SPECULATIVE LOOP PARALLELIZATION

Johannes Doerfert

Bachelor Thesis

Compiler Design Lab
Department of Computer Science and Mathematics
Saarland University

Supervisor: Prof. Dr. Sebastian Hack
Second reader: ... TODO ...
Advisors: Clemens Hammacher and Kevin Streit

April 2012

UNIVERSITÄT
DES
SAARLANDES

Declaration of Authorship

I, Johannes Doerfert, declare that this thesis titled, 'SPECULATIVE LOOP PARALLELIZATION' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"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 and Mathematics

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. One the one hand side there is Polly, a LLVM project to increase data locality and parallelism of loop nests. On the other hand there is Sambamba, which pursues a new, adaptive way of compiling and offers features like method versioning, speculation and runtime adaption. As an extension of the former one and with the capabilities offered by the later one, SPolly can perform state-of-the-art loop optimizations on a wide range of loops, even in general purpose benchmarks as the SPEC 2000 benchmark suite. I will explain when speculation is possible, how runtime information is used and how this is integrated into Polly and Sambama. At the end an evaluation on SPEC 2000 benchmarks and the Polybench 3.2 benchmark suite is presented as well as some discussions on the results.

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Content

List of Figures

List of Tables

Listings

Abbreviations

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

For/Dedicated to/To my...

Chapter 1

Introduction

1.0.1 Motivation

SPolly, short for speculative Polly [TODO or **S**ambamba **Polly** :P], is an attempt to combine two recent research projects in the context of compilers. One the one hand side there is Polly, a LLVM project to increase data locality and parallelism of loop nests. On the other hand there is Sambamba, which pursues a new, adaptive way of compiling and offers features like method versioning, speculation and runtime adaption. As an extension of the former one and with the capabilities offered by the later one, SPolly can perform state-of-the-art loop optimizations on a wide range of loops, even in general purpose benchmarks as the SPEC 2000 benchmark suite.

The presented results will show that speculative optimizations performed by SPolly not only try to increase parallelism but also data locality – two bottlenecks of modern computation are tackled at the same time.

1.0.2 Overview

This thesis is about the work on SPolly, a speculative loop optimizer based on Polly and Sambamba. The key idea is to enable more loop optimizations due to speculation. To demarcate this from guessing, static and dynamic information is collected and used by an heuristic to choose promising candidates.

The rest of the thesis will be organised as follows. The background theory and the tools SPolly is based on will be illustrated in chapter 2. SPolly starts in chapter 3 with the realization, followed by an evaluation (chapter 4) on the SPEC 2000 and Polybench 3.2 benchmark suites. Chapter 5 discusses the results in the context of possible future work. Before chapter 7 will conclude this thesis, a case study on different versions of the matrix multiplication example is presented in chapter 6.

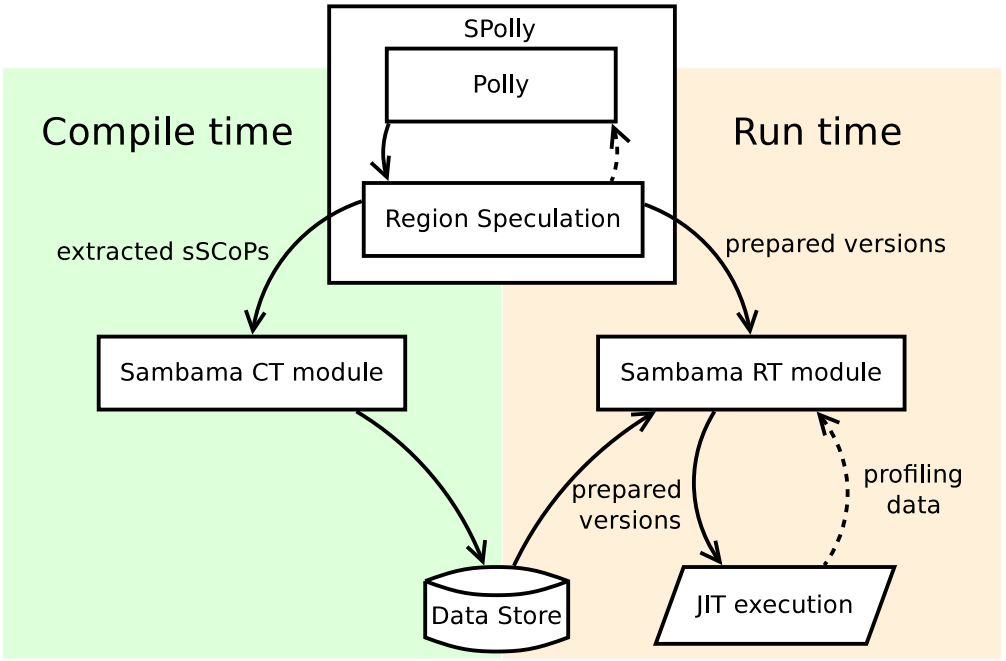


FIGURE 1.1: Overview [TODO]

Chapter 2

Background Theory

This work takes heavy use of different techniques, theories 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 I may take them for granted afterwards. As most of the key ideas behind the techniques will suffice, further details will omitted. Interested readers may fall back on the further readings instead.

2.1 LLVM - The Low Level Virtual Machine

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

[TODO where to put this ?] For simplicity most of the examples and argumentation in the thesis will deal with C like code instead of LLVM-IR.

Further Reading

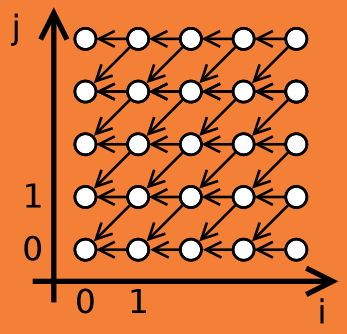
- A Compilation Framework for Lifelong Program Analysis & Transformation [1]
- `http://www.llvm.org`

2.2 The Polyhedral Model

Polyhedra are mathematical objects living in a vector space of arbitrary dimension. They describe a set of points with flat surfaces, thus they can be seen as solution set of affine inequalities. If they are bounded in all dimensions we may speak of polygons instead. The polyhedral model is a technique in compiler construction which relates loop nests with affine bounds and steppings to polyhedra. The dimension of the surrounding space is determined by the depth of the loop nests. Each loop corresponds to one dimension but there might be one more to take the scheduling of the loops into account. Each possible iteration vectors, a vector containing all induction variables of surrounding loops, is represented as a point within the polyhedra. The polyhedral model also contains a set of vectors [TODO is it ok to call them vectors ?] to model dependencies between two iterations, thus between two points. One (composed) affine transformation within this model may change all dimensions so changes in the structure of the loop nest or the scheduling can be described in a mathematical way. The intention is to analyze and transform the loop nest within the polyhedral model to find an optimal solution with respect to possible parallelism or data locality. An example loop nest and the corresponding model is given by listing 2.1a and 2.1b, respectively. Listing 2.2 shows an equivalent loop nest which is now parallelizable in the outermost loop.

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

(A) source



(B) polytope model for 2.1a

FIGURE 2.1: example loop nest

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

FIGURE 2.2: parallelizable loop nest

Further Reading

- Loop Parallelization in the Polytope Model [3]
- PoCC - The Polyhedral Compiler Collection [4]

2.3 Polly - A Polyhedral Optimizer For LLVM

Exploiting parallelism and data locality to balance the work load and to improve cache locality are the main goals of the Polly research project. With the polytope model, an abstract mathematical representation is used to get optimal results for a particular objective. It is capable of detection valid loop nests which were converted into the polytope model. Employing cloog/isl Polly is able to perform transformations within the model. The code generation part will create LLVM-IR again but if possible with thread level parallelism (via OpenMP) or vector instructions (SIMD). Despite the research status, Polly works quite well on small examples (listing 2.1a to listing 2.2) as well as on large benchmarks like the SPEC 2000 benchmark suite. All parts of Polly this work is directly based on, are now explained in more detail as they directly influenced the implementation.

2.3.1 Static Control Parts

Static Control Parts (SCoPs) describe a region which can be safely optimized using the polytope model. As this is only possible for SCoPs everything after the SCoP detection stage may consider the input as such. To find the maximal SCoPs within a function, all regions are checked for some conditions which need to be fulfilled. Starting with the underlying CFG only simple regions with well structured control flow are considered as SCoPs. This ensures that all loops and branches are perfectly nested. Other criteria restrict memory accesses or function calls. The later one is for example only allowed if the called function does not read or write any memory location at all, while the former one may not alias with other accesses within the SCoP. The trip count of an included loop needs to

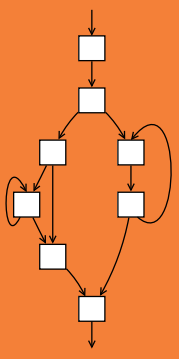


FIGURE 2.3: Possible SCoP CFG

be an affine function only depending on loop invariant "parameters" and surrounding induction variables. This restricts the loop bounds to be affine functions too. As the restriction on a SCoP are quite tight, it is hardly surprising that general purpose programs do not contain as many SCoPs as desirable. One goal of this work was to improve the situation by relaxing some of the restrictions. The first part of the evaluation will provide information on the results.

2.3.1.1 SCoP Detection

2.3.2 Loop Optimizations

2.3.2.1 Parallel Code Generation

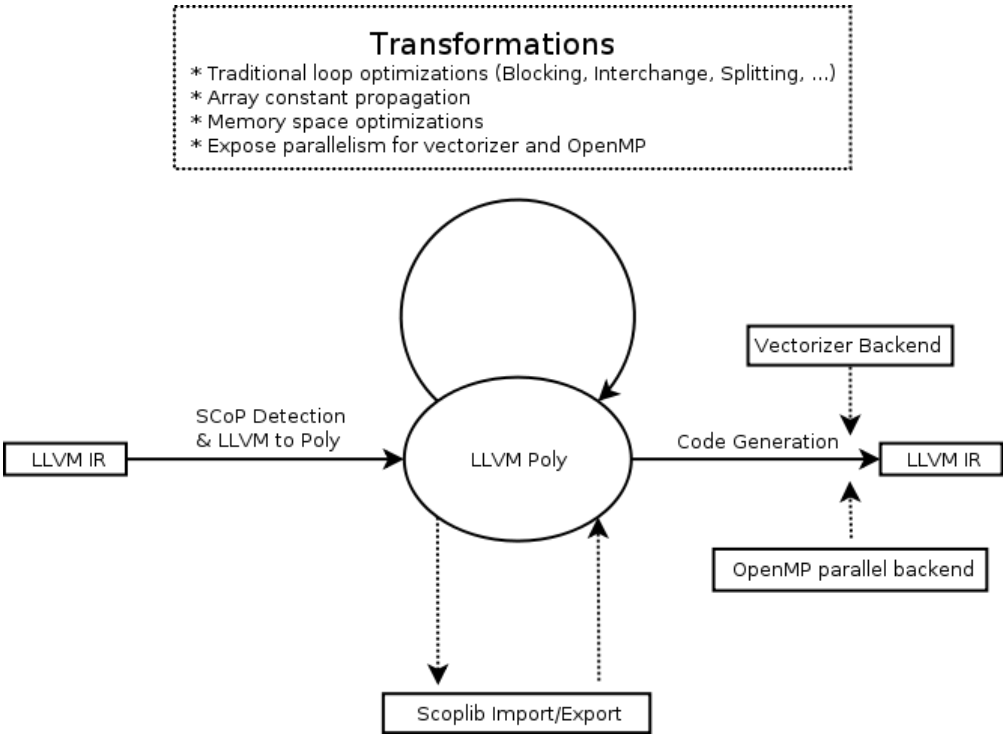


FIGURE 2.4: Pollys architecture [8]

Further Reading

- Polly - Polyhedral optimization in LLVM [5]
- Enabling Polyhedral Optimizations in LLVM [6]
- A Framework for Automatic OpenMP Code Generation [7]
- `http://polly.llvm.org`

2.4 Sambamba - A Framework For Adaptive Program Optimization

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

2.4.1 ParCFGs

2.4.2 Parallelizer

Further Reading

- Sambamba: A Runtime System for Online Adaptive Parallelization [9]
- `http://www.sambamba.org`

Chapter 3

Implementation

SPolly in its entirety is a compound of three parts. The first one is called region speculation and it is embedded into Polly. The second and the third part are Sambamba modules, one for the compile time and the other one for the runtime. The region speculation part acts like a storage and interface for all discovered sSCoPs, thus it contains most of the transformation code. The Sambama passes concentrate on the program logic which is at the moment far more evolved in the runtime part. The compile time component plays a minor role but it can be enriched by new functionalities in the future. Beyond the three parts of SPolly I implemented a basic Profiler and Statistics module for Sambamba which have been very helpful during the development and may become permanent features of Sambama as there are any comparable counterparts at the moment. During the implementation various bugs occurred and even if most of them arose through my own fault I triggered some in the code base of Polly and Sambamba too. A full table of reported bugs is listed in table **??**. Table 3.1 compares the work in a quantitative manner as it lists the lines of code (LOC) added for SPolly as well as for Sambama and Polly parts. [TODO one sentence about the table content]

Table 3.1: Lines of code for Polly, SPolly and Sambamba components

component	LOC
SPolly	XXX
Region Speculation	XXX
SPolly Sambama CT	XXX
SPolly Sambama RT	XXX
Polly	XXX
SCoP Detection	XXX
Code Generation	XXX
Sambamba	XXX
Parallelizer	XXX
TODO MORE	

[TODO rephrase] Table 3.4 lists all available command line options added in the context of SPolly. Although all of these options work without the Sambamba modules, yet Sambama at all, the last three would only produce sequential executable code without any parallelization. [TODO] As for now I am not quite sure if this could be of any practical use, but to my understanding Polly could get a similar option in the near future.

3.1 Speculative Polly

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

As the Polly architecture is nicely illustrated by figure 2.4, it has been extended in figure 3.1 to capture the changes introduced by SPolly. In comparison the region speculation, the fork join backend and the sSCoP backend have been added as they can be used without Sambamba.

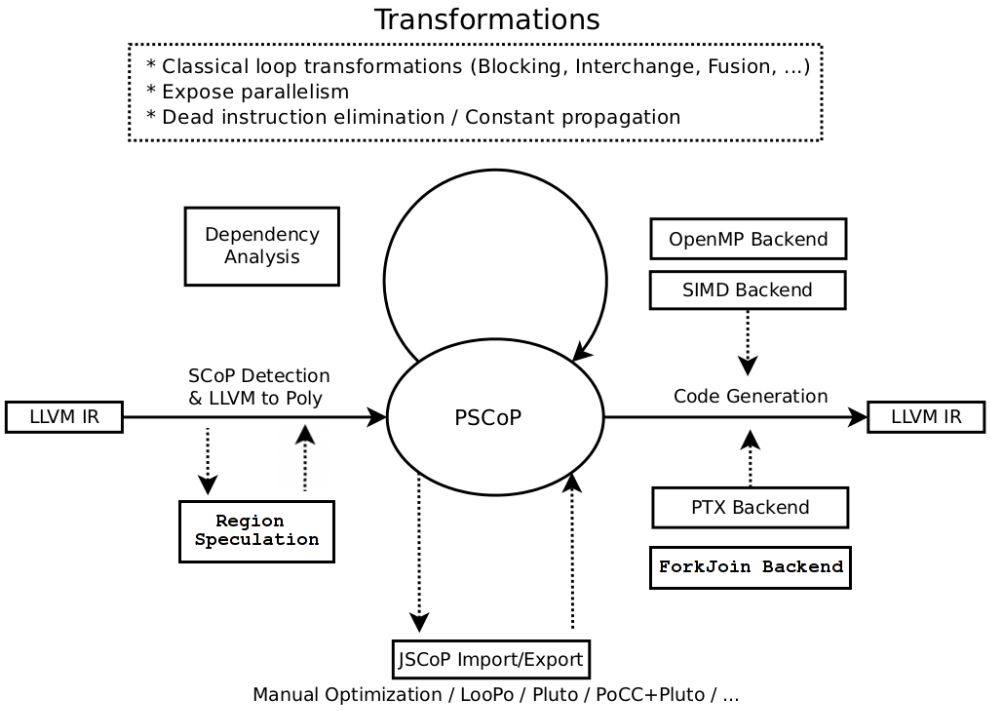


FIGURE 3.1: SPolly architecture

3.1.1 Region Speculation

The region speculation (RS) of SPolly has several tasks to fulfill. The first one includes the communication with Polly or more precise, with the SCoP detection. Each region analyzed by the SCoP detection needs to be considered as possibly speculative valid SCoPs, thus all information exposed during the detection are stored. If the region contains a validation not listed in **??** or if it is without any validation the information are discarded, else a new sSCoP is created which initially validates itself. These validation mainly computes information needed for later transformations. In the following these computations as well as the creation of profiling and parallel versions for a sSCoP are explained briefly.

3.1.1.1 Region Scores

Region scores are the heuristic used to decide which sSCoPs may be interesting to speculate on, thus for which region should profiling and parallel versions be created and used. As the former ones may change the score again it is reasonable to create parallel version later if the collected data suggest to do so. Initial efforts to create these scores did not use any kind of memory, thus every call needed to reconsider the whole region. To avoid this unnecessary computations the current score is a symbolic value which may contain variables for variables not known statically, branch probabilities and the introduced tests. Evaluation of these symbolic values will take all profiling information into account and yield a comparable integer value. Only during the initial region score creation region speculation will [TODO word! traverse?]. All instructions will be scored and the violating ones will be checked for their speculative potential. As memory instructions are guarded by the STM for the case the speculation failed, calls may not be reversible, thus without any speculative potential at all. Such function calls are not checked earlier since the region speculation needs the information about possible other branches within this region. Listing 3.2c provides such

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

(A) complete static sSCoP

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

(B) Branch within a sSCoP

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

(C) irreversible call within a sSCoP

FIGURE 3.2: example sSCoPs

an example but these cases will be revisited in the next two chapters too. Table 3.2 lists the scores for the examples in figure 3.2.

TABLE 3.2: Scores for the sSCoPs presented in various listings

listing	score
3.2a	576 (if `A,B` and `C` may alias)
3.2b	$63 * (11 + ((7 * \text{@if.then\_ex\_prob})/100) + ((5 * \text{@if.else\_ex\_prob})/100))$
3.2c	$((0 \text{ smax } \%N)/16) * (6 + (-1000 * \text{@if.then\_ex\_prob}/100)$

3.1.1.2 Memory Accesses

3.1.1.3 Aliasing

3.1.1.4 Non Invariantcy

3.1.1.5 SCoP extraction

3.1.1.6 Profiling Versions

3.1.1.7 Parallel Versions

3.1.2 Code Generation

As part of the extension of Polly a new code generation type was added. Apart from sequential, vectorized and OpenMP annotated code generation SPolly is capable of creating a unrolled and blocked loop, which can be easily translated into an ParCFG, thus parallelized by a Sambama module. Listing 3.3b presents this transformation. The special case where lower and upper bound as well as the stride are statically known constants, the second loop, which computes remaining iterations, is completely unrolled. This kind of loop unrolling and blocking may find its way into Polly in the near future.

Parallelization

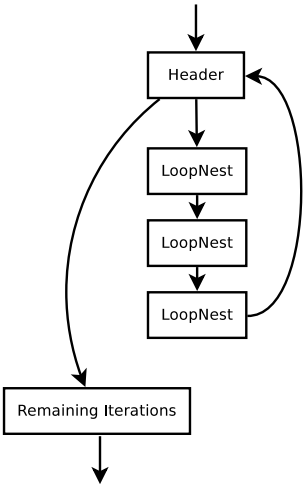
In order to secure the speculative executions with Sambambas STM (see **??**) the Sambama parallelizer needs to be used. As this parallelizer does not yet support loop parallelization per se, some transformation needed to be done first. The new code generation type was created to do the difficult one without recomputing information provided during by the polytope model (during the code generation) anyway. As these transformation yields code as in listing 3.3b the creation of a ParCFG (see **??**) remains. Figures 3.3c and 3.3d visualize these changes.

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

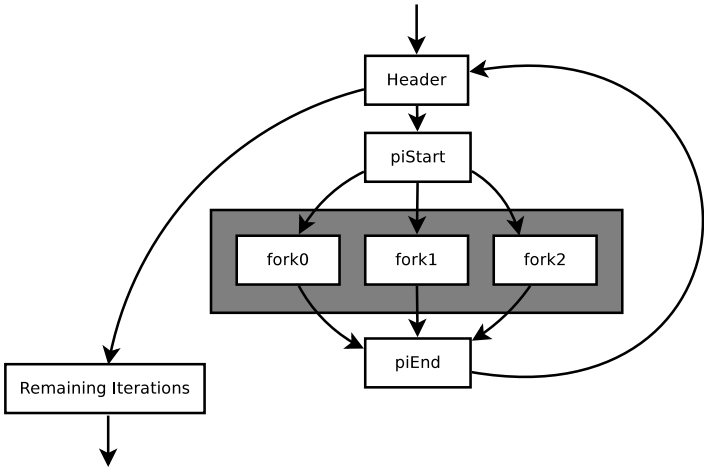
(A) source

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

(B) after N-fork creation



(C) SPolly forked CFG

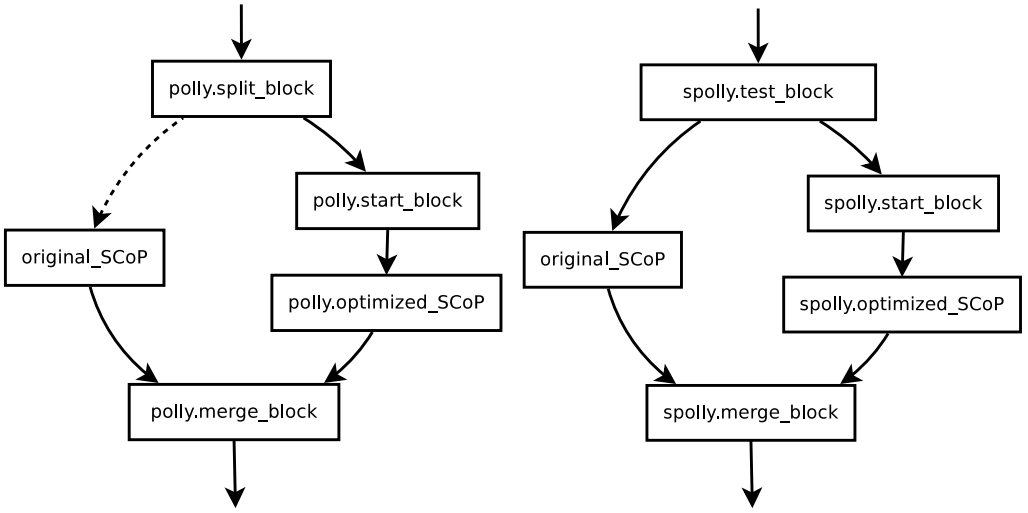


(D) SPolly ParCFG

FIGURE 3.3: Forked CFG produced by SPolly and resulting ParCFG

3.1.3 Introduced tests

To reduce the overhead of misspeculation tests are introduced in front of each sSCoP. At the moment there are two different kinds available, invariants tests and alias tests. Only the later one is used in optimized versions because the former one needs to check the invariant every iteration, thus it produces a huge overhead. In contrast to profiling versions which uses both tests to refine the score of a sSCoP. Placing the test section was quite easy, since Polly itself introduces a (constant) branch before the SCoP anyway. Figure 3.4a and 3.4b shows the (simplified) CFG introduced by Polly and SPolly, respectively. The dotted edge in the CFG produced by Polly is not taken (constant false), but remains in the CFG. As far as I know, SPolly is the first extension to Polly which uses the untouched original SCoP.



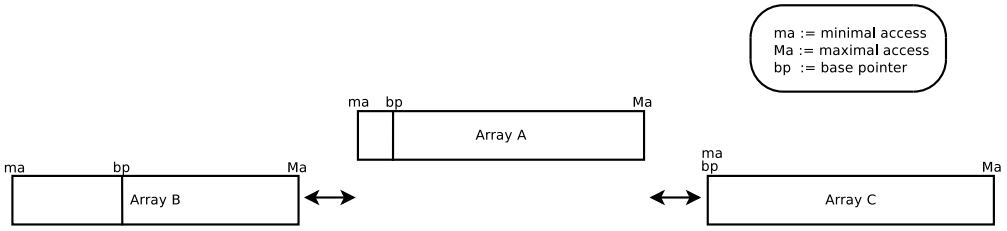
(A) Polly CFG (B) SPolly optimized CFG

FIGURE 3.4: CFG produced by Polly and SPolly, respectively

3.1.3.1 Alias tests

Testing for aliasing pointers in general would not be feasible so another way was chosen. Only sSCoP invariant pointers are tested once before the sSCoP is entered. If the test succeeds, thus no aliases are found, the optimized version is executed. At compile time all accesses for each base pointer are gathered and marked as possible minimal, possible maximal and not interesting access. At runtime all possible minimal and maximal, respectively, accesses are compared and the minimal and maximal access for each base

pointer is computed. The test itself compares again the minimal access for a base pointer with the maximal accesses for the others and vice versa. At the end of this comparison chain the result replaces the constant guard in the split block before the original SCoP and the speculative optimized one. If all base pointers are invariant in the SCoP the test is complete, thus aliasing can be ruled out for the sSCoP at runtime. Figure 3.5a illustrates the concept of the alias tests while listing 3.5b and figure 3.5c provide an example with the derived accesses.



(A) Alias test from a birds eye view

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
//    I1
    C[i*N+j] = 0;
    for (k = 0; k < N; k++) {
//       I2          I3           I4           I5
      C[i*N+j] = C[i*N+j] + A[i*N+k] * B[k*N+j];
    }
  }
}
```

(B) Aliasing accesses

Acc	bp	ma	Ma
I1	C	0	N * N
I2	C	0	N * N
I2	C	0	N * N
I3	A	0	N * N
I4	B	0	N * N

(C) Statically derived accesses

FIGURE 3.5: Alias tests concept

Complete Checks

Alias tests may rule out aliasing in sSCoPs completely, thus some sSCoPs become valid SCoPs after this tests are introduced. Such non speculative optimizations are done by the compile time part of the Sambamba module and may be included in Polly too. An example for such an sSCoP is given in figure **??**. The presented code is the well known matrix multiplication example which is a valid SCoP if the arrays do not alias. For the sake of completeness it is to mention that this code could be rewritten as array of pointers which would lead to non complete checks. Further discussion on different kinds of this example are appended as **??**.

3.1.3.2 Invariants tests

are introduced for score refinement in the profiling versions. The idea is to monitor changes in sSCoP invariant variables without analysing the call graph during runtime. Listing 3.6a gives an example of an sSCoP for which invariant tests can be introduced and 3.6b shows the modified source (high level).

```
int c;

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

(A) source

```
int c;

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

(B) source with invariant tests

FIGURE 3.6: Invariant test introduced by SPolly

3.2 Sambama Compile Time Module

The compile time part of the Sambamba module locates all sSCoPs within the given input module and transfers each one afterwards in a separated function. These extracted sSCoPs are stored within the created Sambamba bitcode or respectively executable file. Extracting every single sSCoP decreases the performance but allows to easily change and combine different optimized sSCoPs, even if they originated from the same function. This minimal functionalities helped to focus on profiling and execution during runtime, but there is a lot more potential in this part. Further work could, for example include compile time preparation of sSCoPs. Imaginable is more than just a precomputed profiling or optimized version of a sSCoP. As there are a lot of parameters which could have significant impact on the performance, several optimized versions of an sSCoP could be created and stored, in order to choose the best depending on the system and the actual run, thus depending on runtime information. While table 3.5 gives an brief overview of available options for Polly (* means, not accessible from command line) it is not clear which ones will fit best for a particular environment and sSCoP. As the method

versioning system of Sambamba evolves, the compile time part should, in order to reduce the workload at runtime and increase the ability to adapt.

3.3 Sambama Runtime Module

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

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

TABLE 3.3: Functionality of the SPolly Sambamba module

3.3.1 Profiling

3.4 Profiling For Sambamba

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

for promising candidates, even without any runtime information. This heuristic could be part of future work since even with a profiler by hand, there are cases where the gathered information (see [TODO]) are not helpful at all.

Table 3.4: Command line options to interact with SPolly

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

Table 3.5: Brief overview of Polly's optimization options

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

[1] Not available from the command line

Chapter 4

Evaluation

The evaluation, as essential part of this thesis, allows to compare the this work with others which share the same goals. Since it is hard to put effort in figures, the evaluation may provide information about the results of them. Correctness, applicability and speedup have been the stated goals which were tested using the SPEC 2000 and Polybench 3.2 benchmark suites. While the former one contains general benchmarks used to evaluate most new optimizations, the later one is especially designed for polyhedral optimizations as done by SPolly. These benchmarks might not reflect the reality in everyday optimization but they might be used to compare SPolly and Polly (as presented in the diploma thesis of Tobias Grosser [6]).

4.1 The Environment

(more details in table 4.1). running Arch linux with an *Intel(R) Core(TM) i5 CPU M 560 @ 2.67GHz* and 6GB RAM. Parallel versions could use up to four simultaneous running threads. The second one ... TODO ... As the compile time evaluation is machine independent, it was performed on the general purpose machine only. Contrary, the runtime evaluation has been performed twice, once on each machine.

Table 4.1: The evaluation environment

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

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

TODO picture of the chain

4.2 Compile Time Evaluation

The main part of the compile time evaluation aims to get quantitative results about valid and invalid sSCoPs. These results correspond with the applicability of this work, as they both outline how many regions can be taken into account now and which work is needed to increase this number. As mentioned earlier this part is mainly machine independent, since the quantitative results are. There is one case where compile time transformation can improve the program with no need of speculation at all. These cases are explained and evaluated separately in section 4.2.3.

4.2.1 Preperation

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

4.2.2 Quantitative Results

4.2.2.1 SPEC2000

TODO why not all spec2000 benchmarks ?
TODO

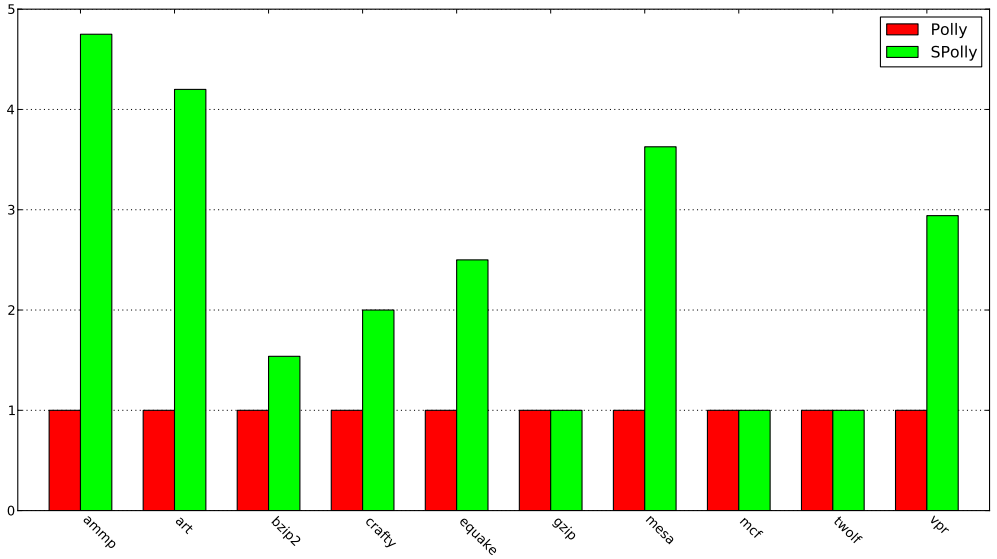


FIGURE 4.1: Numbers of valid and speculative valid SCoPs

4.2.2.2 Polybench 3.2

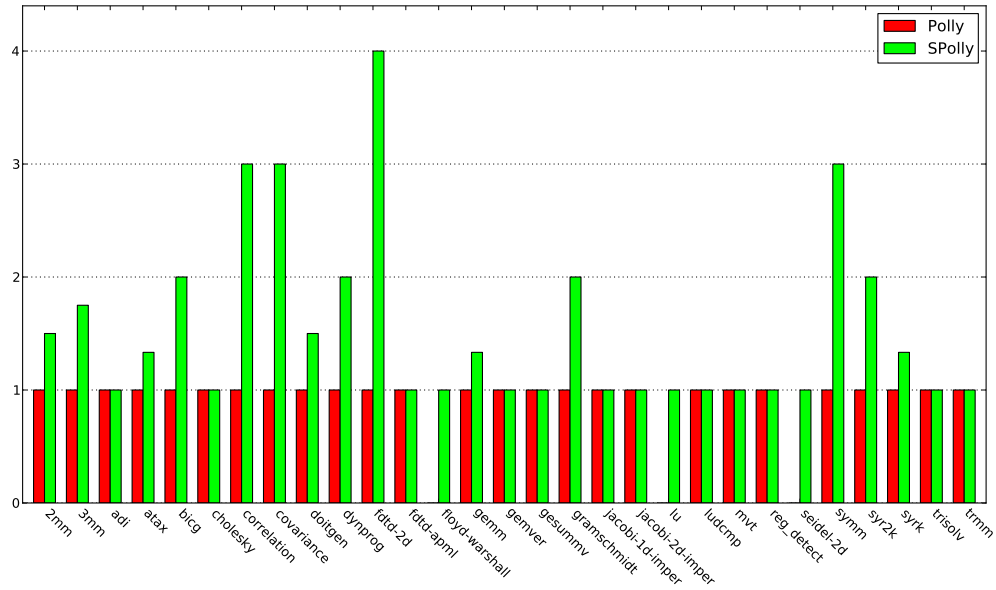


FIGURE 4.2: Numbers of valid and speculative valid SCoPs

TABLE 4.2: Results of running Polly and SPolly on SPEC 2000 benchmarks

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

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

4.2.2.3 Available tests

Alias tests

Invariant tests

4.2.3 Sound Transformations

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

4.3 Runtime Evaluation

4.4 Problems

During the work with LLVM 3.0 and a corresponding version of Polly a few problems occurred. Some of them could not be reproduced in newer versions they were just be tackled with tentative fixes, as they will be resolved as soon as Sambamba and SPolly will be ported to a newer version. Others, which could be reproduced in the current trunk versions, have been reported and listed in figure 4.3. All bugs were reported with a minimal test case and a detailed description why they occur.

TABLE 4.3: Reported bugs

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

Chapter 5

Discussion

5.1 Quantitative Results

5.2 Qualitative Results

5.3 Interpretation

Chapter 6

Case Study

6.1 Matrix Multiplication

As matrix multiplication is a well known example for loop transformation, especially polyhedral based ones, I took a closer look at several possible implementations. These results

6.1.1 Case 1

6.1.2 Case 2

6.1.3 Case 3

Chapter 7

Conclusion

Appendix A

Matrix Multiplication Example

Bibliography

[1] 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.

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

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

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

[5] 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.

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

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

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

[9] 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.

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