%) is computed. All numbers are generated on the machine described in Table 5.1. If not mentioned explicitly, the base algorithm remains 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 validated after each iteration.

Notes

Even if the STM embedded into the Sambamba framework does not provide a commit order yet, SPolly will speculatively execute loops in parallel. For the matrix multiplication example with proper inputs this is sound. Furthermore, SPolly is able to create sound versions for the cases A to C because the alias tests for these cases are complete.

Case A

Figure 6.1 shows the matrix multiplication as used in many presentations and benchmarks. This case is quite grateful because the global arrays are disjoint 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: Matrix multiplication 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: Matrix multiplication case B

Case B (see Figure 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.

Cases C and D

Cases C and D as presented in Figures 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 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*N+j] += A[k*N+i] * B[j*N+k];
}
```

(A) Matrix multiplication case C

```
void matmul(float **A,
            float **B,
            float **C, int 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];
}
```

(B) Matrix multiplication case D

FIGURE 6.3: Matrix multiplication 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 manual optimizations which could be performed by clang (and therefore by Polly and SPolly) too, if type based alias information 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 Figure 6.4. Even if the algorithm stays the same, the accesses functions are substantially altered. Polly will not recognize them as affine anymore, hence they will be overestimated. The resulting polyhedral representation has loop carried dependencies between all iterations in every loop. Polly cannot and SPolly will not parallelize a loop based on these information. Future work on SPolly could allow to distinguish between encountered and overestimated dependencies. This would allow more speculation but it remains to determine when this is useful.

```
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: Matrix multiplication case E, extracted from the ammp benchmark (SPEC 2000)

6.1.1 Results and Discussion

Regarding the results in Table 6.1, SPolly achieve up to 80 fold speedup compared to gcc and clang; Polly, with the default options, only 34 fold speedup. Although smaller input data will not yield improvements in such dimensions, the tenor remains. SPolly as presented here is at least as effective as Polly but with much wider applicability. It is able to parallelize the cases A to C in a sound way without manual interaction. Case D is special, as speculation is needed to parallelize it. Nevertheless, the execution with SPolly yields more than 8x speedup (without an STM). Polly, on the other hand, can only handle case A. With the assumption that no pointers alias it is able to optimize and parallelize cases B and C, too. As this is obviously not true in general, the source has to be reviewed manually to enable such optimizations. Apart from the parallization, the cases A to C could benefit from the polyhedral transformation, hence from improved data-locality. Comparing, for example, the results of case C and D when parallelized with Sambamba, this yields an addional 3 fold speedup.

Parallelized versions

Comparing the different parallelized versions of the cases A, B and C in terms of execution time, the OpenMP version created by SPolly performed best, but only on large tile sizes. The OpenMP versions almost always perform better than the ones parallelized by Sambamba, likely because of the transaction queue approach still used for the evaluation of this work. We expect better results with the TBB[17] implementation as described earlier. Nevertheless, Sambamba and its parallelizer are needed to handle case D in the first place. And once the commit order is implemented, the STM will be used to secure the execution for arbitrary input.

SPollys sSCoP replacement

As SPolly utilizes only Polly functionalities (without any speculation) when the "-replace-sound" option is used, it is at first surprising that the results for those cases differ compared to Pollys. To explain this we have to look at the given options in more detail. Polly is designed to be used with O3 as done here, but SPolly is not. It only utilizes the scheduling and code generation pass of Polly for optimizations. When they are invoked by hand (without O3), Polly yields the same results for case A, in fact the execution time will not differ. We may conclude that for this case the introduced alias tests do not cause measurable overhead.

Backend comparison

Except for a tile size of 256, there is almost no difference between the execution times for both new code generation backends, but for this special tile size only the splitting backend is currently able to exploit parallelism. When the trip count of the first parallelizable loop is less than used thread count, here 16, the blocked and unrolled loop nest version will not be executed at all, instead the computation is done by the "remaining iterations loop" as indicated in Figure 4.5. The reason is the single bounds check right before the 16 forks/iterations which form the parallel section. Using the splitting backend, the first 4 of the 16 created loop nests will be executed concurrently while the others are skipped. Despite the fact that only one fourth of the possible parallelism was exploited, we achieved a good result for this particular case. Even the version created by the blocking backend was twice as fast as the gcc and clang versions. The reason is the improved data-locality for the two inner loops.

Summary

Even for this small case study it is hard to talk about "the" best parallelization method. Regarding only the tile size we can see an enormous impact on the runtime for some of the cases (Polly case A or SPolly with "replace sound") but it is still unclear whether other, not yet tested values may yield better results. In the general case many options and combinations could perform surprisingly well. For long running programs it might be imaginable that SPolly internally creates a table similar to the presented one. Promising options and combinations to create fast versions could be tried as well as specialized versions which are only intended for certain inputs.

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	A	B	C	D	E	Options	Tile Size
gcc	9221	9130	9154	8917	8926	best of O1, O2, O3	
clang	9268	9289	9180	9315	8376	best of O1, O2, O3	
Polly	240	9289	9180	9315	8376	O3, polly[1]	8
Polly	264	9289	9180	9315	8376	O3, polly	16
Polly	269	9289	9180	9315	8376	O3, polly	32
Polly	348	9289	9180	9315	8376	O3, polly	64
Polly	926	9289	9180	9315	8376	O3, polly	256
SPolly	377	377	365	9180	8376	isl[2], OpenMP, rs[3]	8
SPolly	371	370	349	9180	8376	isl, OpenMP, rs	16
SPolly	378	382	354	9180	8376	isl, OpenMP, rs	32
SPolly	517	520	473	9180	8376	isl, OpenMP, rs	64
SPolly	1044	114	114	9180	8376	isl, OpenMP, rs	256
SPolly	364	369	351	1110	8376	spolly, sp[4], bb[5]	8
SPolly	371	365	342	1110	8376	spolly, sp, bb	16
SPolly	377	373	345	1110	8376	spolly, sp, bb	32
SPolly	514	510	458	1110	8376	spolly, sp, bb	64
SPolly	4636	4630	4179	1110	8376	spolly, sp, bb	256
SPolly	363	363	350	1111	8376	spolly, sp, sb[6]	8
SPolly	359	362	344	1111	8376	spolly, sp, sb	16
SPolly	371	371	355	1111	8376	spolly, sp, sb	32
SPolly	509	503	460	1111	8376	spolly, sp, sb	64
SPolly	584	541	456	1111	8376	spolly, sp, sb	256

[1] -polly -enable-polly-openmp -enable-polly-vector (isl[2] included)

[2] -polly-opt-isl, enables isl based tiling and scheduling

[3] Replace sSCoPs with parallel OpenMP versions if checks were sound. For this evaluation no alias analysis was used, instead alias tests are introduced each time.

[4] parallelized with Sambamba

[5] blocking backend

[6] splitting backend

Chapter 7

Conclusion and Future Work

7.1 Future Work

The preceding Chapters already mentioned several possible approaches for future work but it is worth to summarize and supplement the list.

The Sambamba framework and especially the method versioning is designed for multiple specialized versions of a single function. Up to this point there is still open work in this context, both on the side of Sambamba and of SPolly. For the latter one preparations have already been made to catch up when the method versioning implementation of Sambamba proceeds. The sSCoP extraction explained in Section 4.2.3 allows multiple loop nest optimizations and specializations with minimal costs in terms of space as well as for analyses and transformations. Once a SCoP is extracted, the new created function is well suited for local transformations. Parameters, e.g., for loop trip counts, could be replaced by constants if profiling information indicated a certain probability for such a case. Furthermore, SPolly could create multiple optimized versions of the same sSCoP using different options and values e.g., for the tile size. The case study indicated that changes for just this particular value may have significant impact on the performance (3.35x faster compared to the default value). Considering now the vector width or the fusion strategy, it is unclear whether changes could yield similar speedups for certain, not yet investigated, situations. Apart from specialization, future work could allow SPolly to differentiate between real and overestimated dependencies or, to be more precise, it could allow SPolly to remove the overestimated dependencies during the code generation. This approach would certainly exploit more parallelism but still restrict the level of speculation. Furthermore, all speculative approaches could benefit from improved profiling, especially dependency profiling seems plausible in this context.

7.2 Conclusion

SPolly is a speculative extension for the polyhedral optimizer Polly which improves the applicability and provides first specialization approaches. The key concepts are not fully usable yet, but still effective. Compared to Polly, SPolly reveals 1.5x more parallelizable loops on the SPEC 2000 benchmarks and it is able to exploit more parallelism, even in the absence of Sambamba and an STM. Used as a non-speculative optimizer SPolly utilizes the functionality of Polly and combines it with complete alias checks to secure the parallel execution. As this behaviour is only a spin-off, the real strength of SPolly lies in the speculation. Region scores, the heuristic used to rank detected sSCoPs, combine static information with profiling data and the results of the introduced checks. Based on these ranks, SPolly will not only create a speculatively parallelized version for promising sSCoPs but also utilize the strength of the polyhedral model to improve their data-locality.

In the context of the case study, SPolly shows that both parallelization and polyhedral optimization can be effectively combined with speculation to improve their applicability and the performance for more realistic versions of the matrix multiplication example. Based on these results it is eligible to state SPolly as an effective extension to the polyhedral optimizer Polly, but with wider applicability on general purpose code and even better results for the matrix multiplication.

List of Figures

List of Tables

Appendix A

Case Study Evaluation Template

All presented cases of the matrix multiplication example have been embedded into a small template. Figure A.1 shows the template where the matrix multiplication (matmul) function is not instantiated. Apart from the main method and the initialization function, time measurement and a correctness check are included. The printed data was used to verify the computation (via a matrix dump) and to compute the mean of all execution times.

```
#include <stdio.h>
#include <time.h>

#define N 1024
void matmul(/* adapted as needed */);

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;
        }
    }
}

void dump_array() {
    int i, j;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
          printf(" %f ", C[i][j]);
        }
      printf("\n");
    }
}

int main() {
    double sum;
    struct timespec ts1, ts2;

    int i;
    for (i = 0; i < 10; i++) {
      init_arrays();
      clock_gettime(CLOCK_THREAD_CPUTIME_ID, &ts1);
      matmul(/* adapted as needed */);
      clock_gettime(CLOCK_THREAD_CPUTIME_ID, &ts2);
      fprintf(stderr, "  -- %li sec %li nsec \n", ts1.tv_sec, ts1.tv_nsec);
      fprintf(stderr, "  == %li sec %li nsec \n", ts2.tv_sec, ts2.tv_nsec);

      dump_array();
    }
}
```

FIGURE A.1: Case study evaluation template

Appendix B

Polly – Optimization Options

Different options may yield different results. Since this simple truth holds for Polly too, the options, listend in the Table below, may yield improvements even for already well performing SCoPs.

Table B.1: Overview about the optimization options of Polly

Option	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

Appendix C

SPolly as Static Optimizer

Table C.1 lists which non-speculative functionalities of SPolly can be used without the presence of Sambamba. The new code generation options can be used exactly the same as the OpenMP and vecorizer backend options. However, the "-spolly" options are exclusively allowed with "-polly-detect". Optimizations and parallel code generation are explicitly done by the region speculation, and the opt tool may break if they are used from the outside, too.

Table C.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 (no effect if checks are not sound)
-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

Bibliography

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

[12] 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.

[13] Uday Bondhugula and J. Ramanujam. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical report, 2007.

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

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

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

[17] Intel Corporation. Intel (r) threading building blocks. *Intel Technology Journal*, 12(319872):27–38, 2008. URL http://www.intel.com.

[18] 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. URL http://www.sambamba.org.

[19] 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.