POLYHEDRAL SCHEDULING CACHE

Hendrik Leidinger

Bachelor's Thesis

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

Supervisor:
Prof. Dr. Sebastian Hack

Advisor:
Johannes Doerfert

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

Submitted: April 2017

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: ______________________________

"A program that produces incorrect results twice as fast is infinitely slower."

- John Ousterhout

SAARLAND UNIVERSITY

Abstract

Compiler Design Lab
Department of Computer Science

Bachelor of Science

by Hendrik Leidinger

In the more recent past, polyhedral optimizations have become very popular due to the increasing number of parallel architectures [1]. These optimizations improve data locality, expose parallelism and are performed in a mathematical framework, called the polyhedral model. Polly [2] provides an infrastructure for polyhedral optimizations in LLVM [3]. LLVM is a compiler infrastructure offering frontends for many programming languages. Polly detects optimizable code regions, translates them into the polyhedral model, performs mathematical optimizations and translates the result back into the LLVM intermediate representation.

Although Polly is a very powerful tool, there remains a major problem: With polyhedral optimizations enabled, the compile-time increases unproportional, making it unpractical for every day use. Recent measurements [4] show, that Polly increases compile-time by about 13.15% even though timeouts are used to limit analysis and optimization time for each benchmark. Some test codes, like the ASC Sequoia Crystalmk benchmark [5], produce an overhead of more than 100%.

In this work a caching mechanism for Polly is proposed in order to minimize this overhead in continuous compilations. To achieve this, computed results are stored and reused for similar inputs. The term "similar" is used, since among others, generalization techniques prior to caching are introduced. This allows the results to be reused for equivalent but different inputs.

Acknowledgments

First, I want to thank my advisor Johannes Doerfert. He provided the topic, offered help to me and answered my questions at any time. The discussions were constructive and helped me very much to come to the right decisions for this thesis. Furthermore I want to thank him for proofreading my thesis over and over again.
Special thanks go to my parents. Their confidence and serenity motivated me to get to where I am now. Moreover I'd like to give thanks to Lea Groeber, Carolyn Guthoff and Sebastian Roth for proofreading my thesis.
Last but not least I want to thank Professor Sebastian Hack and Professor Andreas Zeller for reviewing my thesis.

Contents

Declaration of Authorship **i**

Abstract **iii**

Acknowledgments **iv**

1 Introduction **1**

2 Background **3**
- 2.1 LLVM 3
- 2.2 Integer Set Library 3
- 2.3 The Polyhedral Model 4
- 2.4 Polly 5
- 2.5 Caches/Hashmaps 7
  - 2.5.1 Hash function quality 8

3 Related Work **9**

4 Concept **11**
- 4.1 The Basic Algorithm 11
  - 4.1.1 Storage 12
  - 4.1.2 The Hash Function 13
  - 4.1.3 Equality testing 16
- 4.2 Generalization and Specialization 17
  - 4.2.1 Generalized hash function 19
  - 4.2.2 Implementation 20

5 Applications **23**
- 5.1 Compile time reduction 23
- 5.2 Code Duplication Detection 23
- 5.3 Offline/Manual generation of optimized schedule 24

6 Evaluation **25**

6.1 Test Case: FFmpeg 25

6.1.1 Default Settings 26

6.1.2 Advanced Settings 31

6.2 Test Case: LLVM Test Suite 32

7 Future Work **39**

8 Conclusion **41**

List Of Figures **43**

List Of Tables **45**

Appendix A Command line options **47**

Bibliography **49**

Chapter 1

Introduction

Polyhedral optimizations can be performed in many different compilers with tools like Graphite for GCC [6], Polly for LLVM [7] or the IBM XL compiler [8]. All these approaches share very high computation costs, which restrict the opportunities for usage in practice [4, 9].

In this thesis a caching mechanism for Polly is developed. The main objective is to reduce compile-time when using Polly in a continuous development environment. In this case, the cache amortizes the costs of Polly within multiple compilations of a software project. The mechanism checks for each detected optimizable code region, a so called static control part (SCoP), whether there is an already optimized identical SCoP in the cache. If the lookup was successfull Polly proceeds with the cached result. If not, Polly computes the optimized version of the SCoP and a new pair, consisting of the input SCoP and its optimization, is added to the cache. To increase the probability of a cache hit, the input SCoP can be generalized before performing a cache lookup. In this case it needs to be specialized after optimization.

This approach unlocks its potential in a continuous development environment, as small code changes do not tend to have a strong impact on already computed SCoPs.

The remainder of this thesis is structured as follows. In Chapter 2 background information, especially about Polly and LLVM, is presented. This is followed by an examination of related work in Chapter 3. Afterwards, Chapter 4 explains the approach in more detail. We start by describing the basic algorithm, including hash-value creation, collision checking and general organization of the cache. Afterwards we expand the scheme by generalization and specialization. A perspective on how the cache can be used for different applications, including code duplication detection, is given in Chapter 5. An evaluation is done in Chapter 6 and relies on the LLVM test-suite [10] and on the FFmpeg video codec library [11]. Chapter 7 describes some ideas to further improve the performance of the cache, before we conclude in Chapter 8.

Chapter 2

Background

This chapter outlines the most important topics necessary to understand the approach. Clearly, not all topics can be presented in full detail. Thus the interested reader may refer to the given bibliography for more information on a specific topic.

2.1 LLVM

LLVM is a compiler framework, which allows to perform optimizations on a program throughout its whole lifetime [12]. At the core of LLVM is a Low Level Intermediate Representation (LLVM-IR), to which each source file is lowered and on which all optimizations are performed. LLVM-IR is a target agnostic, assembler like language. Similar to other compilers, LLVM consists of three components: the front-end, the middle-end and the back-end. The front-end parses the input and converts it to LLVM-IR. In the middle-end the code is analyzed and transformed. Basically it consists of distinct passes that can be invoked in order to perform a certain optimization. The back-end is responsible for creating machine code i.a. for x86 or ARM architectures.

2.2 Integer Set Library

The integer set library (isl) is a C-library for the administration of Presburger integer sets and maps. It is developed by Sven Verdoolaege [13]. The main data-structures are defined in the following.
A Presburger integer set is the union of basic integer sets. A basic integer set is a function $\mathbb{Z}^n \rightarrow 2^{\mathbb{Z}^d} \ : \ s \rightarrow \mathcal{S}(s)$, where

$$\mathcal{S}(s) = \{x \in \mathbb{Z}^d \ : \ \exists z \in \mathbb{Z}^e : Ax + Bs + Dz + c \geq 0\}$$

and $A \in \mathbb{Z}^{m*d}, B \in \mathbb{Z}^{m*n}, D \in \mathbb{Z}^{m*e}, c \in \mathbb{Z}^m$.
In other words $\mathcal{S}(s_0...s_n)$ is described by a conjunction of constraints of the form:

$$c + a_0x_0 + ... + a_dx_d + b_0s_0 + ... + b_ns_n + d_0z_0 + ... + d_ez_e \geq 0$$

where $s_0, ..., s_n$ are parameters, $z_0, ..., z_e$ are existentially quantified variables, $x_0, ..., x_d$ are set variables and $a_0, ..., a_d, b_0, ..., b_n, d_0, ..., d_e$ are constants. Inequalities of this kind are also called Presburger inequalities. The lefthand-side expression is also called a Presburger expression.

A Presburger integer map is the union of basic integer maps. A basic integer map is described similarly by the function $\mathbb{Z}^n \rightarrow 2^{\mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2}} \ : \ s \rightarrow \mathcal{S}(s)$, where

$$\mathcal{S}(s) = \{x_1 \rightarrow x_2 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \ : \ \exists z \in \mathbb{Z}^e : A_1x_1 + A_2x_2 + Bs + Dz + c \geq 0\}$$

and $A_1 \in \mathbb{Z}^{m*d_1}, A_2 \in \mathbb{Z}^{m*d_2}, B \in \mathbb{Z}^{m*n}, D \in \mathbb{Z}^{m*e}, c \in \mathbb{Z}^m$.
As an example consider the code in Figure 2.1. The statement S is executed only for

```
1 for (i = 0; i <= X && (i % 2) == 0 ; i++)
2   for(j = 0; j <= Y; j++)
3     S(i,j);
```

FIGURE 2.1: Example

even i and only if $0 \leq i \leq X$ and $0 \leq j \leq Y$. This circumstance is described by the following integer set, a so called iteration domain for S:

$$\{(i,j) \mid \exists a : (i \geq 0 \ \wedge \ -i + X \geq 0 \ \wedge \ j \geq 0 \ \wedge \ -j + Y \geq 0 \ \wedge \ i - 2a \geq 0 \ \wedge \ 2a - i \geq 0)\}$$

or in simplified notation:

$$\{(i,j) \ : \ 0 \leq i \leq X \ \wedge \ 0 \leq j \leq Y \ \wedge \ i \,\%\, 2 = 0\}$$

2.3 The Polyhedral Model

The polyhedral model was introduced to perform optimizations on so called static control parts (SCoPs) in a mathematical way[9]. Roughly speaking, a SCoP is a program part with static control flow, that is, loop bounds and conditionals are Presburger expressions with surrounding loop variables and unknown, but constant parameters as operands. Loop variables may only be increased by a constant and array subscripts need to be Presburger expressions.

With the help of the polyhedral model, the iteration domain of a statement is described by a Presburger integer set as depicted in Figure 2.2. A is a constant matrix, x is

$$\mathcal{D} = \{x \mid Ax + c \geq 0,\ x \in \mathbb{Z}^n,\ A \in \mathbb{Z}^{m*n},\ c \in \mathbb{Z}^m\}$$

FIGURE 2.2: Iteration domain

a vector consisting of the surrounding induction variables, and c is a constant vector, that may also contain parameters. Intuitively, the iteration domain describes the set of induction variable assignments for which the statement is executed. A statement, where all induction variables are replaced by valid assignments, is called a statement instance. In order to perform optimizations on a SCoP, it is crucial to know the dependences between different statement instances. For example, consider the following memory accesses:

```
1 int A[2];
2 A[0] = 10;
3 A[1] = A[0] + 20;
```

An optimization must not swap these two statements, since there is a read-after-write (RAW) dependency. If those two statements are swapped, the value of A[1] is undefined, since the value of A[0] is initially undefined. There exist two more important dependences, namely write-after-read (WAR) and write-after-write (WAW). The interested reader may refer to [14] for more information.
Having regard to its dependences, a static control part is optimized by changing the execution order of the statement instances for each statement. This may also imply, that the result can be executed in parallel to some extent.

2.4 Polly

Polly is a tool, that enables polyhedral optimizations in LLVM [2]. The main analysis and transformation steps are illustrated in Figure 2.3. First, it detects SCoPs in a program and transforms them into the polyhedral representation. This representation will then be optimized based on the computed dependences. Finally the resulting SCoP is translated back into LLVM-IR. Polly uses the integer set library (isl) to represent SCoPs in the polyhedral model.
Within Polly, a SCoP is represented as a tuple consisting of a list of statements and a context. The context is an integer set constraining the parameters of the SCoP. Each statement is a quadruple comprising its name, domain, schedule and a list of memory accesses. The domain is constructed according to the rules in section 2.3. The schedule is an integer map that maps each element of the domain to a specific point in time,

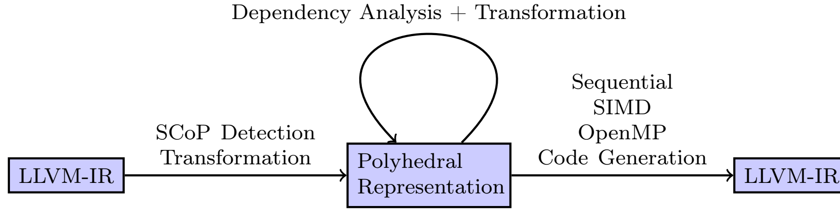


FIGURE 2.3: High level steps of Polly [2]

where the statement is executed. This "point" is described by a vector. A statement instance is executed before another statement instance, if and only if its schedule vector is lexicographically smaller. Upon creation, the schedule describes the original execution order of the statement instances as described by the input program. Each memory access consists of the access function and the type of the access. Possible access types may be read, write or may-write. The memory access function is an integer map that maps each domain element to an address in memory. Polly was designed in such a way, that there is no need for changing anything else except the schedule upon optimization of the SCoP. Thus, it is sufficient to store the new schedule instead of the entire new SCoP in the cache.

As an example, consider the SCoP in Figure 2.4. This code implements matrix addition on two 1000x1000 matrices. The polyhedral representation is depicted in Fig-

```
1  for (j = 0; j < 1000; j++)
2    for(i = 0; i < 1000; i++)
3      C[i][j] = A[i][j] + B[i][j]; // Stmt P
```

FIGURE 2.4: Two dimensional SCoP with three multidimensional memory accesses

$$\mathcal{S} = (\{\}, [P]) \tag{2.1}$$

$$P = (\mathcal{D}_P, \Theta_P, [(read, \mathcal{A}_P), (read, \mathcal{B}_P), (write, \mathcal{C}_P)]) \tag{2.2}$$

$$\mathcal{D}_P = \{[i,j] : 0 \leq i < 1000 \wedge 0 \leq j < 1000\} \tag{2.3}$$

$$\Theta_P = \{[i,j] \rightarrow [j,i]\} \tag{2.4}$$

$$\mathcal{A}_P = \{[i,j] \rightarrow A[i,j]\} \tag{2.5}$$

$$\mathcal{B}_P = \{[i,j] \rightarrow B[i,j]\} \tag{2.6}$$

$$\mathcal{C}_P = \{[i,j] \rightarrow C[i,j]\} \tag{2.7}$$

FIGURE 2.5: Polyhedral Representation of the SCoP in Figure 2.4

ure 2.5. The SCoP $\mathcal{S}$ has no parameters and the list of statements consists only of the element P, as one can see in Equation 2.1. P is composed of the domain $\mathcal{D}_P$, schedule Θ_P and a list of memory accesses as depicted in Equation 2.2. This list contains two read accesses $\mathcal{A}_P$ and $\mathcal{B}_P$ and one write access $\mathcal{C}_P$. The domain $\mathcal{D}_P$

states, that statement P is only defined for $0 \leq i < 1000$ and $0 \leq j < 1000$, as one can see in Equation 2.3. Θ_P in Equation 2.4 defines the schedule of the statement instances as given by the input program. The memory accesses are, for the given schedule, described by the identity function, as seen in Equations 2.5, 2.6 and 2.7. In terms of data locality this code is not optimal. Normally arrays are stored contiguously in memory as depicted in Figure 2.6. Thus, the example code forces the elements of each array to be accessed at non-consecutive positions. As a result, the CPU cache will not correctly predict future memory accesses, which leads to significantly slower execution times. Polly would modify the schedule like this:

0xFFA0	A[2][1]
0xFF80	A[2][0]
0xFF60	A[1][1]
0xFF40	A[1][0]
0xFF20	A[0][1]
0xFF00	A[0][0]

FIGURE 2.6: Example array of size 3x2 in memory

$$\Theta_P = \{[i, j] \rightarrow [i, j]\}$$

which means, that in the optimized loop nest the two loops are swapped. This results in the accesses being contiguous. This conversion is valid, since there are no dependences between any pair of statement instances in this example.

2.5 Caches/Hashmaps

A cache is a fast buffer, that either tries to reduce access times to slower memory or to avoid expensive recomputations of already computed results. Usually the size of a cache is severely limited. There are many different approaches to decide which data to keep in the cache and which not in order to cope with this restriction. Since our cache stores SCoPs in the file system, it is not limited in size.

Our caching algorithm stores computations in a hash map. In order to do this, a hash function needs to be defined. Furthermore an equals-function is needed to check for the equality of two elements. A hash function maps values of an input set onto fixed-size integer values. An equals-function takes two values as input and returns *true*, if these values are considered equal and *false* otherwise. In order to search for an element in the hash map, it is sufficient to search for it in the collision list of its hash value, thus to limit the search to all cached elements that have the same hash value.

An example hash map is depicted in Figure 2.7. In this example, the hash function returns the first character of the input string as a key. A new element is attached to the collision list at position *key* (modulo array size) in the array. In order to search for an element, the equality with each element in the collision list at position *key* (modulo array size) is checked.

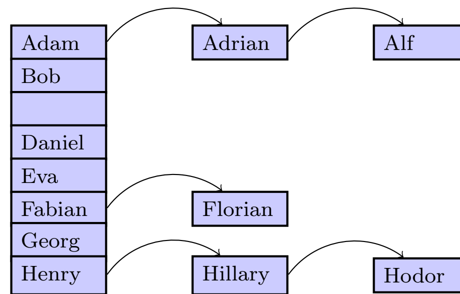


FIGURE 2.7: A simple hashmap

2.5.1 Hash function quality

According to the red dragon book [15], the quality of a hash function of a given hash map can be computed as follows:

$$\sum_{j=0}^{m-1} \frac{b_j(b_j+1)/2}{(n/2m)(n+2m-1)}$$

where m is the number of slots, b_j is the number of items in the j-th slot and n is the total number of items. A good hash function produces a value between 0.95 and 1.05. A bad hash function maps many elements to the same hash value, producing long collision lists. In this case the quality function produces a higher value.
The quality function is used in Chapters 4 and 6 to classify the distributions of our hash function.

Chapter 3

Related Work

Existing approaches are mainly concerned with equivalence checking of program parts, that are similar to SCoPs. But to the best of our knowledge there exists no caching mechanism for a polyhedral optimizer upon creation of this thesis.

Sven Verdoolaege et al. [16] present a method to check if two static affine programs are equivalent. The checks are performed on the dependence graphs of these programs. Roughly speaking a dependence graph models the data dependencies of a program's operations. The authors use a slightly different definition of a dependence graph than is usually used for the optimization of static affine programs. Applying a further normalization step on these dependence graphs, the authors were able to show that if two normalized dependence graphs are equivalent, then the corresponding input programs are equivalent. Intuitively their algorithm checks whether two dependence graphs are equivalent, which then implies that the two programs are equivalent. Their approach is not complete since it is undecidable to check for equivalence of two static affine programs [16]. They aim for supporting designers, that try to improve the performance of array and loop intensive programs by signaling whether the two programs are equivalent. Since their definition of a static affine program is similar to the definition of a SCoP, one could also imagine using this technique for our cache.
Denis Barthou et al. [17] show that checking for the equivalence of two Systems of Affine Recurrence Equations (SARE) is undecidable in general by reducing the tenth Hilbert problem to the equivalence of two SAREs. One could imagine a SARE as a set of equations where array elements on the lefthand-side are assigned to the result of a term on array elements on the righthand-side. Each array element may only be written once. For example:

$$O[i] = f(I[i], J[i], K[i]),\ 0 \leq i \leq n,$$

is a valid SARE which assigns the result of $f(I[i], J[i], K[i])$ to $O[i]$ for all i. Since this equivalence checking is undecidable, there is no program that is able to check for equivalence of arbitrary pairs of SAREs. The authors therefore propose an algorithm, which is at least able to check for the equivalence of some pairs of SAREs. Their algorithm converts the SAREs into memory state automata (MSA) and transforms the problem of equivalence checking into a reachability problem in the corresponding MSAs.
Since the set of all SAREs is a subset of the set of all SCoPs, we already know that comparing two SCoPs is undecidable in general.

Chapter 4

Concept

Figure 2.3 shows the high level steps of Polly. We need to expand this diagram in order to insert our caching algorithm. The result is depicted in Figure 4.1 and described in the following.

4.1 The Basic Algorithm

Intuitively, our algorithm represents the searching algorithm of a hash map: After the detection of a SCoP and the transformation into the polyhedral representation, the cache generates a hash value and compares the remaining SCoPs in memory with the input SCoP. If there is an equal SCoP in the cache, then Polly's dependence calculation and schedule generation can be skipped by directly importing the optimized, stored schedule. If not, then the schedule needs to be computed and the result is inserted into the cache. The following sections explain how hash function, equality testing and storing, including importing and exporting, have been realized.

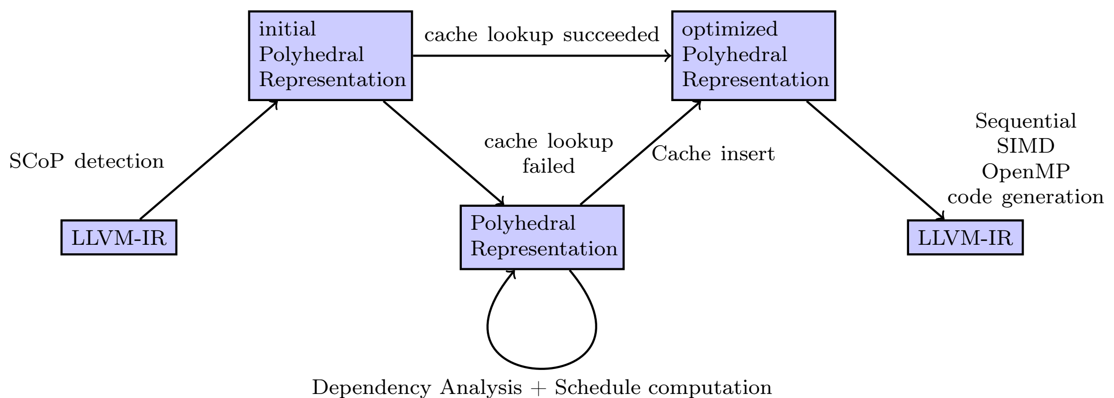


Figure 4.1: High level steps of the caching algorithm

4.1.1 Storage

We decided to use the file system to store a SCoP, its new schedule and dependences. Since the file system is part of the operating system, we do not have to use external software as it would have been the case with a database system which makes the implementation more simply. For each SCoP the cache creates several files: *Scop.dump* for the input SCoP itself, *Schedule.dump* for the new schedule and *DepLvl$\boldsymbol{X}$.dump* for each dependence with dependence level $\boldsymbol{X} \in \{0, 1, 2\}$. The latest version of Polly distinguishes three dependence levels. In the first level the coarsest analysis is applied. It distinguishes only accesses in different statements. The second level additionally differentiates between accessed arrays in the same statement. Finally, the finest analysis also distinguishes memory accesses within the same statement.

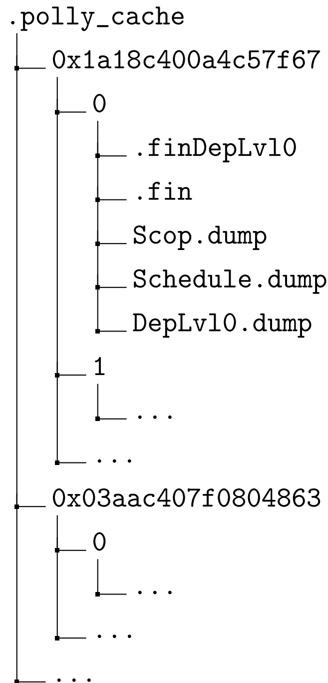


FIGURE 4.2: Example Cache content

Apart from the above-mentioned files, further hidden files are needed to prevent race conditions. In particular, the cache adds one empty file *.fin* indicating that the SCoP and the new schedule are entirely written to disk. Additionaly, one empty file *.finDepLvl$\boldsymbol{X}$* is stored for each dependence level $\boldsymbol{X} \in \{0, 1, 2\}$. This file tells the cache that the dependence of level $\boldsymbol{X}$ has been entirely written to disk. These files are needed, although LLVM does not support multithreading, since multiple instances of LLVM could be executed simultaneously. All files describing a SCoP are stored in a numbered folder. The number n of the folder describes the n-th collision list entry of the hash value of this SCoP. SCoPs with the same hash value are stored in a folder named with the hash value of the SCoPs in it. Figure 4.2 shows an example cache content. The files contain the information in the JSON format [18]. We chose this format as it is human readable and Polly already offers an import and export interface.

Table 4.1: Structure of the hash value

Number	Property	#Bits
(1)	Maximum loop depth	1
(2)	Number of Statements	4
(3)	Number of memory accesses	5
(4)	Number of read accesses	4
(5)	Number of write accesses	4
(6)	Context dimension	4
(7)	Number of induction variables	4
(8)	Sum of bounds of all induction variables	15
(9)	Sum of all constants of all memory accesses	15
(10)	Sum of all coefficients of all memory accesses	8

4.1.2 The Hash Function

The hash function encodes several properties of the SCoP in the hash value. Table 4.1 describes the structure of the hash value. Each line describes the property as well as the number of bits that are reserved in the hash value to encode this property. As an

```
1  for (i = 0; i < 1000; i = i + 1)
2    A[i] = A[i] + 1;
```

Figure 4.3: One dimensional SCoP with two memory accesses

example consider the code in Figure 4.3. This code increments the first 1000 values of an array by one. Now the cache would create a hash value as depicted in Figure 4.4.

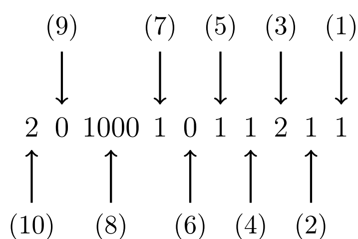


Figure 4.4: Hash value of the SCoP in Figure 4.3

The number of bits have been chosen by examining the SCoPs that Polly found in the LLVM test-suite. The results are depicted in Figure 4.5. The figures have two different background colors. If the hash of a SCoP property would produce a number which can not be expressed by the limited amount of bits reserved for the respective property, then it is assigned to the maximum value these bits can represent. These bars have a dark grey background.

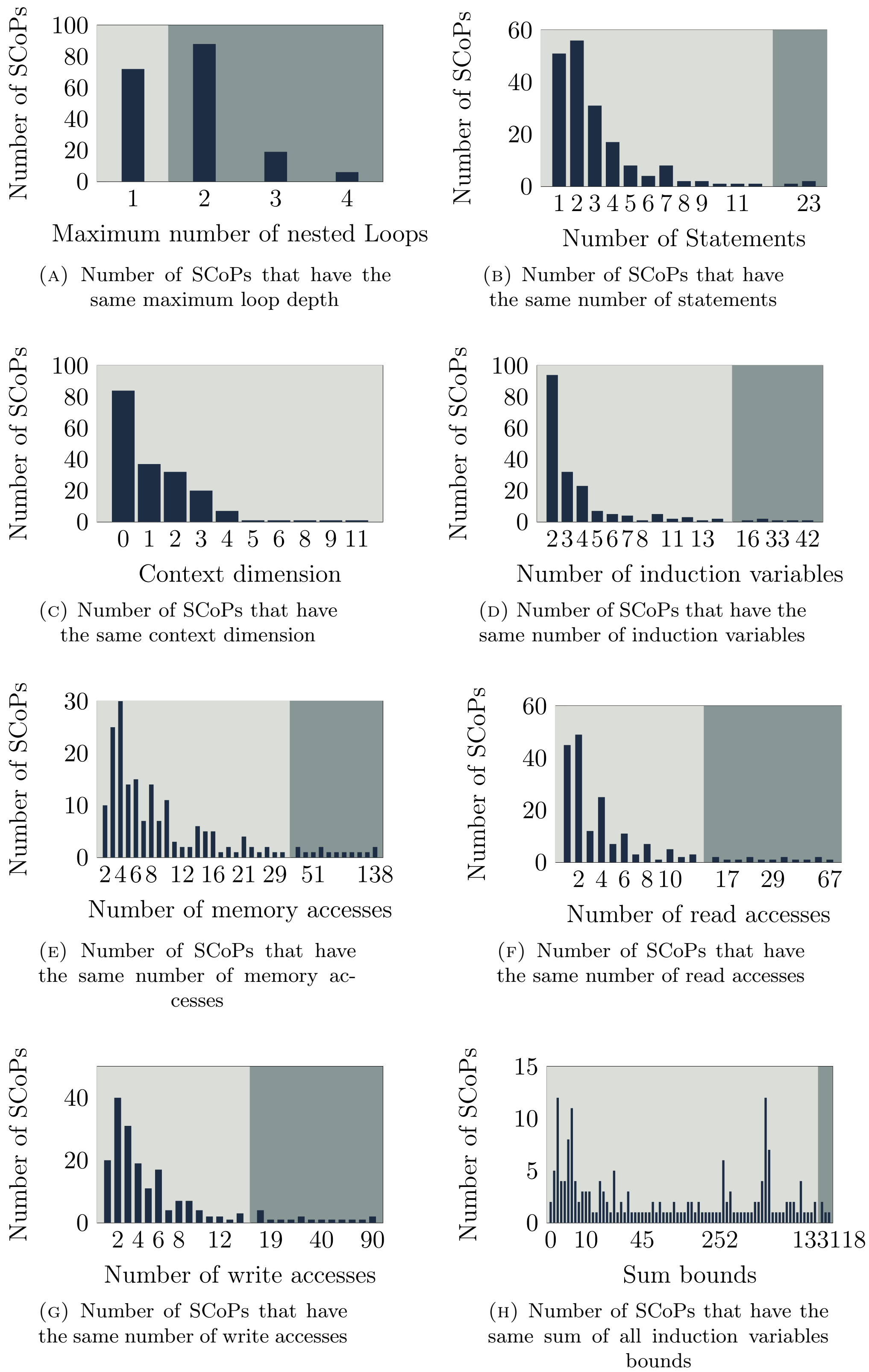
For the maximum loop depth we decided to use only one bit since there are only very few SCoPs, that have a maximum nested loop size of more than two, as one can see in Figure 4.5a. The same holds for the properties in Figures 4.5b, 4.5c, 4.5d, 4.5f and 4.5g

Table 4.2: Quality of (combinations of) SCoP properties

Property	Quality
(1)	36.51
(2)	19.81
(3)	27.26
(4)	28.96
(5)	7.57
(6)	15.32
(7)	11.07
(8)	2.85
(9)	10.53
(10)	8.21
(1)-(7)	2.59
(8)-(10)	1.1513
(1)-(10)	1.1449

with the exception, that we use four bits for each of them. The total number of memory accesses scatters a bit more than the previous properties, as one can see in Figure 4.5e. Therefore five bits are reserved for this property. The best results were achieved by property number (8), as shown in Figure 4.5h. Among all others, this property produces the best distribution. Unfortunately the sum gets very huge for some SCoPs. This is the reason why we decided to reserve fifteen bits for this property. There are a great many SCoPs where property number (9) is just zero, as depicted in Figure 4.5i. But apart from that, the distribution of the remaining SCoPs is very good. This is why another 15 bits are reserved for this property, since the range of these numbers is also very high. For the last property 8 bits were reserved, since there were only 3 SCoPs that produced a higher number.

Table 4.2 shows the hashing quality of the properties and of some combinations. One can see that no property alone achieves an acceptable distribution. The concatenation of properties (1)-(7) is also not sufficient. Suprisingly, the concatenation of properties (8)-(10) achieves a much better distribution. The concatenation of all properties hardly improves the quality. This is interesting, since it implies that properties (1)-(7) and (8)-(10) are strongly related. Altogether one can say that only the concatenation of all properties achieves an acceptable, if not necessarily perfect distribution.



(A) Number of SCoPs that have the same maximum loop depth

(B) Number of SCoPs that have the same number of statements

(C) Number of SCoPs that have the same context dimension

(D) Number of SCoPs that have the same number of induction variables

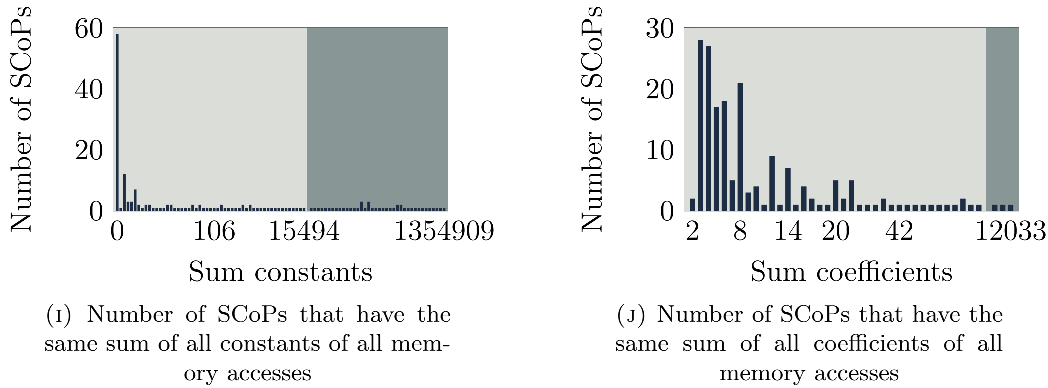
(E) Number of SCoPs that have the same number of memory accesses

(F) Number of SCoPs that have the same number of read accesses

(G) Number of SCoPs that have the same number of write accesses

(H) Number of SCoPs that have the same sum of all induction variables bounds

FIGURE 4.5: SCoP properties and their occurrences in SCoPs of the LLVM testsuite



(I) Number of SCoPs that have the same sum of all constants of all memory accesses

(J) Number of SCoPs that have the same sum of all coefficients of all memory accesses

FIGURE 4.5: SCoP properties and their occurrences in SCoPs of the LLVM testsuite (continued)

4.1.3 Equality testing

After the determination of the hash value, the equals function needs to compare the input SCoP to the SCoPs in the collision list. In general this function checks for identity with one exception: Before the comparison, all context variable names of the cached SCoP are replaced by the context variable names of the input SCoP.

Figure 4.6 describes the algorithm in pseudo code. First, it compares the context in line 2. Then it tries to find an identical statement for each statement of S_1 in lines 5-40. If there is a statement in S_1, where there exists no equal statement in S_2, then the two SCoPs are not equal and the algorithm returns *false*. Two statements are considered equal, if and only if their domain and schedule are the same and for each memory access of the first statement, there exists an identical memory access in the second statement. Thus the algorithm compares the domains in line 9, the schedule in line 13 and the memory accesses in lines 16- 31. Similar to the statements, the algorithm tries to find an identical memory access for each memory access in S_1. If there is a memory access in $Stmt_{S_1}$, where there exists no identical memory access in $Stmt_{S_2}$, then the statements are not equal and the algorithm continues with the next statement in S_2. Note, that the algorithm does not have to check, if the memory accesses and the number of statements are equal for S_1 and S_2, since this is already done by the hash function.

In principle, this code implements the equality function. However, in practice, one needs to perform some optimizations in order to get competitive code. For example it is sufficient to compare only those statements and memory accesses, that are not matched already. The worst-case runtime of this algorithm is in $\mathcal{O}(n^2+m^2)$, where n is the number of statements of S_1 or S_2 and m is the number of memory accesses of S_1 or S_2. Note, that due to the definition of the hash function, the number of statements and memory accesses are equal for S_1 and S_2. Thus we have $\mathcal{O}(n^2+m^2)$ invocations of is_equal(). This is very expensive, since the runtime of is_equal() is exponential. To mitigate this,

```
1  bool compare (Scop S1, Scop S2) {
2    if (!is_equal(S1.context, S2.context))
3      return false;
4
5    for (Statement StmtS1 : S1.Stmts) {
6      bool equalStmt = false;
7      for (Statement StmtS2 : S2.Stmts) {
8        //Compare Domain
9        if (!is_equal(StmtS1.domain, StmtS2.domain))
10         continue;
11
12       //Compare Schedule
13       if (!is_equal(StmtS1.schedule, StmtS2.schedule))
14         continue;
15
16       bool equalMemAccs = true;
17       //Compare MemAccesses
18       for(MemoryAccess MemaccS1 : StmtS1.MemAccesses) {
19         bool equal = false
20         for(MemoryAccess MemaccS2 : StmtS2.MemAccesses) {
21           if (!is_equal(MemaccS1, MemaccS2))
22             continue;
23
24           equal = true;
25           break;
26         }
27         if (!equal) {
28           equalMemAccs = false;
29           break;
30         }
31       }
32       if (!equalMemAccs) {
33         continue;
34       }
35       equalStmt = true;
36       break;
37     }
38     if (!equalStmt)
39       return false;
40   }
41   return true;
42 }
```

FIGURE 4.6: Pseudo Code of the Equality Testing of two SCoPs

Polly could define an ordering for statements and memory accesses. In this case we do not have to search for statements and memory accesses in S_2. Thus the runtime would be $\mathcal{O}(n+m)$, which is a clear improvement.

4.2 Generalization and Specialization

Apart from the basic functionality described in Chapter 4.1, the caching algorithm supports generalization and specialization, too. The idea is to increase the number of cache hits by creating a more general description out of the input SCoP and comparing it to the generalized SCoPs in the cache. The process of creating a more general SCoP is called generalization. The reverse operation is called specialization. Generalization implies, that the schedule optimizer has to find an optimized schedule for the generalized

SCoP instead of the original input SCoP. Thus generalization needs to be done prior and specialization right after obtaining the optimized schedule from Polly or from the cache. The new version of the algorithm is depicted in Figure 4.7.

Polly already applies some normalization steps, which means that even without general-

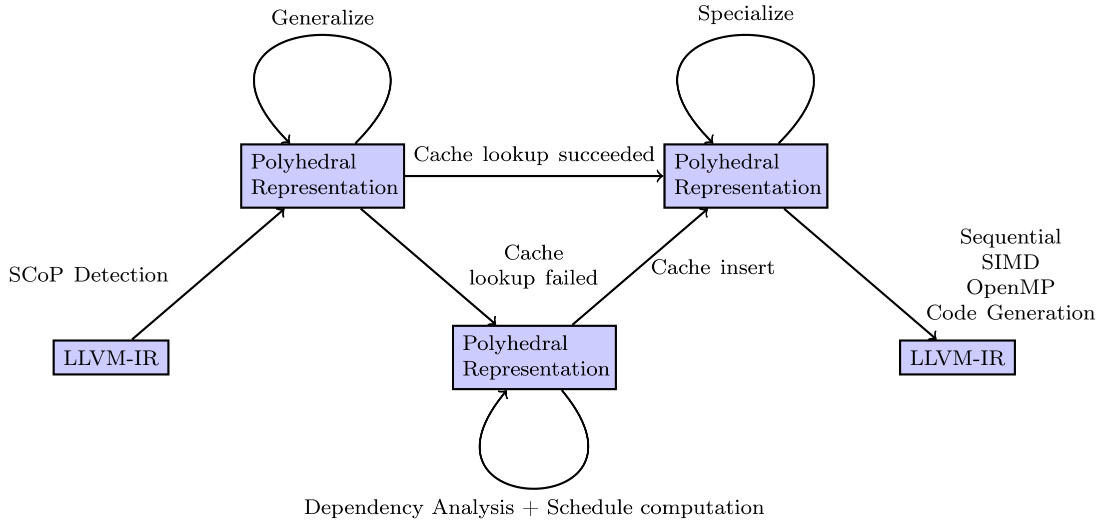


FIGURE 4.7: High level steps of the caching algorithm with generalization and specialization

ization there is a mapping from different code snippets to one polyhedral representation. For example the SCoP in Figure 4.8 is also represented by the polyhedral representation

```
1 for(i = 1; i <= 10; i++)
2   for (j = 1; j <= 10; j++)
3     C[i-1][j-1] = A[i-1][j-1] + B[i-1][j-1]; // Stmt P
```

FIGURE 4.8: Variation of the SCoP in Figure 2.4

in Figure 2.5, because Polly first converts this SCoP into the SCoP in Figure 2.4. Even if we change the right-hand expression of the statement from:

```
1 A[i-1][j-1] + B[i-1][j-1];
```

to something like

```
1 A[i-1][j-1] * B[i-1][j-1];
```

we again get the same polyhedral representation, as the actual computation is not represented in the polyhedron model, but only in the input program.

With the help of generalization we can further extend the set of SCoPs, that can be mapped to the same polyhedral representation. In principle, we are allowed to replace the upper and lower loop bounds by new parameters, since the result is still a Presburger expression. For the same reason we can replace any constant offset of an array

subscript by a new parameter. It is, however, not allowed to replace the coefficients of an array subscript by new parameters, since the result is not a Presburger expression. With that in mind the cache would generalize the SCoP in Figure 2.4 such that it looks like Figure 4.9. One can see, that even in this small example we already add ten new

```
1 for(i = lowerI; i < upperI; i++)
2   for (j = lowerJ; j < upperJ; j++)
3     C[i + N][j + M] = A[i + O][j + P] + B[i + Q][j + R]; // Stmt P
```

FIGURE 4.9: Generalization of the SCoP in Figure 2.4

parameters. In fact, it turns out, that this leads to serious runtime increases, as we will see in Chapter 6. In order to mitigate this problem, the cache also supports the generalization of the induction variable bounds only. For example the SCoP in Figure 4.10 represents the generalization up to induction variables of the SCoP in Figure 2.4. In this

```
1 for(i = lowerI; i < upperI; i++)
2   for (j = lowerJ; j < upperJ; j++)
3     C[i][j] = A[i][j] + B[i][j]; // Stmt P
```

FIGURE 4.10: Generalization of the SCoP in Figure 2.4

case only four new parameters have been introduced. But it turns out, that this is only a small improvement as compared to full generalization.

4.2.1 Generalized hash function

The hash function works just the same as in the case without generalization, with the exception that properties (8)-(10) are omitted. This leads to a poorer hashing quality, as we will see in Chapter 6. Consider for example the SCoPs in Figure 4.11.

```
1 for (int i = 0; i < 100; i++) {
2   A[i] = A[i] + 5;
3   B[i+5] = B[i] + 10;
4 }
5 for (int i = 0; i < 100; i++) {
6   C[i] = C[i] + 10;
7   D[i] = D[i] + 20;
8 }
```

(A) SCoP A

```
1 for (int i = 0; i < 100; i++) {
2   A[i] = A[i] + 5;
3 }
4 for (int i = 0; i < 100; i++) {
5   B[i] = B[i] + 10;
6   C[i] = C[i] + 10;
7   D[i] = D[i] + 10;
8 }
```

(B) SCoP B

FIGURE 4.11: Two SCoPs that are not equal, but the generalized hash function creates the same hash value for both

These SCoPs are not equal by definition of the equality function. Neither in the general case nor in the basic case. But in the general case, the SCoPs produce the same hash

value, since properties (1)-(7) are the same in both SCoPs. Thus the two SCoPs collide. However, in the basic case, the sum of all constants of all memory accesses is not the same, since the array subscript of the write access to array B is i+5 in the first SCoP.

4.2.2 Implementation

The cache generalizes domain, schedule and all memory accesses for each statement. Since these are stored as Presburger integer sets and maps, it needs to replace all of them by their generalized versions. Domain and memory accesses do not change during optimization. Thus specialization can just replace them by the original ones. The optimized schedule, however, needs to be replaced by a new schedule where all new variables have been replaced by its old values. As an example for generalization consider the following integer set:

$$\{p\} \rightarrow \{[i_0] : 0 \leq i_0 \leq 1000 \ \wedge \ (p > 0 \vee p < 0)\}$$

It consists of the following basic sets:

$$\{p\} \rightarrow \{[i_0] : 0 \leq i_0 \leq 1000 \ \wedge \ p > 0\}$$

$$\{p\} \rightarrow \{[i_0] : 0 \leq i_0 \leq 1000 \ \wedge \ p < 0\}$$

The first basic set consists of the following constraints:

$$i_0 \geq 0$$

$$-i_0 + 1000 \geq 0$$

$$p \geq 1$$

The second basic set has the following constraints:

$$i_0 \geq 0$$

$$-i_0 + 1000 \geq 0$$

$$-p \geq 1$$

Currently the cache uses these constraints to create a generalized set from scratch. For this the following new constraints are created:

$$i_0 \geq V_0$$

$$-i_0 + V_1 \geq 0$$

$$p \geq V_2$$

$$i_0 \geq V_3$$

$$-i_0 + V_4 \geq 0$$

$$-p \geq V_5$$

These constraints form new basic sets with updated parameters:

$$\{p, V_0, V_1, V_2\} \rightarrow \{[i_0] : V_0 \leq i_0 \leq V_1 \ \wedge \ p > V_2\}$$

$$\{p, V_3, V_4, V_5\} \rightarrow \{[i_0] : V_3 \leq i_0 \leq V_4 \ \wedge \ p < V_5\}$$

These generalized basic sets finally yield the following Presburger integer set:

$$\{p, V_0, V_1, V_2, V_3, V_4, V_5\} \rightarrow \{[i_0] : (V_0 \leq i_0 \leq V_1 \ \wedge \ p > V_2) \vee (V_3 \leq i_0 \leq V_4 \ \wedge \ p < V_5)\}$$

Chapter 5

Applications

This chapter briefly describes some applications of the cache.

5.1 Compile time reduction

The most obvious application is the reduction of the compile time when using Polly during the compilation with LLVM. The cache is able to reduce schedule and dependency computation time. Since both computations are exponential in time [14, 19], it is indeed faster to search for an optimized schedule in the cache instead of computing it all over again as we will see in Chapter 6.

5.2 Code Duplication Detection

Alongside its primary task, it turned out that the cache can be used as a powerful code duplication detector with only small modifications. For this the cache creates a random number upon initialization. Then for each SCoP that is inserted or found, a file named after the random number is stored into the folder of the SCoP. For a new input SCoP a code duplication warning is emitted if:

- the cache is able to find an equal SCoP in memory and
- the random file's name of this cached SCoP is the same as the current random number of the cache

Since we create a new random number for each compilation unit, a code duplication warning is only emitted if we inserted a SCoP and found this SCoP in the cache during the same compilation unit or if we found the same SCoP twice. Thus there exist two

```
1  void computeA(){
2    for (int j = 0; j < 10000; j++){
3      for (int i = 0; i < 10000; i++){
4        A[i + 10][j] = A[i][j] + 1;
5      }
6    }
7  }
8
9  void computeB(){
10   for (int j = 0; j < 10; j++){
11     for (int i = 0; i < 10; i++){
12       A[i][j] = A[i][j] + 1000;
13     }
14   }
15 }
```

FIGURE 5.1: Code duplication

```
1  void generalAB(int UB, int offset, int cst){
2    for (int j = 0; j < UB; j++){
3      for (int i = 0; i < UB; i++){
4        A[i + offset][j] = A[i][j] + cst;
5      }
6    }
7  }
8  void computeA(){generalAB(10000, 10, 1);}
9  void computeB(){generalAB(10, 0, 1000);}
```

FIGURE 5.2: Possibly fix for the code duplication in Figure 5.1

SCoPs, at different positions in the project, that are very similar and therefore could be combined to one function.
This technique works with and without generalization. With generalization we can get even better results. For example consider the code snippets in Figure 5.1. With the help of full generalization the cache can detect that computeB() is the same as computeA(). One possible fix for this duplication is depicted in Figure 5.2.

5.3 Offline/Manual generation of optimized schedule

The cache might be used to compute the new schedule of some cached SCoPs when the host CPU has not much work to do. For example, this might be done for SCoPs which are used very often and where the optimization time takes very long. The cache already allows to do this manually. Since the cached files are accessible and human readable, a user can change the optimized schedule by himself. Then, it is used during the next compilation.

Chapter 6

Evaluation

For the evaluation, Polly's performance with and without cache is tested in two different settings. In the first one, a continuous development environment is simulated. The second one examines the performance of the cache during the compilation of multiple software projects. Table 6.1 shows the testing environment that was used in both cases.

Table 6.1: The Testing Environment

CPU	AMD Athlon II x2 240e
clock speed	2.8GHz
L2 Cache	1MB
#cores	2
RAM	8GB
OS	Manjaro Linux 16.10

6.1 Test Case: FFmpeg

FFmpeg is a converter for audio and video sources [11]. It is very loop intensive and therefore suitable as a case study to simulate a continuous development environment. The testing environment measures compile time of FFmpeg with and without cache. This is done using two kinds of settings, the default and the desired settings. The latter are settings that the developers of Polly would like to have as default, but due to the current compile-time increase, are not. We cloned the lastest version of FFmpeg on github and checked out to the commit, 1000 commits before the latest. Then ffmpeg was compiled with LLVM/Polly with and without Cache. After that, we forwarded 100 commits and did the same again up to the latest commit. Each of the two test cases was repeated five times.

6.1.1 Default Settings

In the first part we measured the compile time when using Polly with default settings. We distinguish three cache settings: full generalization, generalization of induction variables and no generalization. All cases are compared to the compilation time of Polly without cache.

6.1.1.1 Without Generalization

The left chart in Figure 6.1 compares the compile time of FFmpeg when using Polly with cache and no generalization and without cache.

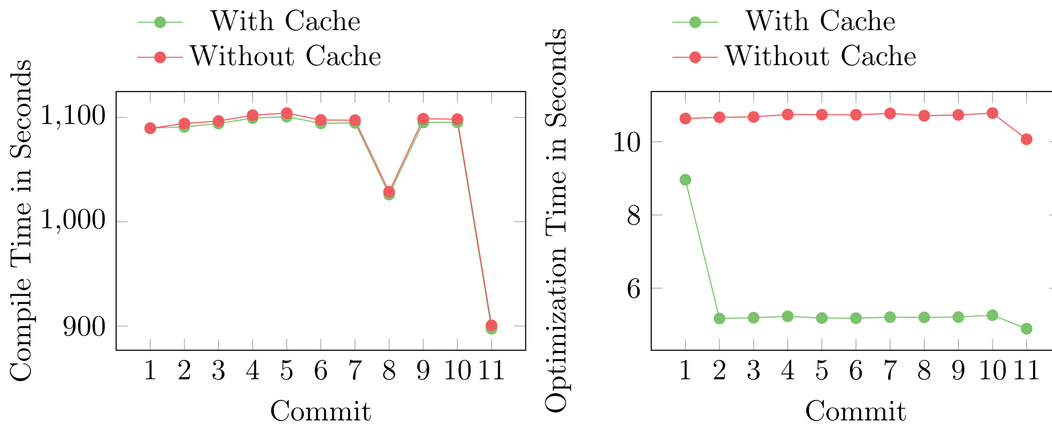


FIGURE 6.1: Compile (left) and optimization time (right) with cache and no generalization and without cache

One can see a tiny, but not mentionable improvement with cache. But when only the optimization time of the SCoPs on the right chart is considered, which is the only part that the cache is able to improve, one can see a strong improvement by about 50% compared to the optimization time without cache. Figure 6.3 shows the distribution of the runtime among the individual steps of the cache. One can see that apart from the schedule optimizer, the schedule importer and the collision checker require by far most of the time. Probably one could further improve the importation time by using a customized format instead of JSON. Furthermore

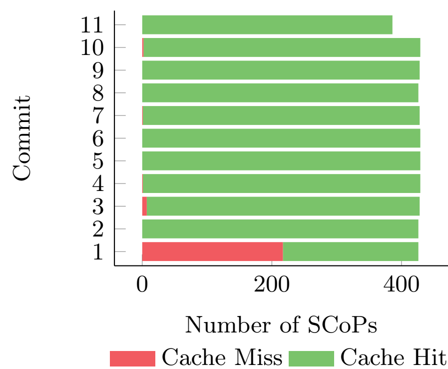


FIGURE 6.2: Cache hits and misses for each commit with default settings and no generalization

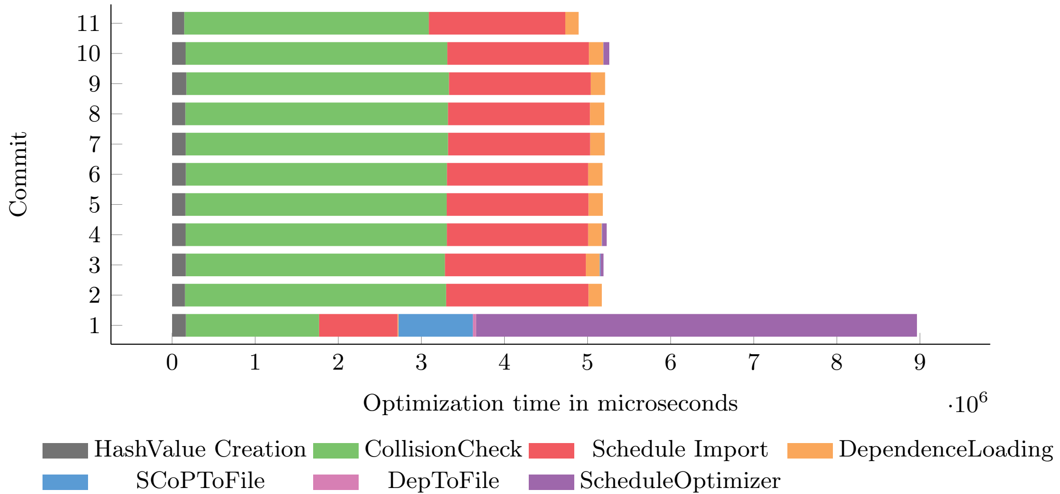


FIGURE 6.3: Distribution of the runtime among the different steps of the cache

one could probably improve the collision checker by comparing strings instead of integer sets in the integer set library. Figure 6.2 thoroughly confirms our assumption that small code changes will not have a strong impact on already computed SCoPs. While on the first commit the cache can already find nearly half of the SCoPs in memory, on all following commits the number of cache misses is not noteworthy. Last but not least consider Figure 6.4 which shows the distribution of the SCoPs by the hash function. One can see that the distribution is good, since most of the buckets contain only one SCoP and the maximum bucket size is three. The formula in Section 2.5.1 returns the value $\sim$1.05479 for this distribution. Thus the hash function produces a good distribution for the SCoPs. Altogether one can say that the cache cuts a fine figure with default settings, reducing the optimization time of SCoPs by about 50%.

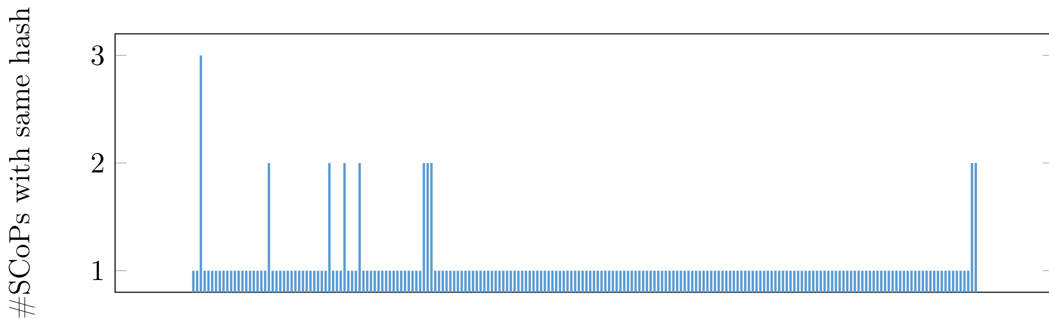


FIGURE 6.4: Hash function distribution for non-generalized SCoPs

6.1.1.2 Taking Generalization into account

The situation becomes very different with regard to generalization. In this case the overall compile time increases dramatically, as depicted in Figure 6.5, left chart. One can see that the compile time nearly doubles with full generalization and even with generalization of induction variables. The right chart in Figure 6.5 confirms the assumption that generalization is the reason for the increase. As we only consider the optimization time, it becomes clear, that the cache is responsible for the time increasement. Figure 6.6 shows

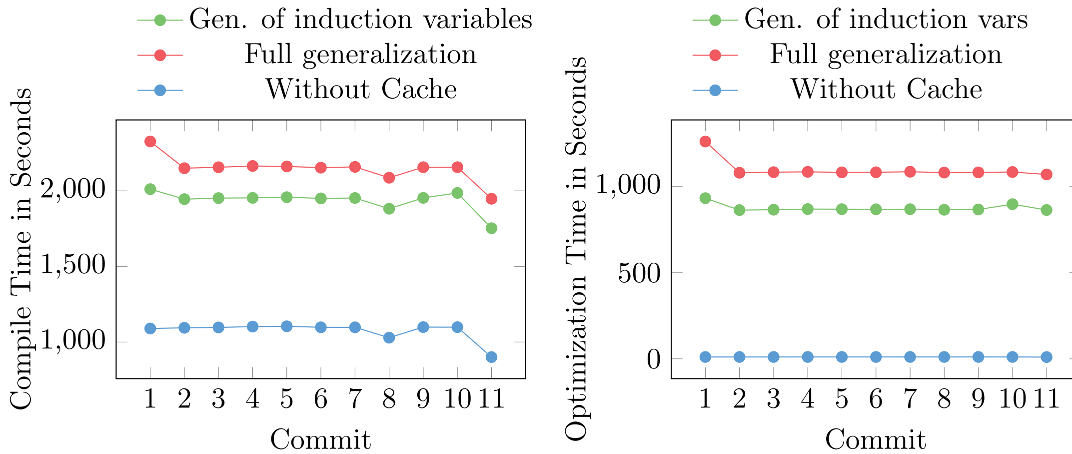


FIGURE 6.5: Compile (left) and optimization time (right) with generalization, generalization of induction variables and without generalization

the runtime distribution among the different steps of the cache with full generalization on the right and generalization of induction variables on the left chart. Generalization, specialization and collision checking consume by far most of the time in both cases. One reason for that could be the fact, that the context increases much more than expected.

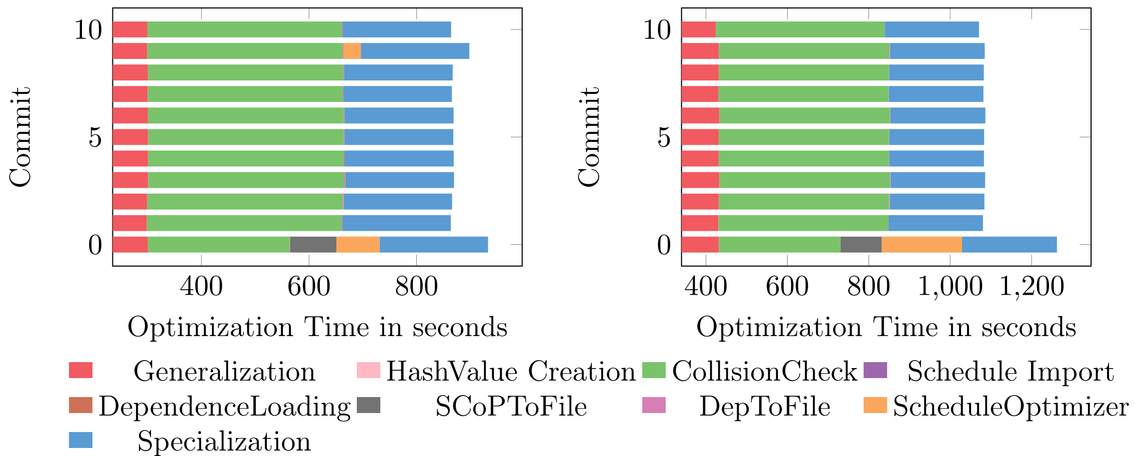


FIGURE 6.6: Distribution of the optimization time among the different steps of the cache for generalization of induction variables (left) and full generalization (right)

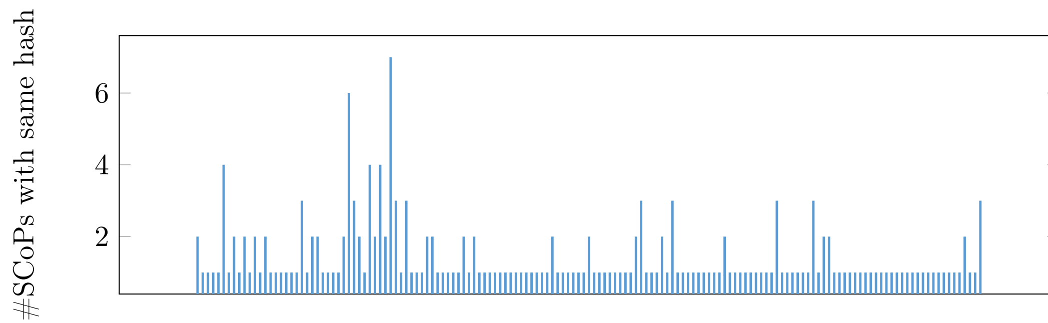
FIGURE 6.7: Cache hits and misses for each commit with generalization of induction variables (left) and full generalization (right)

Thus the number of checks that have to be performed in the equality function increase as well. Note, that some colors are not visible in the chart, simply because these steps consume much less time. Figure 6.7 shows the number of SCoPs found by the cache for each commit. Unfortunately, generalization hardly improves this number, making it somehow redundant in its current state, since this was the reason why generalization was introduced in the first place. The quality of the hash function decreases significantly as one can see in Figures 6.8a and 6.8b, as in the case of generalization the function only uses properties (1)-(7). The quality of the distribution of fully generalized SCoPs is ∼1.34 and the quality of the distribution of SCoPs that have been generalized up to induction variables is ∼1.49. Thus the hash function produces too much collisions in the case of generalization. Apart from that, there still remains one major problem: The relative number of stored SCoPs, that have been optimized by Polly is considerably smaller than before. While without generalization the percentage of stored SCoPs that have been optimized is about 73%, the percentage of optimized stored SCoPs that have been generalized up to induction variables, reduces to about 62%. The percentage of optimized, fully generalized, stored SCoPs only is 24%. By absolute numbers the following SCoPs have not been optimized:

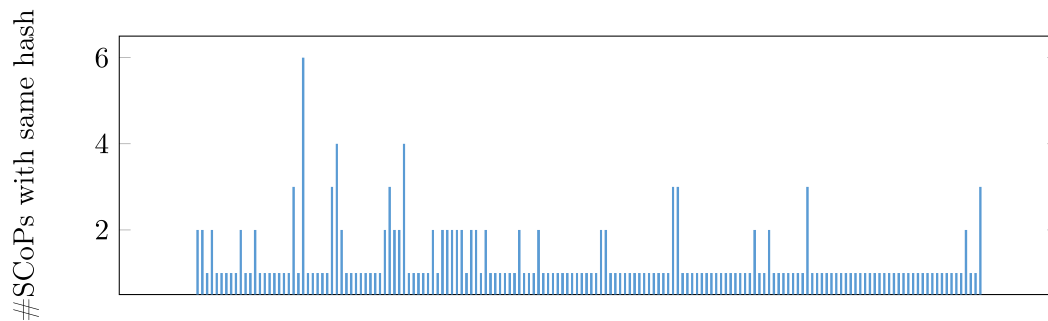
- No generalization: 60 out of 219 SCoPs
- Generalization of induction variables: 81 out of 214 SCoPs
- Full generalization: 160 out of 212 ScoPs

This is due to Polly's default settings, that abort the computation of dependences after some time limit. Furthermore, generalization prevents some optimizations that are possible on the original input SCoP.

According to the results one can say that generalization, as it is now, is unusable, for



(A) Hash function distribution for fully generalized SCoPs



(B) Hash function distribution for SCoPs that have been generalized up to Induction variables

Figure 6.8: Hash function distribution for fully generalized SCoPs and SCoPs that have been generalized up to induction variables

that reason alone, because the number of cache hits hardly increases, but also because it is very slow. However, there might be some optimization opportunities, as we will see in Chapter 7.

6.1.2 Advanced Settings

For the advanced settings we omitted generalization, since the computation did not come to an end in reasonable time. We additionally passed the following commands to Polly:

```
--polly-run-inliner=true
--polly-unprofitable-scalar-accs=false
--polly-dependences-computeout=0
```

The first argument causes an early inliner to run before Polly which increases the total number of SCoPs. The second argument marks statements with scalar accesses as optimizable. Finally, the last argument prevents Polly from stopping the computation of dependences at a specific point in time.

With these options enabled we get an even better result than with the default set-

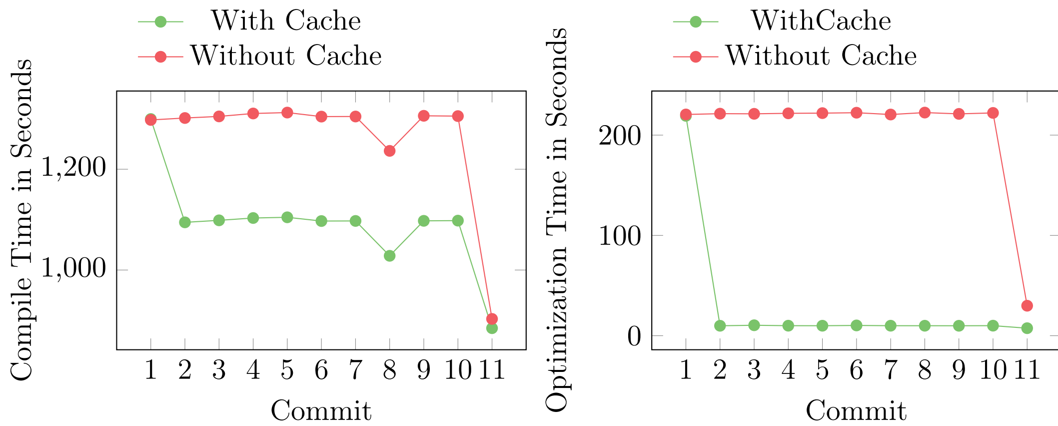


FIGURE 6.9: Comparison of the compile (left) and optimization (right) time of FFmpegs SCoPs with and without Cache and advanced settings

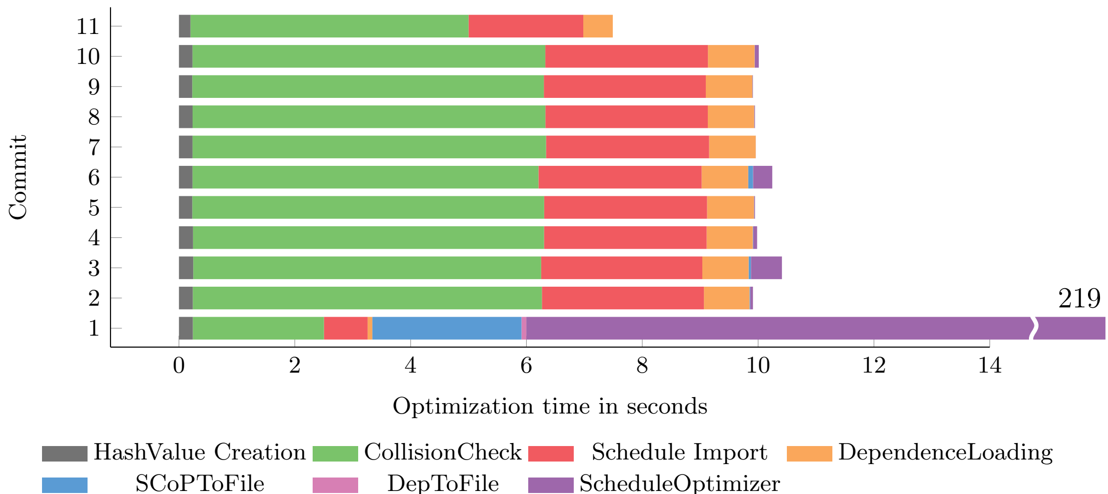


FIGURE 6.10: Distribution of the runtime among the different steps of the cache with advanced settings

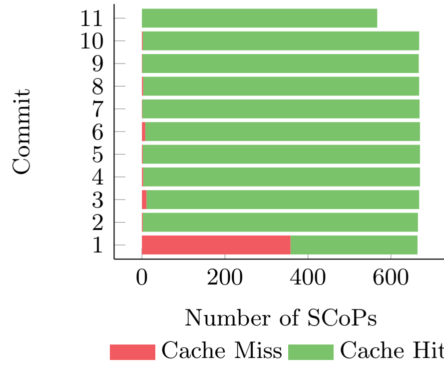


FIGURE 6.11: Cache Hits and Misses for each commit with advanced settings

tings (and no generalization). As depicted on the left of Figure 6.9, the compilation time decreases after the first commit by about 16% or 200 seconds. The optimization time decreases in most of the cases by about 95%, as depicted on the right of Figure 6.9. Apart from that, the distribution among the different steps of the cache is very similar to the distribution with default settings and no generalization. Import and collision checking are still the most expensive operations as one can see in Figure 6.10. The number of cache misses, as one can see in Figure 6.11, is very similar to the number of cache misses we have seen before. Nevertheless, the number of cache misses in the first commit is slightly higher than before. Figure 6.12 shows that the distribution of the SCoPs is again on a high level since the maximum bucket size is still three, although the number of SCoPs has increased. The quality is 1.044 which is within the range of what is considered as a good distribution.

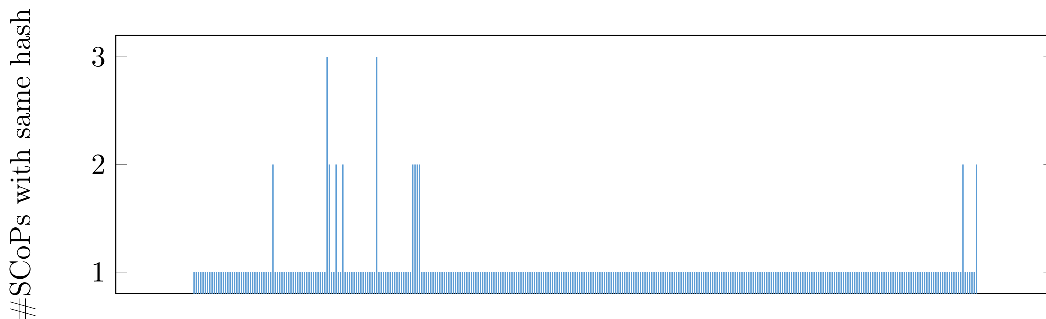


FIGURE 6.12: Hash function distribution for SCoPs with advanced settings

6.2 Test Case: LLVM Test Suite

Apart from a continuous integration environment, it is also interesting to know how the cache performs during the compilation of different software projects. This is for example important for servers, which build packages and distribute them to package management

systems, among others, in Ubuntu [20] and Arch Linux [21] . Such an environment is roughly simulated by the LLVM test suite, which provides an abundance of small programs. Our testing environment examined the performance of the cache for full generalization, generalization of induction variables and no generalization. Each case was repeated ten times. Five times with an initially empty cache and five times with a perfectly informed cache. The test suite was extended by the SPEC CPU2006 benchmark collection [22] and executed with the advanced settings as described in Section 6.1.2. With regard to full generalization it is important to note that for 70 test programs the compilation process did not come to an end within the time or memory limit. The same holds for 25 tests when generalization of induction variables was used.

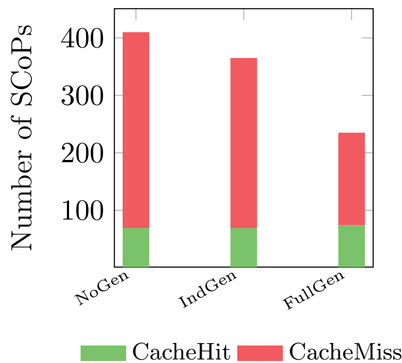


FIGURE 6.13: Cache hits and misses for different settings

Compared to FFmpeg the most noticeable difference is the number of cache hits with an initially empty cache as depicted in Figure 6.13. That was to be expected, since many different projects are compiled instead of one. As a result, there is a greater variety of different SCoPs to be considered. Without generalization the cache hits with an initially empty cache were only about 17%. With generalization of induction variables the result was slightly better with 19%. But since 25 tests did not compile within the time limit, the total number of SCoPs amounts to only 365, whereas without generalization there are 410. With full generalization at least 31,5% of the cache lookups are hits. But in this case only 225 SCoPs have been considered. With that in mind, the numbers for full generalization and generalization of induction variables should be treated with caution.

The pure optimization time of the schedule optimizer without the cache amounts to about 13,15 seconds, as one can see in Figure 6.14. With an initially empty cache and no generalization this time increases by about 20,15% to 15,8 seconds. This increase is mainly due to the time spent on storing the SCoPs in memory which can be improved for example by using a customized representation of the SCoPs instead of JSON. With a full cache, the time decreases by about 50,3% to 6,53 seconds. Taking generalization into account however, the cache again cannot compete with the schedule optimizer. With an empty cache and full generalization the optimization time amounts to 3489.46 seconds. If we only generalize the induction variables it still amounts to 2293.4 seconds. Even with a full cache the time is way beyond the time of plain Polly. Apart from that it remains to say, that these numbers only cover the SCoPs which have been optimized within the time or memory limit. Thus the full time needed for optimization is most

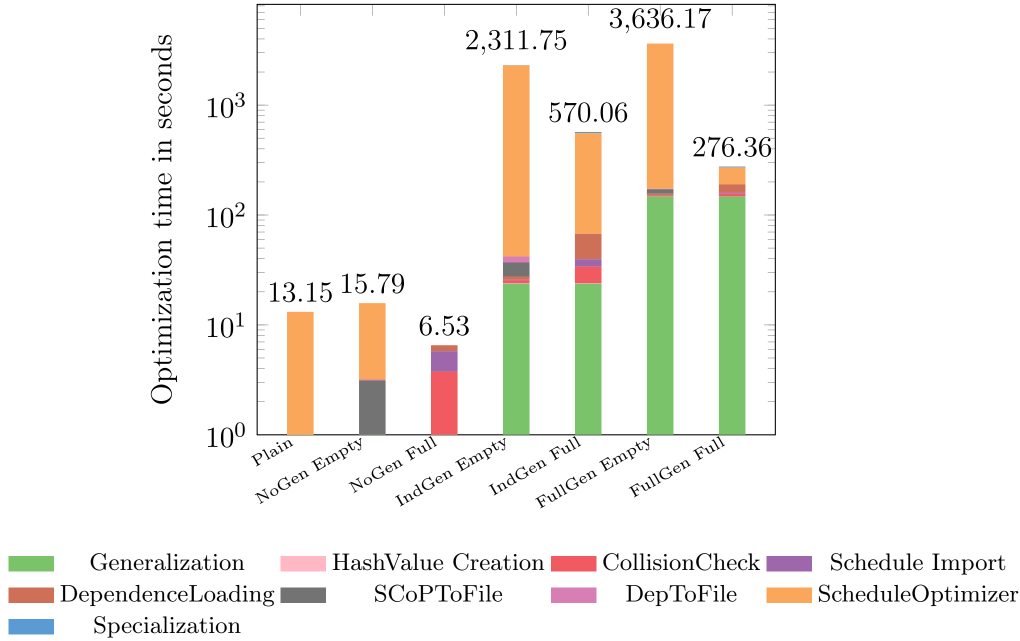


FIGURE 6.14: Optimization time for all different settings (Note that y-axis is logarithmic)

certainly much higher.

As in the case of FFmpeg, full generalization decreases the number of SCoPs that actually have been optimized by the schedule optimizer. Only ~33,5% of all stored SCoPs have been optimized. However, with generalization of induction variables we obtain a comparable result to the case of no generalization.

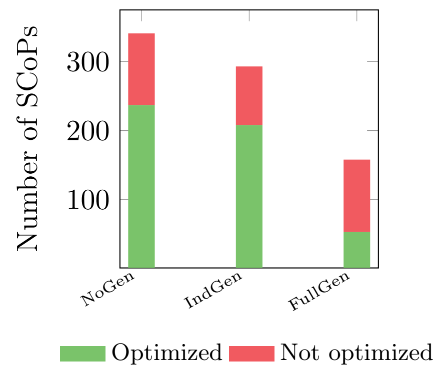
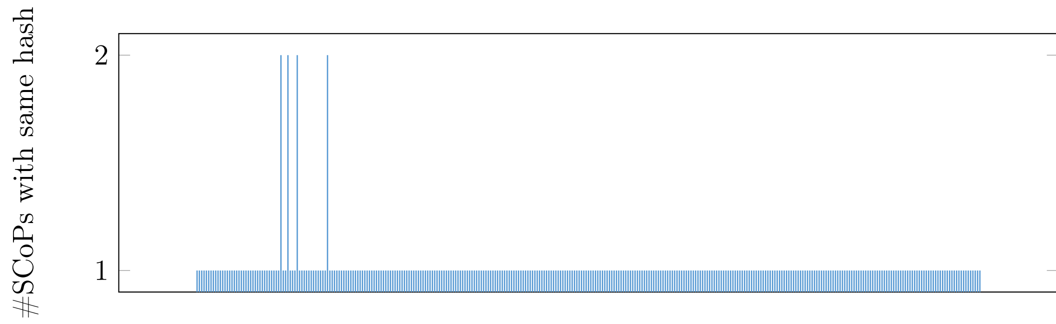


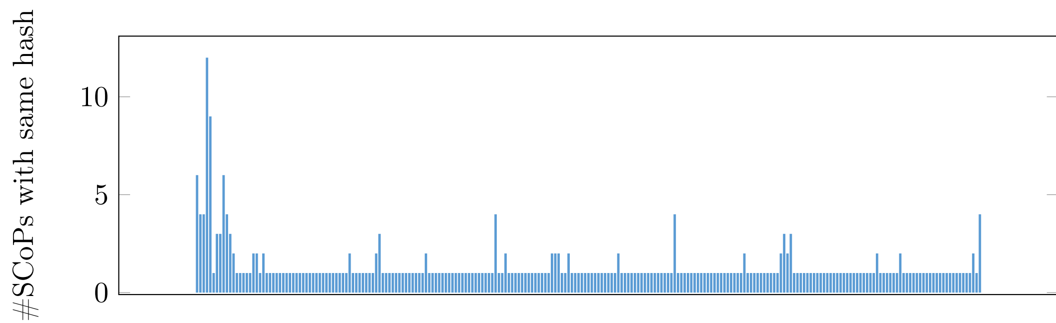
FIGURE 6.15: Optimized SCoPs in the cache for different settings

With generalization of induction variables 71% of the SCoPs in the cache have been optimized and with no generalization about 69,5% (Figure 6.15). Figure 6.16 shows the size of all buckets in the cache for all three generalization levels. Without generalization the quality of the hash function is very good since only four buckets contain more than one element and the size of these equals two. The quality of the distribution is 1.01, which is almost perfect. With generalization, the result is much worse, since some properties had to be omitted. For full generalization the quality is ~1.23 and for generalization of induction variables the result is ~1.65. Thus in the case of generalization the hash function produces way to many collisions.

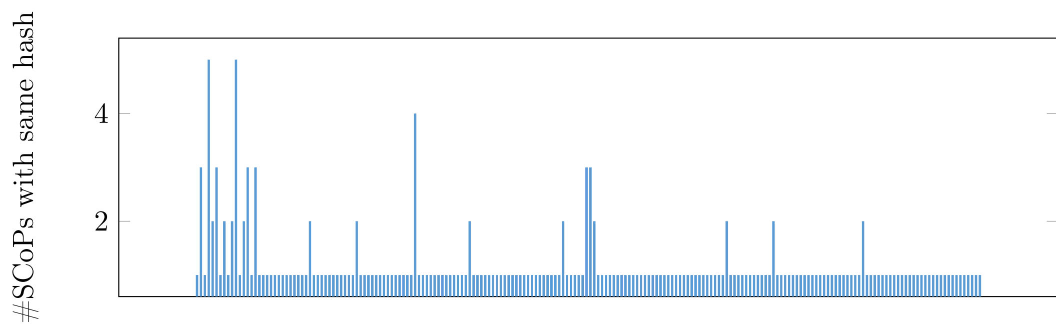
One remaining question is how much generalization affects the actual runtime of compiled code. To achieve this, we compared the runtime of all programs, that have been compiled



(A) Hash function distribution for non-generalized SCoPs



(B) Hash function distribution for SCoPs, that are generalized up to Induction variables



(C) Hash function distribution for fully generalized SCoPs

FIGURE 6.16: Hash function distribution for different settings and LLVM test suite

with and without generalization. Only those programs were considered that compiled within the time limit. The results are depicted in Figure 6.18 in the case of generalization of induction variables and in Figure 6.17 in the case of full generalization. With full generalization the overall runtime decreased from 816.59 to about 811.85 seconds. With generalization of induction variables the runtime decreased from 1026.49 to 986.36 seconds. Note that the overall runtime is higher with generalization of induction variables, since in this case more programs have been compiled within the time or memory limit. The result is very interesting, since generalization decreases the optimization quality of the schedule optimizer. However, this is possible, since Polly does not always improve execution times. Perhaps generalization prevented some bad optimizations which are not possible on the generalized SCoPs. The tests with the most significant difference in

the runtime for full generalization are Polybench/gramschmidt and 7zip. The tests with the most significant difference in the runtime for generalization of induction variables are 7zip and Polybench/floyd-warshall.

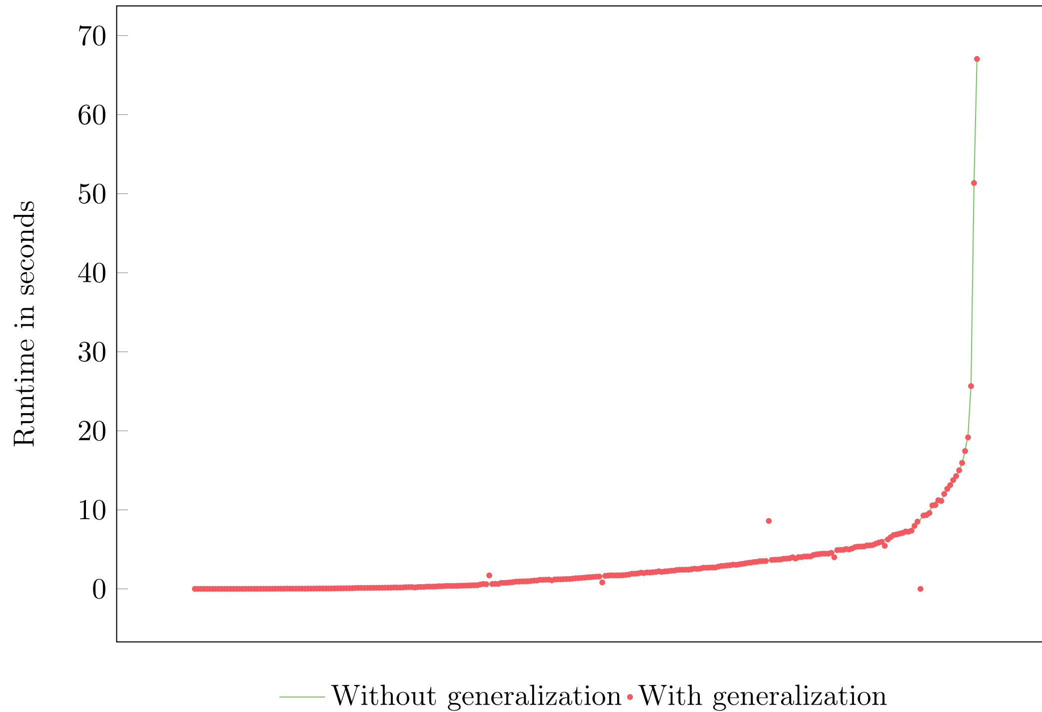


FIGURE 6.17: Comparison of runtime of programs compiled with full generalization and no generalization

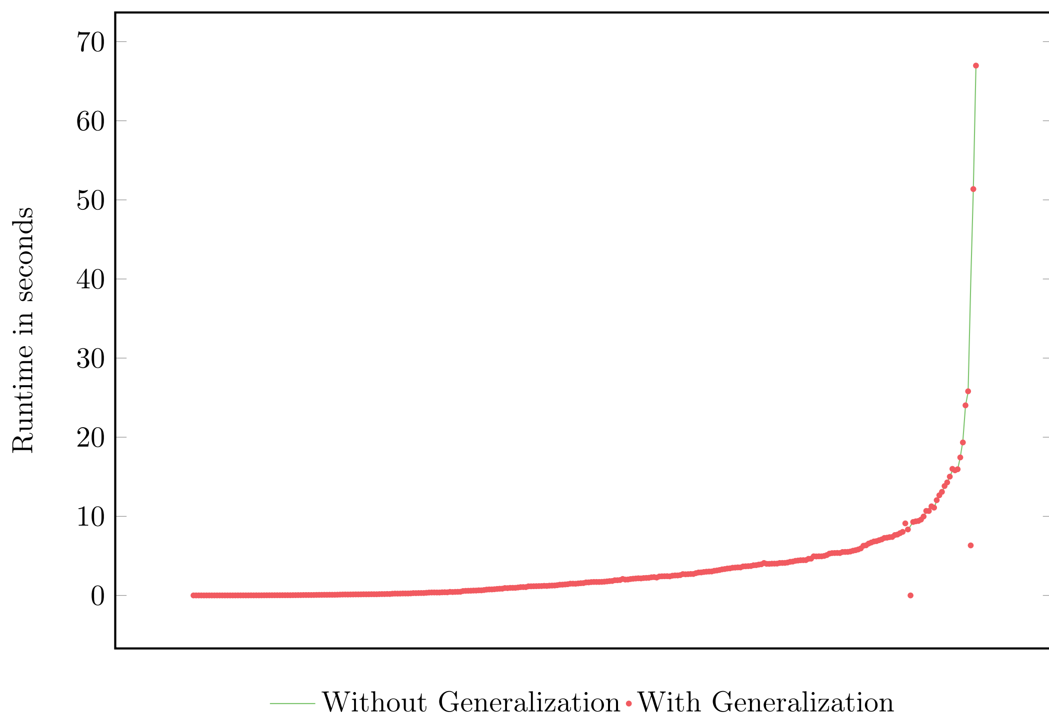


Figure 6.18: Comparison of runtime of programs compiled with generalization of induction variables and no generalization

Chapter 7

Future Work

Concerning the runtime, some techniques used in the current version of the cache are less-than-ideal. This chapter summarizes concepts, that could be used to further improve its performance.

With regard to equality checking it might be more efficient to do syntactic instead of semantic equality testing of the Presburger integer sets and maps. That is, to compare the string representation of integer sets and maps instead of checking for equality in isl, which is exponential in time. In that case, only the sets and maps of the input SCoP need to be converted to strings, too. Currently, the cache imports the SCoP and transforms its string representation to integer sets and maps in isl. Thus, if there are many collisions, there are many conversions to sets and maps. Furthermore there are many invocations of the expensive function is_equal(). However, if strings are compared, the input SCoP needs to be converted into a string only once. Thus there are no conversions from strings to isl sets and maps anymore. Then, the equality function only needs to replace the context variable names with the names of the context variables of the input SCoP and finally do a normal string comparison. This comparison should be sufficient in most cases, since isl normalizes integer sets and maps already to some degree. Thus the running time of the collision checker is only dependent on the size of these strings.

In case of generalization it is perhaps more sensible to do the generalization after collision checking. That is, instead of directly generalizing the input SCoP we could first search for the SCoP in the cache, by checking if the input SCoPs iteration domains, memory accesses and so on are a strict subset of the cached domains and memory accesses. If a SCoP is found in this way, we do not have to do generalization and specialization at all. We only need to apply generalization if the SCoP was not found in the cache. This still does not solve the increased runtime of the schedule optimizer. However, it might make the runtime of a perfectly informed cache more competitive. Thus, as an application it could be sensible to use generalization to fill the cache when time is irrelevant. Within a time-critical compilation process, we could only allow to read from the cache, not to

write new SCoPs into the cache.

Concerning the performance of I/O operations it is certainly not optimal to use JSON. It is currently used since it is already implemented by Polly to export and reimport SCoPs. Since the cache is a closed system, it is more sensible to use a customized format, maybe by using binary files and a more compact representation than it is constituted by JSON.

As already described in Section 4.1.3 the equality function performs much better if memory accesses and statements are ordered. Thus it is perhaps sensible to define an ordering on those in future versions of Polly.

Chapter 8

Conclusion

We presented a caching mechanism for Polly. It mitigates the overhead of Pollys internal schedule optimizer. The evaluation showed that the performance of generalization can not compete with the performance of the schedule optimizer. That is because the runtime for generalization, collision checking and specialization is far too slow. Maybe it is possible to reduce the number of generalizations as described in Chapter 7 in order to generate at least one sensible use case. Without generalization however, the performance is always much better with cache than without cache. In the case of FFmpeg and advanced settings, the overall compile time decreases by about 13,5%. Here the cache saves more than 200 seconds in most cases. Regarding the optimization time the results are even more convincing, since a full cache speeds up the optimization time by about 95%. With different programs, as in the case of the LLVM test suite, the difference is not that huge as in the case of FFmpeg. But even there Polly performs about 50% better with a full cache than without cache.
Apart from its caching performance, it is also very interesting to use the cache as a code duplication detector, especially when using generalization. With the help of the cache one can easily discover program parts that should be consolidated to one function.
In a nutshell one can say that the cache clearly improves the performance of Polly without restricting it in any way. Thus it is a sensible supplementation.

List of Figures

Figure 2.1 Example 4
Figure 2.2 Iteration domain 5
Figure 2.3 High level steps of Polly [2] 6
Figure 2.4 Two dimensional SCoP with three multidimensional memory accesses 6
Figure 2.5 Polyhedral Representation of the SCoP in Figure 2.4 6
Figure 2.6 Example array of size 3x2 in memory 7
Figure 2.7 A simple hashmap 8

Figure 4.1 High level steps of the caching algorithm 11
Figure 4.2 Example Cache content 12
Figure 4.3 One dimensional SCoP with two memory accesses 13
Figure 4.4 Hash value of the SCoP in Figure 4.3 13
Figure 4.5 SCoP properties and their occurrences in SCoPs of the LLVM testsuite 15
Figure 4.5 SCoP properties and their occurrences in SCoPs of the LLVM testsuite (continued) 16
Figure 4.6 Pseudo Code of the Equality Testing of two SCoPs 17
Figure 4.7 High level steps of the caching algorithm with generalization and specialization 18
Figure 4.8 Variation of the SCoP in Figure 2.4 18
Figure 4.9 Generalization of the SCoP in Figure 2.4 19
Figure 4.10 Generalization of the SCoP in Figure 2.4 19
Figure 4.11 Two SCoPs that are not equal, but the generalized hash function creates the same hash value for both 19

Figure 5.1 Code duplication 24
Figure 5.2 Possibly fix for the code duplication in Figure 5.1 24

Figure 6.1 Compile (left) and optimization time (right) with cache and no generalization and without cache 26
Figure 6.2 Cache hits and misses for each commit with default settings and no generalization 26

Figure 6.3 Distribution of the runtime among the different steps of the cache . 27
Figure 6.4 Hash function distribution for non-generalized SCoPs 27
Figure 6.5 Compile (left) and optimization time (right) with generalization, generalization of induction variables and without generalization 28
Figure 6.6 Distribution of the optimization time among the different steps of the cache for generalization of induction variables (left) and full generalization (right) 28
Figure 6.7 Cache hits and misses for each commit with generalization of induction variables (left) and full generalization (right) 29
Figure 6.8 Hash function distribution for fully generalized SCoPs and SCoPs that have been generalized up to induction variables 30
Figure 6.9 Comparison of the compile (left) and optimization (right) time of FFmpegs SCoPs with and without Cache and advanced settings 31
Figure 6.10 Distribution of the runtime among the different steps of the cache with advanced settings 31
Figure 6.11 Cache Hits and Misses for each commit with advanced settings . . 32
Figure 6.12 Hash function distribution for SCoPs with advanced settings . . . 32
Figure 6.13 Cache hits and misses for different settings 33
Figure 6.14 Optimization time for all different settings (Note that y-axis is logarithmic) 34
Figure 6.15 Optimized SCoPs in the cache for different settings 34
Figure 6.16 Hash function distribution for different settings and LLVM test suite 35
Figure 6.17 Comparison of runtime of programs compiled with full generalization and no generalization 36
Figure 6.18 Comparison of runtime of programs compiled with generalization of induction variables and no generalization 37

Figure A.1 Command line options for the caching mechanism 47

List of Tables

Table 4.1 Structure of the hash value 13
Table 4.2 Quality of (combinations of) SCoP properties 14

Table 6.1 The Testing Environment 25

Appendix A

Command line options

The cache supports several command line options in order to perform fine grained modifications on its behaviour. The command line options are listed in Figure A.1. *-cache-genlvl* defines the generalization level, where *none* disables generalization, *ind-vars* enables generalization of induction variables and *full* enables generalization of induction variables and memory accesses. With *-cache-scops* one can adjust if SCoPs shall be written to or read from the cache. By passing *none*, the cache neither writes SCoPs to the cache nor reads them from there. *w* forces the cache to only store SCoPs, while *r* only allows reading SCoPs from the cache. *rw* enables both, reading and writing. *-cache-deps* works analogous for dependences. *-cache-duplwgs* enables or disables code duplication warnings.

`-cache-genlvl=`
`{none, ind-vars, full}`
	`no generalization/`
	`generalization of induction variables/`
	`full generalization (default:full)`

`-cache-deps=` `{none, w, r, rw}`	`store/read/store & read` `dependences (default:rw)`
`-cache-scops=` `{none, w, r, rw}`	`store/read/store &` `read Scops (default:rw)`
`-cache-duplwgs=` `{true, false}`	`enable/disable code` `duplication warnings (default:true)`

Figure A.1: Command line options for the caching mechanism

Bibliography

[1] Lance Hammond Kunle Olukotun. The future of microprocessors. *Queue - Multiprocessors*, 2005. URL https://dl.acm.org/citation.cfm?id=1095418.

[2] Tobias Christian Grosser. Enabling Polyhedral optimizations in LLVM. Master's thesis, Department of Informatics and Mathematics, University of Passau, Passau, Apr 2011. *See* http://www.grosser.es/publicationss.

[3] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* http://llvm.cs.uiuc.edu.

[4] Performance comparison with and without polly from 01.08.2016. URL http://llvm.org/perf/db_default/v4/nts/90283?compare_to=90295.

[5] Asc sequoia benchmark codes. URL https://asc.llnl.gov/sequoia/benchmarks/.

[6] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *Proceedings of the 2006 GCC Developers Summit*, page 2006. Citeseer, 2006.

[7] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[8] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 343–352. ACM, 2010.

[9] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. *The Polyhedral Model Is More Widely Applicable Than You Think*, pages 283–303. Springer Berlin Heidelberg, 2010. URL https://link.springer.com/chapter/10.1007/978-3-642-11970-5_16.

[10] Llvm test-suite on github. URL https://github.com/llvm-mirror/test-suite.

[11] Ffmpeg main page. URL https://ffmpeg.org/.

[12] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[13] Sven Verdoolaege. *isl: An Integer Set Library for the Polyhedral Model*, pages 299–302. Springer Berlin Heidelberg, 2010. URL https://link.springer.com/chapter/10.1007/978-3-642-15582-6_49.

[14] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[15] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Computer Architecture: A Quantitative Approach*. Addison-Wesley, 1986.

[16] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems*, 2012. URL https://lirias.kuleuven.be/bitstream/123456789/359595/1/toplas.pdf.

[17] Denis Barthou, Paul Feautrier, and Xavier Redon. *On the Equivalence of Two Systems of Affine Recurrence Equations*, pages 309–313. Springer Berlin Heidelberg, 2002. URL https://hal.inria.fr/inria-00072302/document.

[18] Json main page. URL http://www.json.org/.

[19] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008.

[20] Ubuntu main page. URL https://www.ubuntu.com/.

[21] Arch linux main page. URL https://www.archlinux.de/.

[22] Spec main page. URL https://www.spec.org/cpu2006/.