Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2012

MSc THESIS

Simdization transformation strategies

Pepijn Westen

CE-MS-2012-12

Abstract

The Single Instruction Multiple Data (SIMD) paradigm promises speedup at relatively low silicon area cost for software that exposes a large amount of loop level parallelism. Automatic simdization–the act of exploiting loop level parallelism by issueing SIMD instructions that operate on multiple data elements at once– remains a daunting task for compilers, especially because SIMD instructions impose restrictions on the organization of the data elements. The CoSy simdization flow is able to effectively transform and simdize a program such that parallelism that is directly exposed in the inner loop is exploited, but it may not be able to exploit parallelism that is exposed at higher levels of a loop. This thesis proposes a new pass in the simdization flow that exposes such parallelism in a form in which it can be exploited. To achieve such transformations and determine which transformation should be applied, the polyhedral approach has been used. A polyhedral representation of loop nests allows for precise dependence analysis and application of transformations. To find the best transformation, we have proposed a cost estimation model that estimates the simdizability and cost of a transformation based on the polyhedral representation of the original loop nest. A prototype of this pass has been implemented, and using a number of tests the new flow has been shown to allow simdization in the CoSy flow where previously no simdization could take place.

Faculty of Electrical Engineering, Mathematics and Computer Science

Simdization transformation strategies
Polyhedral transformations and cost estimation

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Pepijn Westen
born in Den Haag, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Simdization transformation strategies[1]

by Pepijn Westen

Abstract

The Single Instruction Multiple Data (SIMD) paradigm promises speedup at relatively low silicon area cost for software that exposes a large amount of loop level parallelism. Automatic simdization–the act of exploiting loop level parallelism by issueing SIMD instructions that operate on multiple data elements at once– remains a daunting task for compilers, especially because SIMD instructions impose restrictions on the organization of the data elements. The CoSy simdization flow is able to effectively transform and simdize a program such that parallelism that is directly exposed in the inner loop is exploited, but it may not be able to exploit parallelism that is exposed at higher levels of a loop. This thesis proposes a new pass in the simdization flow that exposes such parallelism in a form in which it can be exploited. To achieve such transformations and determine which transformation should be applied, the polyhedral approach has been used. A polyhedral representation of loop nests allows for precise dependence analysis and application of transformations. To find the best transformation, we have proposed a cost estimation model that estimates the simdizability and cost of a transformation based on the polyhedral representation of the original loop nest. A prototype of this pass has been implemented, and using a number of tests the new flow has been shown to allow simdization in the CoSy flow where previously no simdization could take place.

Laboratory : Computer Engineering
Codenumber : CE-MS-2012-12

Committee Members :

Advisor:	Koen Bertels, CE, TU Delft
Member:	Hans van Someren, ACE Associated Compiler Experts
Member:	Stephan Wong, CE, TU Delft
Member:	Stefan Dulman, ES, TU Delft

[1]The work on this thesis was funded in part by the ASAM project (http://www.asam-project.org/)

Contents

List of Figures

List of Tables

Acknowledgements

This thesis project would not have progressed as smoothly without Sjoerd Meijer's help, insight and supervision. I would also like to thank Koen Bertels for his supervision and feedback. Furthermore, I would like to thank Hans van Someren, Christof Douma, Maarten Faddegon, Marius Schoorel and Martien de Jong for reading my thesis and providing valuable feedback. Finally, I would like to thank my parents for their support.

Pepijn Westen
Delft, The Netherlands
September 17, 2012

1 Introduction

While in the past decades CPU frequencies steadily increased at an exponential rate, in recent years this trend has stopped due to physical limitations. Although the increase in frequency has slowed down, feature sizes continue to shrink, allowing for more logic to be included on each die. This has opened up opportunities to speed up program execution through exploiting parallelism, i.e. executing parts of the program at the same time, rather than by increasing sequential execution speeds.

Different types of parallelism can be exploited using different architectures and paradigms. Parallel execution at the instruction level is often achieved through the superscalar execution, VLIW and SIMD paradigms, whereas parallelism inherent in high level algorithms might be better exploitable through multi core architectures.

The single instruction, multiple data (SIMD) class of computers uses instructions that operate on multiple data items in parallel, which can be used to exploit data parallelism. Early examples of such computers are the vector processor-based supercomputers.

While such vector processors have existed for a long time in the field of scientific computing, it was only in the last decade that many scalar general purpose and many application specific processors have been extended with SIMD Instruction Set Extensions (ISEs). This thesis will focus on these extensions rather than the broader class of SIMD computers. Examples of such extensions are the MMX, SSE 1-4 and most recently the AVX extensions for x86 architectures, the AltiVec and VSX extensions for PowerPC architectures and the NEON extensions for ARM architectures. SIMD extensions are often used in complement to the superscalar execution or VLIW paradigm.

These ISEs generally differ from the per-element pipelined execution of traditional vector processors, as enough transistors are available to execute operations on small vectors in parallel. Another difference compared to the traditional vector processors is that such SIMD ISEs do not generally support scatter/gather load and store operations, since doing so in parallel would require additional memory ports. Rather than employing multiple memory ports, SIMD ISEs often use an extended bus width to load entire vectors. Thus, the vector elements need to be stored in consecutive memory locations. Some SIMD extensions only support memory access aligned at the vector-width, requiring the compiler to handle alignment, whereas other SIMD ISEs allow unaligned memory access at a (sometimes considerable) performance penalty. To compensate for the aforementioned memory access restrictions, the extensions can provide a variety of (sometimes domain specific) packing, unpacking and permutation instructions, e.g. the NEON extension has instructions for deinterleaving and interleaving pixel data.

To exploit SIMD ISEs, SIMD instructions can be either be explicitly used in the program code or a compiler can try to infer where SIMD instructions can be used instead of scalar instructions. While the former option may yield more efficient code, it does require very specific knowledge of the target architectures and the resulting program

code is specific to those target architectures. The latter option allows for more portable code, but the programmer is in that case further removed from the process of simdization – the act of transforming the program to use SIMD instructions.

Because of the restrictions on alignment and the restriction that data elements must be stored at adjacent locations in the memory, automatically issuing SIMD instructions by a compiler is often quite an involved process that requires certain program transformations to be performed to expose data parallelism in such a way that it can be exploited using SIMD instructions.

In the following sections of this chapter, we will briefly describe our target architectures, present a problem statement and the thesis contributions. In chapter 2, we present some of the theoretical background of simdization and give an overview of the related work. In chapter 3, we describe the workings of simdization in the CoSy compiler development system and propose an improved simdization flow. In chapter 4 we document our test setup, present the results achieved with and without our proposed changes and analyse the results. Finally in chapter 5 we present our conclusions and suggest a number of possible improvements on our work.

1.1 Target architectures

One of the main goals in this thesis is improving the simdization capabilities of CoSy, a compiler development system that can be used to create optimizing compilers. Since CoSy is a compiler development system rather than a compiler, we will consider two different SIMD instruction set extensions for evaluation, viz. the SSE2 ISE and the AltiVec ISE. A CoSy implementation for an SSE2 code generator is already available, whereas the PowerPC code generator must be extended with AltiVec instructions. While these SIMD extensions share many features, such as the fact that they both have vector lengths of 128 bits, they also differ vastly in some aspects, e.g. the AltiVec set strictly follows the RISC paradigm, whereas SSE2 includes arithmetic instructions that can take one operand from a register and the other from memory.

In general, the AltiVec instructions can be divided into two categories: instructions that load from memory or store to memory, and instructions that take only vector registers or immediates (constants that are included as part of an instruction word) as operands and produce vector registers. The scalar and vector registers are strictly separated, and the only way to move data between them is through memory. A very powerful permute instruction is available, which can place any permutation of the contents of two vector registers into the result vector register. A select instruction can be used to multiplex the contents of two registers; this instruction can be used as a conditional move, and in some cases allows for conversion from control flow to straight code. Most of the standard ALU operations are supported for integers, and most standard single precision floating point operations are supported as well. The most notable missing operation is division. When performing a load or a store, the AltiVec memory unit simply drops the last 4 bits of the memory address, thereby effecting that every memory operation is 128 bit aligned, but also potentially loading different data than expected if the programmer or compiler does not account for this.

The SSE2 instruction set extension provides separate instructions for aligned and

unaligned loads and stores and has some arithmetic instructions which can take either a vector register or data from memory as it second operand. Using an unaligned address with an aligned load or store instruction will result in an exception. SSE2 provides many packing and unpacking instructions, but does not have a permutation instruction that is quite as powerful as that of AltiVec. A conditional move instruction is available. Both single and double precision floating point are supported, and unlike AltiVec, division instructions are also provided. Table 1.1 gives an overview of the distinctive features of both SIMD ISEs.

	SSE2	AltiVec
Extension to	X86, AMD64/Intel 64	PowerPC
CISC/RISC	CISC	RISC
Vector size	128 bits	128 bits
No. of SIMD registers	8 (X86) / 16 (AMD64/Intel 64)	32
Alignment	unaligned load instructions	alignment by software
Integer data types	8, 16, 32 and 64 bit integers	8, 16 and 32 bit integers
Single precision float	add, subtract, mul, div, sqrt, cmp	add, subtract, mul-acc, cmp
Double precision float	add, subtract, mul, div, sqrt	-

Table 1.1: AltiVec and SSE2 features

1.2 Problem statement

Recall that our main objective is to improve the automatic simdization capabilities of the CoSy compiler development system. For a compiler to be able to generate SIMD instruction code based on a scalar sequential specification in the form of C code, some transformation might need to be applied to the program.

To illustrate the importance of program transformations for simdization, a small Finite Impulse Response (FIR) filter is shown in Listing 1.1.

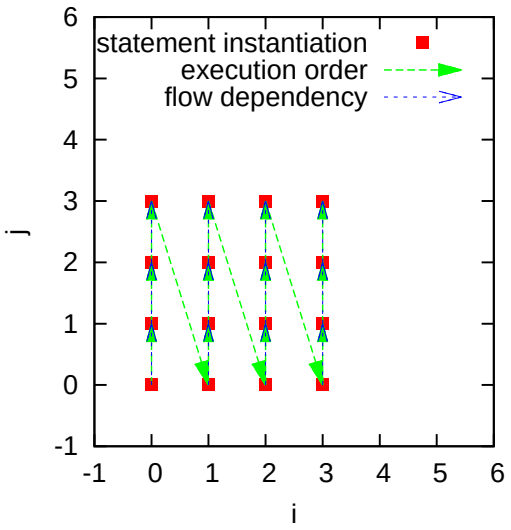
Listing 1.1: C code for convolution

```
void convolve()
{
        int i, j;
/* L0 */for (i = 0; i < N; i++)
        {
/* S0 */        y[i] = 0;
/* L1 */        for (j = 0; j < M; j++)
                {
/* S1 */                y[i] = y[i] + b[j] * x[i+j];
                }
        }
}
```

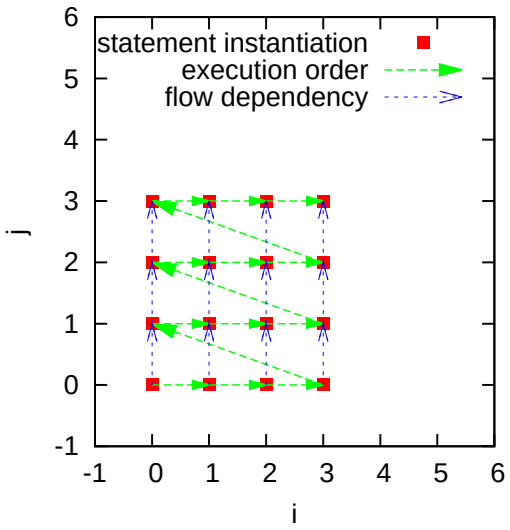
In this example statement S1 reads from array y at position i and writes to position i of the same array. The loop variable i is left unchanged in the inner loop; for all iterations of the loop marked as L1, the same element of y is read and written. This in turn means

that we cannot execute multiple consecutive iterations of loop L1 in parallel, since each iteration uses the result of the previous iteration. Figure 1.1a shows the iteration domain of S1 as a two-dimensional lattice, i.e. the values of i and j for which statement S1 is executed. The iteration values of j are shown on the vertical axis, and the iteration values of i are shown on the horizontal axis. The order of execution and the direction of a dependence from S1 to a previous instance of S1 are indicated by arrows between the iteration points. Note that arrows that indicate the execution order are aligned with the arrows that indicate dependences.

Figure 1.1b shows an alternative execution order that can be attained through a loop transformation, which is more amenable to simdization. In this alternative order, the loop nest iterates through all values of i before incrementing j. The advantage of this order is that multiple consecutive iterations of the inner loop, which now iterates with i, can be executed in parallel. Note that in this new order, the source iteration point of each dependence is still executed before the destination iteration point of that dependence, even though the arrows indicating execution order are perpendicular to the dependence arrows. The transformation that is applied in this example is the loop interchange transformation, which is one of many transformations that can be applied to loop nests. Although loop interchange may be beneficial in this specific example it may be detrimental in other cases, e.g. if the programmer had applied the interchange on the C code, interchanging the loops again would be counterproductive. Simdization of different programs may also require other transformations to be performed, such as scalar expansion, stripmining and loop fission. These transformations are explained in chapter 2.



(a) Domain, sequence and dependences of S1



(b) Domain, sequence and dependences of transformed S1

Figure 1.1: Characterization of the example from listing 1.1 for $N = 4, M = 4$

Compilers generally consist of a front-end, which translates the input code into an

intermediate representation (IR), a middle-end, where high level optimizations and program transformations are performed, and a back-end, where code generation takes place.

Loop transformations are generally best applied relatively early in the compilation flow, namely in the middle end of a compiler, whereas the actual cost of simdization is calculated only in the back-end.

The main problem we seek to address is to find in the middle-end how to transform a specific program to allow simdization, even though the actual simdization is performed at a later stadium of the compilation. To achieve this, we introduce a cost model that allows for comparing different transformations on specific loop nests in the program with regard to simdizability and performance in general, and enables the compiler to select the most beneficial transformation from a solution space. To limit the scope of this problem, we will only focus on whole statement simdization, where multiple instances of a single statement can be executed in parallel.

1.3 Thesis Contributions

An analysis of a large set of vectorization tests from the autovectorization subset of the GCC verification suite has shown that outer loop vectorization is one of the most promising techniques for improving the vectorization capabilities of the CoSy compiler development system. The main contribution of this work is to come up with a (heuristic) method of determining which transformations to apply in order to expose SIMD parallelism in an exploitable form. The need for such a solution is shown by two quite common applications: convolution–a very common operation in the digital signal processing domain–and matrix multiplication.

Of all possible loop transformations, we shall focus on two transformations in particular, namely stripmining and loop interchange. We limit the number of transformations because even for two transformations the number of combinations is large. These two specific transformations were chosen because they

- offer the ability to significantly increase simdizablitily of a loop nest
- can potentially improve cache locality when combined
- do not lead to excessively complex control flow

As part of implementing a prototype for the new transformation flow, the following work was done:

1. Formulation of stripmining and interchange as integer linear programming (ILP) systems and CLooG scattering functions

   By formulating the transformations as a system of equations, we can solve certain sub-problems using ILP solvers. These equations are also used to specify the transformations to CLooG, which can apply the transformations and generate code that scans the loop nest according to the transformed iteration order. In CLooG terminology, this transformation is called a scattering function. For an explanation of the polyhedral model and the function of CLooG, refer to section 2.4.

2. Transformation validity checking

   Before any transformation can be applied, it is necessary to verify that the transformation is correct with respect to the dependences in the loop nest. This can be done using an ILP solver and a series of ILP systems. The algorithm is described in subsection 3.6.2.

3. Definition of SIMD metrics and cost estimation of transformations

   In order to determine which transformation will yield the best performance, we issue a cost estimation per transformation. The cost estimation algorithm is described in 3.6.3.

4. Interface between CLooG and CoSy

   To use CLooG for application of transformations and generation of scanning code, an interface from CoSy to CLooG was implemented. This interface is used to supply the polyhedral specification of loop nests and the transformation. The interface also translates the CLooG representation of the scanning code back to the intermediate representation of CoSy.

5. Loop versioning

   As the transformed code often has non-constant loop bounds, a compiler pass was implemented that extracts a version of the inner loop with constant bounds, which is conditionally executed. This was done because a constant number of iterations is currently a requirement for successful simdization. This pass is briefly discussed in subsection 3.7.2.

6. Implementation of AltiVec extensions to existing PowerPC code generator

   A CoSy code generator for the PowerPC was extended with the capability to generate AltiVec instructions.

1.4 Conclusion

In this chapter, we have introduced SIMD and some of the problems encountered in simdization, provided a description and comparison of our two target architectures. We have formulated our problem statement, viz. determining which transformations to apply in order to expose SIMD parallelism that can be exploited inside the innermost loop in a loop nest, and provided an enumeration of our thesis contributions.

Background and related work on simdization

2

2.1 Introduction

In this chapter some of the basic concepts that are of import for simdization will be explained, followed by a brief introduction to the polyhedral model, and finally a summary of some recent work related to simdization and parallelization in general. In section 2.2 the basic concepts of data dependences are explained, along with some examples of how dependences within loop nests are commonly represented. Section 2.3 explains the loop transformations that are most important for simdization, how they affect and are affected by dependences, and how they may be combined. The basic concepts of the polyhedral model, iteration domains and polyhedral transformations are explained in section 2.4. Section 2.5 discusses related work in the field of simdization.

2.2 Data dependences

When one statement uses the resulting data from another statement, we generally say that the former depends on the latter. In imperative programming languages, dependences are often not explicitly defined in the code, but rather they are implied by the combination of execution order and data element accesses. Three types of dependences can be distinguished [17]:

- The true or flow dependence signifies that one statement reads the result previously created by another. A flow dependence from S_1 to S_2 is denoted as $S_1 \ \delta^f \ S_2$
- The anti dependence signifies that one statement writes a data element after another reads it. An anti dependence from S_1 to S_2 is denoted as $S_1 \ \delta^a \ S_2$
- The output dependence signifies that one statement writes a data element after another has written to it. An output dependence from S_1 to S_2 is denoted as $S_1 \ \delta^o \ S_2$

What these dependences have in common, is that they all specify a partial ordering of the instructions of the program [1]; if there is a dependence $S_1 \ \delta^f \ S_2$, S_1 must be executed before S_2 if the program is to remain correct with respect to the original specification in terms of output. Otherwise, if we have a program without control dependences, we may consider any program order that does not violate any dependences as a valid execution order. Control dependences occur when there is control flow (i.e. conditional branching) that depends on data that cannot be statically determined, e.g. variables or memory locations other than the loop variables. In some cases, a control dependence can be converted into a data dependence, e.g. by using conditional assignments. When

we consider dependences between statements inside a loop, the dependence can be on a statement executed in a previous iteration of the loop, or it can be on a statement executed in the same iteration. In the first case, the dependence is said to be loop carried, since it crosses the iteration boundary of the loop. Loop carried dependences can have a constant dependence distance, e.g. the statement a[i] = a[i-4] inside a single loop that iterates over i has a dependence distance of 4. Such a dependence might then also be denoted as $S_1\ \delta^f_{(4)}\ S_1$. In some cases, the dependence distance is not constant, but we are still able to determine the direction of the dependence. For example, we can denote a loop carried dependence as $\delta_{(<)}$, a loop independent dependence as $\delta_{(=)}$, a dependence of unknown direction as $\delta_{(*)}$. In loop nests of multiple levels, a vector can be used to represent the dependence distances or directions, e.g. $S_3\ \delta^f_{(0,2)}\ S_3$ indicates a distance of 2 in the inner loop, and a distance of 0 in the outer loop.

2.3 Program transformations

In loop-based simdization methods, certain transformations may be applied to loop nests to alter the execution order such that the transformed program can be simdized. This section discusses the loop transformations that we deem important for simdization.

2.3.1 Stripmining

If a loop has a loop control variable i that is incremented in each iteration with some constant value X, we can create inside this loop a new loop with control variable $i2$, which increments $i2$ by X and iterates Y times, and increment i with $X * Y$ in the original loop. The value Y is referred to as the strip size. The new loop is referred to as the strip loop, and the original loop is referred to as the control loop. This procedure does not in itself change the execution order of statements within the loop body, but combined with other transformations such as loop interchange and loop unroll, it can be very beneficial. Because it does not change the iteration order, stripmining cannot violate dependences. It does, however, introduce some added loop overhead. A small example of code before and after a stripmining transformation with size 4 is shown in Listings 2.1 and 2.2 respectively.

Listing 2.1: Example of code before a stripmining transformation

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

2.3.2 Loop unrolling

After a loop has been stripmined, it can be unrolled to increase the degree of instruction level parallelism and to allow basic block level simdization. The unroll transformation merges the loop bodies of several iterations of a loop into a single body, removing some

Listing 2.2: Example of code after a stripmining transformation

```
for (i = 0; i < N; i+= 4) {
        for (i2 = 0; (i2 < 4) && (i + i2 < N); i2++) {
                x[i+i2] = y[i+i2];
        }
}
```

of the overhead of iterating through the loop. The number of merged iterations is called the unroll factor.

If the loop bounds are not known at compile time or are not a multiple of the unroll factor, we need to separately handle the case where the unroll factor is larger than the number of actual iterations (remaining) to be executed. This can be done by making two versions of the loop: an unrolled version that is only entered if the remaining number of iterations exceeds the unroll factor, and a version that has not been unrolled.

Listing 2.3 shows the result of unrolling the inner loop of the code in Listing 2.2 with loop versioning.

Listing 2.3: Example of code after unrolling a loop

```
for (i = 0; i < N; i+= 4) {
        if (i + 3 < N){
                x[i+0] = y[i+0];
                x[i+1] = y[i+1];
                x[i+2] = y[i+2];
                x[i+3] = y[i+3];
        } else {
                for (i2 = 0; i + i2 < N; i2++) {
                        x[i+i2] = y[i+i2];
                }
        }
}
```

2.3.3 Loop interchange

As shown above, a combination of stripmining and unrolling can in some cases expose SIMD parallelism that can be exploited by basic block simdization. In other cases, however, stripmining and unrolling of the inner loop does not yield a loop body with statements that can be simdized, e.g. the code in Listing 2.4 cannot be simdized after stripmining and unrolling because of a dependence between consecutive iterations of the inner loop. In such cases, the loop interchange transformation can change evaluation order by interchanging loops at different levels of the loop nest, e.g. if a loop that iterates over a range of j is nested in a loop that iterates over a range of i, a loop interchange can produce a loop that iterates over i nested inside a loop that iterates over j. Since this transformation does change the order in which statements are executed, the validity of the transformation with regard to each dependence relation should be checked.

Listing 2.5 shows the loop nest from Listing 2.4 after a loop interchange transformation. The resulting code can then be stripmined, unrolled and simdized. The transformation in this example is legal, because the dependence of $S1$ to $S1$ is not carried by the outer loop of listing 2.4, i.e. the dependence of $S1$ on another iteration of $S1$ is contained by the iteration of i. Combining stripmining and loop interchange is often referred to

Listing 2.4: Example of a loop nest that can benefit from a loop interchange transformation

```
for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
/*S1*/          x[i] = x[i] + y[j];
        }
}
```

Listing 2.5: Example of an applied loop interchange transformation

```
for (j = 0; j < M; j++) {
        for (i = 0; i < N; i++) {
/*S1*/          x[i] = x[i] + y[j];
        }
}
```

as loop-tiling or loop-blocking, and can be used to optimize the memory access patterns to reduce cache misses [28] [29].

2.3.4 Loop fission

The loop interchange of the previous example can trivially be applied because the loop nest in question is perfectly nested, i.e. all statements are in the inner loop, and apart from the inner loop, each loop only contains one loop. The example in Listing 2.6 shows an imperfectly nested loop that could benefit from a loop interchange transformation, but because S1 is not inside the inner loop, the two loops cannot trivially be interchanged. The loop fission transformation splits off part of a loop body (which can consists of statements and nested loops), and inserts that part into a new loop with an iteration range identical to that of the original loop. This new loop is then executed either before or after the original loop is executed. Loop fission can be applied to create a perfectly nested loop from an imperfectly nested loop by fissioning all statements that are not in the body of the innermost loop into new loops outside of the original loop nest. Since all iterations of the part of the body that was fissioned into the new loop are executed either before or after the part of the body that remains in the original loop, the order of execution of the statements is altered by this transformation. Therefore the legality of such a transformation must be verified. Loop fission may in select cases be used to prevent cache trashing, yet it can also have adverse effects on cache locality. In the case of our example, we insert a duplicate of the outer loop before the original, and move

S1 into that duplicate. The result of applying the loop fission transformation on the example of Listing 2.6 is shown in Listing 2.7.

Listing 2.6: Example of an imperfect loop nest

```
for (i = 0; i < N; i++) {
/*S1*/  s[i] = 0;
        for (j = 0; j < M; j++)
        {
/*S2*/          s[i] = s[i] + x[j]
        }
}
```

Listing 2.7: Example of loop fission

```
for (i = 0; i < N; i++) {
/*S1*/  s[i] = 0;
}
for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++)
        {
/*S2*/          s[i] = s[i] + x[j]
        }
}
```

2.3.5 Scalar expansion

In the previous example, loop fission is applied to move a statement that writes an array, but it cannot be applied if the written variable is scalar. Listing 2.8 shows an example where a scalar is used, and as a consequence of the anti dependences $S1\ \delta^a\ S0$ and $S2\ \delta^a\ S1$, loop fission cannot be legally applied. In such a case, scalar expansion may be done, which expands a scalar value into a vector that is then indexed by the loop variable of the loop in which it is nested, thus creating a separate variable for each iteration of the loop and breaking any anti and output dependences carried by that loop. Naturally, this transformation should be used with care, since it does increase memory use and provides benefit only if it enables other beneficial transformations. By applying scalar expansion to a strip dimension, a limited expansion may be achieved. Listing 2.9 shows how the example of Listing 2.8 after scalar expansion for the dimension of loop variable i. After this scalar expansion, this imperfect loop nest can be fissioned into three perfectly nested loops.

2.3.6 Overview

In the context of automatic simdization, scalar expansion and loop fission can be regarded as transformations that enable other transformation to be used with a greater degree of freedom. Outer loop simdization, i.e. parallel execution of multiple iterations of a

Listing 2.8: Example of a scalar variable in imperfectly nested loop

```
int s;
for (i = 0; i < N; i++) {
/*s0*/  s = 0;
        for (j = 0; j < M; j++) {
/*s1*/          s = s + x[j];
        }
/*s2*/  y[i] = s;
}
```

Listing 2.9: Example of scalar expansion

```
int s_[N];
for (i = 0; i < N; i++) {
/*s0*/  s_[i] = 0;
        for (j = 0; j < M; j++) {
/*s1*/          s_[i] += x[j];
        }
/*s2*/  y[i] = s_[i];
}
```

statement of some outer loop through SIMD instructions, can be attained with the loop interchange transformation. A combination of stripmining and loop interchange can be employed to achieve outer loop simdization in a manner that potentially preserves or improves upon the cache locality of data accesses. Finally, unrolling the inner loop allows for basic block level simdization.

2.4 The polyhedral representation

In this section we discuss the polyhedral model of loops, which is a compact and powerful model to represent the iterations of statements inside loop nests. This model allows for analysis of loop nests on a more abstract level than is commonly possible in the syntactic representation, in which the loop is represented by a combination of control flow statements and manipulations of the loop control variables.

For the loop nest to representable in this manner, it is subject to certain constraints.

- The loop variable update value be constant, i.e. known at compile time.
- The loop bounds and control statement conditions must be a linear combination of loop invariants and (statically known) constants.

Such a loop nest is referred to as a Static Control Part (SCoP) [8]. In the polyhedral representation, each statement has a domain which can be represented as a polyhedron or a union of polyhedra. The domain of a statement is the set of all iterations of the loop nest in which the statement is executed. Each polyhedron describes a convex hull that contains the integer iteration points by defining the boundaries as a set of linear

(in)equalities. For example, the domain of $S1$ in the untransformed loop nest in listing 2.4 can be represented as $D_{S1} = \{< i, j > | 0 \leq i \leq N-1, 0 \leq j \leq M-1\}$, where N and M are static program parameters, and i and j are the loop variables. Triangular domains can also be represented, e.g. $D_{Sx} = \{< i, j > | 0 \leq i \leq N-1, i \leq j \leq M-1\}$.

A polyhedron represents which integer points are in the domain of the statement, but it does not specify any order of execution, which is implicitly assumed to be according to a lexicographic ordering of the points.

Lexicographic ordering simply means that we compare two vector points dimension by dimension (starting at the most significant dimension) until we find that one of the vectors has a smaller value, then that vector will precede the other lexicographically [17]. In this paper, we will assume that the lexicographic significance is according to the order of the dimensions from top to bottom. For example, $[0, 9]^T$ is lexicographically smaller than $[1, 0]^T$ since $0 < 1$, and $[1, 9, 9, 9, 9]^T$ is lexicographically greater than $[1, 0, 0, 0, 0]^T$ since $1 = 1$ and $0 < 9$. In the previous example of a polyhedral representation of D_{S1}, dimension i has greater lexicographical precedence than dimension j.

The iteration domain can be considered a normalised representation of the iterations of the loop nest, where each iteration point corresponds to an iteration of the original loop nest. The values of the original loop control variables for some iteration point can be computed as a function of the iteration point (i.e. the inverse function of the normalisation).

To be able to thoroughly analyze the loop nest, dependences need also be taken into consideration. For this reason, we also attach to each statement a list of all the memory elements it reads from and writes to, along with the indexing functions if the elements are arrays. If the indexing functions are linear combinations of constants and loop variables, the dependence distances can be precisely determined by formulating and solving Parametric Integer Linear Programming (PILP) problems[12]. For example, for each memory read, a set of PILP formulations is used to find the lexicographically maximal iterations of statements that write to the memory location, where the write statements must precede the read statement lexicographically. A PILP solver is able to solve this problem for the entire iteration domain of the statements, but due to the restriction that the source statement must lexicographically precede the destination, it must be formulated as a series of PILP problems. These PILP problems can be solved by a number of frameworks and libraries, such as ISL[27], PPL/BPL[2], and PIP/Piplib[11]. The polyhedral dependence analysis implementation in CoSy makes use of PIP to determine the precise dependences.

If a loop body contains multiple statements or loops, their execution order can be represented by adding an extra dimension to the iteration domain and encoding the execution order in that dimension [5].

2.4.1 Transformations on polyhedra

One of the great advantages of the polyhedral model is that complex transformations can easily be specified, verified and applied–although in terms of worst case computational complexity, the latter two are difficult problems. Unlike transformations in the classical representation, these transformations need not be applied to entire loops, but can be

applied to individual statements. A transformation, which we will denote as θ, is a mapping of iteration points in the original iteration domain of a statement to iteration points in a different space. The space into which the iteration points are mapped is often referred to as the scattering space or the scheduling space. The lexicographical order of iteration points in the scattering space denotes the execution order of the statements. Thus we can effectively change the execution order of the statements by transforming the polyhedra in this manner. Note that if two statements $S1$ and $S2$ from the same loop nest are transformed respectively with the two different transformations θ_{S1} and θ_{S2}, the transformations should transform the iterations points of the statements into the same scattering space, so that the order of the iteration points of the two statements can be determined.

Some transformations, like loop interchange, can be represented as a multiplication of the original iteration vector with a matrix. For example, a loop interchange transformation that maps from a 2-dimensional iteration domain into a 2-dimensional space with a different execution order, can be represented as

$$\theta_{S1}(\begin{bmatrix} i \\ j \end{bmatrix}) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix}$$

However, stripmining cannot be represented as a matrix multiplication. Instead, stripmining may be achieved by introducing extra variables to represent the control loop and the strip loop, and specifying extra constraints to establish their relation to the original dimension [5], e.g. to stripmine a loop variable i with size 8, we can introduce i_p and i_s, and we add the constraints $8 * i_p <= i <= 8 * i_p + 8 - 1$, $0 <= i_s <= 8 - 1$ and $8 * i_p + i_s = i$. These added variables can then be used in an affine scheduling function that establishes the lexicographical precedence of the dimensions of the scattering space, which can be specified as a set of linear equalities. The entire scattering function for a statement is denoted as θ, and is generally implemented in a matrix of (in)equalities. Such a matrix can be used to create ILP problem formulations and to generate code for the transformed loop nest.

Note that not every transformation necessarily results in a program that is equivalent to the original program in terms of output. To validate transformation that consists of a set of scattering functions θ_{S_1} .. θ_{S_k}, where S_1 .. S_k are statements in the loop nest, it must be verified that for each dependence $S_x\ \delta\ S_y \mid x, y\ \in\ [1..k]$, the transformed left hand side of the dependence $(\theta_{S_x} S_x)$ lexicographically precedes the transformed right hand side of the dependence $(\theta_{S_y} S_y)$.

2.4.2 Code generation

Code represented in the polyhedral model cannot be directly executed, and must first be translated back to a syntactic representation before it can be processed further by the compiler. Translating back from a polyhedral representation to efficient code is rather an involved process, the details of which are outside the scope of this thesis. The current state of the art polyhedral code generation algorithm is implemented in CLooG [5], which can also apply transformations on polyhedra.

Given a set of statements, polyhedral representations of the statement domains, and a set of scattering functions, CLooG is able to create scanning code for the transformed

polyhedral domains, i.e. a syntactic representation of a loop nest that iterates through the statements in the lexicographical order of the scattering space.

2.5 Related work

Now that we have introduced some of the basic concepts of (automated) simdization in the previous section, we will summarize some of the parallelization and simdization related work to provide some context and to show where our work is positioned.

2.5.1 Explicitly coded parallelism

For (heterogeneous) multi-core architectures, different solution paradigms are needed, especially when much of the parallelism is large grain parallelism. In those cases, the programmer may want to explicitly specify parallelism or even specify how exactly the parallelism is to be exploited.

SIMD parallelism may be explicitly exploited through the use of intrinsics, functions that operate on SIMD vector data types that correspond directly to SIMD instructions. Ökmen's simdization of a raytracing algorithm [20] makes use of this approach. The use of intrinsics allows for fine control of simdization, and can be used in complement with automatic simdization.

A more general alternative to intrinsics are Intel's array notations [16], which allow for using ranges instead of scalars when indexing arrays, resulting in a Fortran-like specification of data parallelism. This allows the programmer to write the program in a way that exposes parallelism, which should then be easily exploitable by a compiler for simdization as well as other types of parallelization.

For heterogeneous (distributed memory) multi-core systems, the Open Computing Language (OpenCL) [21] allows for exploiting both coarse grain and fine grain parallelism through the specification of parallel kernels. It is especially fit for targeting systems that are tailored for exploiting large quantities of data parallelism, such as GPU cores and the Cell architecture. Apart from SIMD parallelism, Single Program Multiple Data (SPMD) is an important type of parallelism in OpenCL.

2.5.2 Basic block simdization approaches

Because of the advent of modern SIMD ISEs, which tend to have smaller vector sizes and support a multitude of packing operations, Larsen and Amarasinghe introduced the concept Superword Level Parallelism (SLP) [18]. Unlike traditional vectorization techniques, which attempt to extract loop level parallelism based on loop transformations and loop-level dependency analysis, SLP aims to exploit data level parallelism at a basic block level, which is claimed by the authors to be a more robust. A basic block is a sequence of code without any control flow in between, so if the first statement is executed all the following statements in the basic block are guaranteed to execute as well, making dependency analysis somewhat simpler. Unrolling of loops can remove some of the control flow, and basically converts loop level parallelism into instruction level parallelism, some of which may be exploitable with SIMD instructions. To create the

appropriate SIMD instructions, SLP packs together isomorphic operations. The process of packing operations is rather like the process of scheduling of instructions for a VLIW architecture, and it also has the disadvantage of the large solution space that needs be explored to find an optimal solution. A 0-1 integer linear programming formulation has been devised, however the graph representing the search space for many of the industry standard SPEC95fp benchmarks proved to be too large to handle, and a heuristic algorithm was devised instead. The SLP algorithm was evaluated by targeting the AltiVec instruction set with a source-to-source compiler, which means that the algorithm operates on a reasonably high level. Reported speedups over compilation without vectorization range from 1.24 to 6.7, where the speedup was below 1.8 in 6 out of the 7 benchmarks.

Barik et al. have introduced a framework [3] that integrates a dynamic-programming based vector instruction selection algorithm with the scheduling and register selection. Like the SLP algorithm, this method operates on a basic block level, and can exploit opportunities to pack multiple scalar operations into a vector operation. The proposed algorithm is intended for dynamic compilation, and the experimental evaluation is done using the Jikes RVM. The authors report a very modest increase of up to 10% in compile time, and speedups of up to 1.57. Furthermore, an improvement of up to 13.78% over a Jikes implementation of the SLP algorithm is reported.

Shin et al. present a technique [24] to allow exploitation of SLP even in the presence of control flow, by applying if-conversion, which results in predicated instructions. After the if-conversion, the SLP algorithm can be applied. After exploitable SLP is extracted, the predicate of superword operations can be removed by using superword select instructions, and predicated scalar operations are converted back to regular control flow using an algorithm that attempts to minimize the number of branches. Note that in an instruction set that supports predicated execution, the latter conversions may not be necessary. Also note that the proposed method will generate code that always executes the vector instructions of both branches of an if-else construct, and also inserts an extra select operation (unless predicate execution is supported by the targeted SIMD ISE). To evaluate the technique, the original SLP implementation was extended with the additional algorithms, and the AltiVec ISE – which does not support predicated execution – was targeted. The extended SLP-CF method reliably resulted in increased speedups in the presence of control flow, with speedups ranging from 1.1 to 2.62, and an average 1.65. Significantly higher speedups were achieved for the same benchmarks with greatly reduced input sizes, which is interpreted as an indication that locality optimization may result in higher speedups for representative sample size.

2.5.3 Loop-based vectorization

The previously mentioned basic block level techniques are very effective in exploiting data parallelism in scalar operations, but due to their computational complexity they cannot be used on fully unrolled loop nests with large iteration spaces, since the solution space quickly becomes unmanageably large.

Traditional loop-based vectorization has been a topic of research almost since the advent of vector processors, and although modern SIMD ISEs differ from traditional vec-

tor processors in both functionality and implementation, these methods are still highly relevant in both research tools and state of the art production compilers. Unlike the basic block vectorization methods, loop-based vectorization allows for extracting hidden data parallelism from nested loops with large iteration spaces, without the need to fully unroll the loops. That being said, there are many transformations that may need to be performed to allow the compiler to successfully exploit SIMD parallelism. These transformations include loop interchange, loop peeling, etc. Which combination of transformations yields the best result depends on the specific loop-nest and the target architecture, and efficiently exploring the search space remains an active topic of research.

The loop based vectorization techniques can be used to expose parallelism that can be exploited with a basic block based simdization algorithm, so the techniques can be used complementary to each other.

Larsen et al. have proposed an algorithm for partial simdization in the context of software pipelining [19], which allows for improved resource utilization, as the scalar resources are also fully exploited. This is especially advantageous in superscalar and VLIW architectures and relatively small vector lengths. This optimization takes place in the back end of the compiler.

To be able to efficiently explore the parallelization possibilities in a loop nest, the polyhedral or polytope representation can be used to compose transformations [14]. An automatic polyhedral parallelizer and locality optimizer that makes use of loop tiling to generate OpenMP parallelisations of loop nests [6]. Trifunovic et al. have presented a framework and cost-model to explore this solution space in order to determine the cost of different combinations of transformations without need for actual code generation [26]. This solution is reported to perform well both in terms of speedup – when compared to fixed transformation strategies – and compilation time overhead. Bastoul has presented an algorithm to efficiently generate scanning code from a polyhedral representation [5]. CLooG, the library that implements this algorithm, is used in many frameworks, such as the Graphite framework in GCC [25] and Polly in LLVM [15].

2.5.4 Alignment

As mentioned before, SIMD ISEs often have rigid alignment constraints for load and store operations. To simdize a statement containing unaligned sub-expressions, the compiler may need to issue two aligned vector loads of consecutive addresses, and combine the two registers with a permute instruction in order to produce the unaligned vector. If all loads and stores in a statement have the same misalignment, a prologue loop can execute the statement as scalar until alignment is reached, and then the vectorized statement with aligned loads and store can be executed. However, this is not possible if a statement contains subexpressions with different misalignments. If a sequence of subexpressions operates at the same misalignment and the statement operates upon a stream of data, it may be advantageous to work with the aligned vectors, and realign them only when subexpressions with different misalignments need to be combined in an expression.

Pryanishnikov et al. proposed an interprocedural alignment analysis method [23] that can better determine alignment of objects passed as function parameters, which

can in certain cases remove the overhead of loop versioning or conservative alignment assumptions.

Eichenberger et al. proposed an algorithm and several heuristic policies intended to reduce the number of realignment operations [10].

Fireman et al. presented two algorithms [13] to compute a shift-optimal solution for the special case in which there are two different misalignments, and the case where each array only occurs once in the expression. For these cases they show that the previously proposed heuristics may deliver suboptimal results. These two special cases are claimed to cover most cases, but the presented evaluation is based on a set of generated loops.

S. de Smalen has presented an implementation of stream alignment within the CoSy compiler development system [9], and has shown through experiments that minimizing the number of realignments may not always result in the best performance.

The alignment analysis and optimization techniques discussed here can be used in complement with the loop based and basic block based simdization techniques.

2.6 Positioning of our work

While the SLP approach is appealing in its simplicity, it requires very aggressive unrolling to expose parallelism. Since the CoSy compiler development system already implements basic block simdization, we will focus on application of loop transformations in order to expose statement-level SIMD parallelism that can be exploited using basic block simdization without extensive unrolling. We restrict our target architectures to SIMD instruction set extensions, and focus on achieving simdizable memory strides. Similar to the work by Trifunovic et al., our approach explores combinations of strip-mining and loop interchange, and attempts to predict which transformation will result in the best performance. We make use of the polyhedral representation to determine the validity of transformations, analyse the impact of dependences on simdizability, apply the transformations.

2.7 Conclusion

In this chapter, we have described the basic concepts of loop dependences. We have given an overview of the loop transformations which are most important for exposing SIMD parallelism in the inner loop. Furthermore, we have touched on the polyhedral loop representation and its advantages. We have provided a broad overview of related work in the area of exploiting data parallelism, simdization and some of the subproblems of simdization.

Solution approach: exposing SIMD parallelism using stripmining and interchange

3

3.1 Introduction

In the previous chapter, we have provided an overview of the basic techniques used in simdization. In this chapter, we will describe how simdization is achieved in CoSy and provide an in-depth description of the proposed transformation approach. Section 3.2 provides a brief overview of the CoSy compiler development system. Section 3.3 contains a high level explanation of the current simdization flow and the proposed changes to the flow. The assumptions and restrictions for our approach are documented in section 3.4. Section 3.5 describes how we represent iteration domains, dependences and transformations. The transformation space, the verification of transformations and the cost estimation model and algorithms are explained in section 3.6. Finally, the application of the transformation and the subsequent loop versioning that is often necessary before basic block simdization are described in section 3.7.

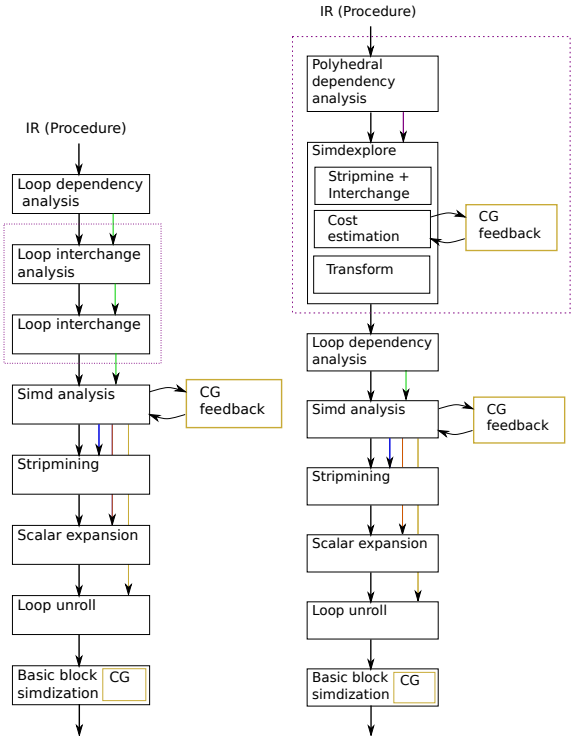
3.2 The CoSy compiler development system

Like most compilers, a CoSy compiler consists of a front-end, a middle-end and a back-end. The front-end translates the high-level program specification into an intermediate representation (IR), after which the middle-end perform high level transformations and optimizations. Representations and constructs not supported by the back-end may also be lowered away in the middle-end. Due to the need for retargetability, CoSy is a highly modular system. The passes are called engines, and generally each engine is specified to operate on some IR element, such as a basic block, a procedure or a compilation unit. An engine can make changes/transformations to the IR, or it can attach extra information that is used or consumed by other engines. For maximum flexibility, engines may provide options and target interfaces which allow for changing the behaviour of the engine respectively by specifying the desired behaviour through options or offloading decision making to a target specific implementation. The back-end of a CoSy compiler employs a rule based system to transform the middle-end IR (CCMIR) into a low level IR (LIR). The back-end furthermore offers a host of engines for tasks such as register allocation and scheduling. Eventually, the back-end will emit code, which is generally in assembly format. The architecture specific implementation of a back-end will henceforth be referred to as the code generator.

3.3 CoSy simdization flow

While simdization is rather target specific, it is generally best applied on a more abstract internal representation, i.e. before struct assignments are lowered, operator strength reduction is applied, and selects are converted into control flow.

Figure 3.1a shows the current CoSy simdization flow, which generally consists of engines that perform *loop dependency analysis*, *loop interchange*, *stripmining*, *scalar expansion*, and *loop unrolling*. These transformations have been briefly discussed in Chapter 2.



(a) The old flow (b) The new flow

Figure 3.1: The old and the new simdization flow

The engine *lcdepana* performs loop-carried data dependency analysis based on classic compiler theory [17] using tests such as the GCD test, and constructs a dependence graph with direction vectors for dependence relations. The engine *looplicana* performs a loop interchange analysis and attempts to achieve better exploitation of cache locality by advising the *loopinterchange* engine. The *simdana* engine performs SIMD analysis. Using code generator feedback, it determines if the code can be simdized and instructs the *stripmining*, *scalarexpand* and *loopunroll* engines. In the case that *simdana* determines that the code is simdizable, the result of this flow of engines will be that the innermost loops are unrolled by an appropriate factor to expose heterogeneous parallelizable instructions. The *simd* engine performs the basic block simdization. To achieve this, it performs a search of possible operation packings and rewrites the packed operations into

Compiler Known Functions, using rules defined as part of the code generator. The code generator feedback mechanism that is used by the *simdana* engine is generated from the same rules.

While the simdization flow shown in Figure 3.1a is effective in the cases where the inner loop is parallelisable, the *looplicana* engine does not optimize for simdizability, and as such its transformation advice may be detrimental to simdizability. For example, an inner loop carried flow dependence with distance one is very good for memory re-use, while it also represents a dependence chain that blocks inner-loop simdization, since operations that are interdependent cannot be performed in the same SIMD instruction. To solve this problem, as also introduced in Section 1.2, a new transformation flow shown in Figure 3.1b will be proposed to replace *looplicana* and *loopinterchange* with the engine *simdexplore*. This engine explores a subspace of stripmining and loop interchange transformations in order to find the best transformation, which may or may not be one that is simdizable. In this way, we solve the transformation flow order problem.

To determine the feasibility of such transformation and to apply them, a polyhedral representation of the loop nest is employed. A prototype for polyhedral analysis is already available in the CoSy framework. Using this polyhedral model, the correctness of each transformation is verified, and for each correct transformation, a cost estimate is produced. After selecting the best transformation based on the cost estimates, the loop nest is rewritten to the transformed loop nest. The regular simdization flow can then proceed as before. Like the *simdana* engine, this process also makes use of code generator feedback from the *simd* code generator rules. Note that in the long term, it may be desirable to integrate the functionality of *simdexplore* and *simdana*, since there is a large overlap in functionality. The reason this was not done in the first place was to limit the complexity of the implementation due to time limitations.

In summary, the new flow consists of three parts: the polyhedral modeling of the loop nest, the exploration of the transformation space and finally the application of the most beneficial transformation and the simdization itself.

3.4 Assumptions

Certain choices have been made in the implementation and in the internal modeling of information which limit the applicability of the simdization flow by imposing certain restrictions.

A number of restrictions were inherited from the CoSy polyhedral loop analysis prototype, which is used to characterize domains as polyhedra and to determine precise dependence vectors. The first of these restrictions is that the bounds of any loop dimension must be representable as a linear function of the surrounding loop dimensions, i.e. the dimensions of lexicographically higher precedence. While parametrised polyhedra are supported by many polyhedral manipulation libraries and even by CLooG, parametrised domains are not supported in the prototype. Although support for parametrised loop nests would allow the analysis to accept a far greater number of input loops, simdization of such polyhedra would require some knowledge of the boundaries anyway. In many cases, such knowledge could be acquired though interprocedural analysis, so support for parameters might still provide benefit.

Furthermore, for the dependence vectors to be precisely determined, the memory indexing functions must be linear combinations of the loop variables.

Some further limitations were introduced by the specification of the internal model of the stripmining and interchange transformation, the most significant of which is that only perfect loop nests can currently be transformed, and the strip size of the stripmining operation is assumed to be the same for each stripmined loop. While a similar exploration scheme to the one proposed in this thesis is also feasible in non-perfectly nested loops, this requires a more elaborate transformation representation and implies an expansion of the exploration space, since loop fission would also have to be included in the transformation space. This decision to limit our model to perfectly nested loops was made to limit the complexity of both the model and the implementation. Scalar expansion in order to circumvent anti-dependences is not included in the implemented simdization flow. No use is made of associative or commutative properties of operators, even though this might expand the set of valid transformations in specific cases where a reduction with such operators occurs. This decision was made to limit the scope of the project. In our implementation we have focused mainly on simdization of floating point operations, and these special properties do not hold for floating point addition, due to (accumulation of) rounding errors.

Furthermore, during cost estimation, little information on iteration counts in the scattering domain is known or used, since cost estimation is performed without actually applying the transformation. Since the number of iterations in an innermost loop may depend on the loop variables of any of the outer loops, it is difficult to reason about the iteration count of the inner loop in the general case. While some polyhedral tools and libraries offer facilities for counting integer points [7] [4] in a polyhedron, there was insufficient time to look into these techniques and the implications of using them.

3.5 Polyhedral representation

Before the possible transformations can be explored, we must first establish the polyhedral representation of the loop nest. In this section, we will describe some of the models and representations which are used in the transformation and cost estimation.

3.5.1 Iteration domains and their polyhedral representation

Recall from section 2.4.1 that the iteration domain of a statement can be modeled as a polyhedron. In this section we will describe how such a polyhedron is represented. Polyhedral libraries, parametric ILP solvers and tools commonly make use of matrices to represent a set of equalities and inequalities. The use of this representation allows for easy interfacing with tools, and as such, this representation was also adopted in CoSy's polyhedral analysis prototype to represent iteration domains, conditions and dependence functions.

In this matrix representation, the first column signifies whether the row is an inequality or an equality, followed by one column for each variable, followed by one column for each parameter, and finally a column which contains the constants of the equations. The variables represent the loop control variables in the loop nest. As

mentioned in the previous section, parameters are not used in the current prototype, so there are no parameter columns in our representation. Each equality or inequality is a row in a matrix. These rows have the form $[ineq, c_0, c_1, c_2, .., c_k, c_{const}]$. In case of an equality, the element in the first column will be 0, and the row represents the constraint $[c_0, c_1, c_2, .., c_k, c_{const}] \cdot [v_0, v_1, v_2, .., v_k, 1] = 0$. For inequalities, the first element in the first column is 1, and the row represents the constraint $[c_0, c_1, c_2, .., c_k, c_{const}] \cdot [v_0, v_1, v_2, .., v_k, 1] \geq 0$.

For example, the polyhedron $D_{S1} = \{< i, j > | 0 \leq i \leq N - 1, 0 \leq j \leq M - 1\}$, which models the iteration domain of statement $S1$ from the example 2.4, could be represented with the following matrix:

$$\begin{bmatrix} ineq/\overline{eq} & i & j & N & M & c \\ \hline 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & -1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & -1 \end{bmatrix}$$

Recall, however, that such a parametric representation is currently unsupported by our prototype, and as such we require that N and M are compile-time known values. The same domain, but with $N = 16$ and $M = 24$ could be represented with the following matrix:

$$\begin{bmatrix} ineq/\overline{eq} & i & j & c \\ \hline 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 15 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 23 \end{bmatrix}$$

The columns are ordered from left to right by decreasing lexicographical precedence of the variable. This specific matrix representation is used in the interfaces of PIP, PolyLib and CLooG.

3.5.2 Dependences

Dependences are modeled as a source statement, a destination statement, a dependence type, a matrix specifying the conditions under which the dependence exists, and a matrix specifying the dependence function, with each row specifying one element in the dependence vector. Note that the dependence vector may be specified either as a function from an iteration point in the destination domain to an iteration point in the source domain, or as a function from an iteration point in the source domain to an iteration point in the destination's domain. In our representation, the dependence vector is a function from the destination statement's domain to the source statement's iteration domain.

The dependence type specifies if it is a flow dependence, an anti-dependence or an output dependence. The conditions specify the constraints on the domain of the dependence destination under which the dependence exists, and will henceforth be referred to as *constr*.

For example, let us again look at statement $S1$ from listing 2.4. Note that $S1$ both reads and writes $x[i]$, whereas the inner loop iterates through the loop dimension of variable j. This means that $S1$ depends on the previous iteration of the inner loop, provided that $j > 0$. The dependence vector is in this case $[i, j - 1]$, which is represented by the following matrix:

$$\left[\begin{array}{c|ccc} ineq/\overline{eq} & i & j & c \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{array}\right]$$

where each row represents a linear function from the point in the domain of the dependence destination to the iteration point in the source iteration domain, and the rows are ordered according to lexicographic precedence.

The condition that $j > 0$ is represented by the following matrix:

$$\left[\begin{array}{c|ccc} ineq/\overline{eq} & i & j & c \\ \hline 1 & 0 & 1 & -1 \end{array}\right]$$

Which is a restriction that applies to the iteration domain of the dependence destination.

In contrast to a direction vector, the dependences are precisely specified; when presented with an iteration point in the domain of the dependence destination, the dependence vector can be used to calculate the precise iteration of the source statement.

There may be multiple dependences with different conditions or dependence functions between a pair of statements.

3.5.3 Transformation

While both BPL/PPL and CLooG – which are used to solve ILP problems and generate loop nest code respectively – represent transformations as a set of equalities and inequalities, such a representation is inconvenient for representing the transformation internally for cost estimation.

For this reason, a simpler representation was devised. This representation is limited by the assumption that we only apply transformations on perfectly nested loops, the only transformations that are applied are stripmining and loop interchange, and the stripmining size is shared for each loop. Furthermore, each dimension can be stripmined at most once, because as far as we know there are no benefits to be gained from stripmining a single dimension multiple times.

With these assumptions, we can represent each loop dimension using a unique integer as identifier according to the original lexicographical precedence of that loop. We have made the arbitrary decision to identify the loops with ascending numbers starting with the outermost loop towards the innermost loop. To represent the stripmining transformation, we can then use even numbers to represent the control loops, and odd numbers to represent the strip loops. If we order these identifiers by lexicographical precedence in the scheduling space, we can represent any combination of stripmining and interchange as a sequence of integers.

For example, stripmining without interchange of a loop nest of depth k can be represented as $[0, 1, 2, 3, .., 2k-2, 2k-1]$, where 0 and 1 respectively represent the control loop and the strip loop derived from the outermost loop in the original nest and $2k - 2$ and $2k - 1$ represent the control loop and the strip loop derived from the innermost loop in the original nest.

Since stripmining without interchange does not change the execution order of iteration points, it introduces overhead without any benefit. For this reason, we do not stripmine a dimension if the control loop and strip loop derived from the same loop in the original domain appear as consecutive elements in this sequence. As such, the previous example actually represents a transformation which neither changes the execution order nor introduces extra dimensions, i.e. a transformation that does not change the loop nest. Because the ordering of the identifiers in this sequence represents the lexicographical ordering of the dimensions in the scheduling domain, we can represent loop interchange by permuting this sequence of identifiers.

The advantage of this representation is that the transformations can easily be generated in order to explore the possibilities, and that relevant information for cost estimation can easily be extracted from it. On the other hand, it is not as flexible as a more general representation such as a CLooG scattering function, which can also represent a host of other transformations.

3.6 Transformation exploration

Now that we have defined a way to model a transformation, we will construct a set of transformations which is worth exploring, while ideally leaving out those transformations that cannot be beneficial to simdization. Then we will propose a method to estimate a cost for each transformation, which can be used to find which transformation is most beneficial.

3.6.1 Exploration space

If we consider a loop nest with a depth of 2, stripmining each dimension once will yield a loop nest of depth 4, as each loop of the original loop nest will be split into a control loop and a strip loop. In that case it is still feasible to explore all possible permutations of control loops and strip loops, given that we use a fixed stripmining size. For depth $d = 2$, stripmining yields a scattering dimensionality of $2 * d = 4$, which can be permuted in $(2 * d)! = 24$ different orders.

As an example, listing 3.1 shows an untransformed loop nest of two dimensions, listing 3.2 show the same loop nest with both dimensions stripmined with a strip size of 4, and listings 3.3 and 3.4 show two of the possible permutations of the stripmined loop nest.

Using this full exploration scheme, a loop nest of depth $d = 4$ would yield $8! = 40320$ permutations, which is too many to handle at compile time. Furthermore, many of these transformations are ineffective in the context of simdization. Since the memory indexing functions are linear combinations of the loop variables with integer constants

Listing 3.1: Untransformed

```
for (i=0; i<16; i++)   //0,1
 for (j=0; j<32; j++) //2,3
  x[i+j] = 0;
```

Listing 3.2: $\theta = [0, 1, 2, 3]$

```
for (i=0; i<4; i++)          //0
 for (i2=0; i2<4; i2++)      //1
  for (j=0; j<8; j++)        //2
   for (j2=0; j2<4; j2++)    //3
    x[4*i+i2+4*j+j2] = 0;
```

Listing 3.3: $\theta = [0, 2, 3, 1]$

```
for (i=0; i<4; i++)          //0
 for (j=0; j<8; j++)         //2
  for (j2=0; j2<4; j2++)     //3
   for (i2=0; i2<4; i2++)    //1
    x[4*i+i2+4*j+j2] = 0;
```

Listing 3.4: $\theta = [3, 2, 1, 0]$

```
for (j2=0; j2<4; j2++)       //3
 for (j=0; j<8; j++)         //2
  for (i2=0; i2<4; i2++)     //1
   for (i=0; i<4; i++)       //0
    x[4*i+i2+4*j+j2] = 0;
```

as coefficients, it is not possible for any control loop to be more simdizable than the strip loop mined from the same dimension. Furthermore, the absolute memory stride of a control loop is equal to or greater than that of the strip loop mined from the same dimension, and therefore nesting the control loop inside the strip loop is unlikely to improve cache locality.

In fact, in order to perform outer-loop simdization, it is sufficient to stripmine a single dimension and move the strip to the inside of the loop nest. Similarly, loop blocking is achieved by stripmining a small number of dimensions and moving the strips to the inside of the loop nest.

Since a loop blocking scheme both allows for outer loop simdization and for potentially improving cache locality, the decision was made to limit the exploration space for loop nests of depths greater than or equal to 3 to loop blocking.

More specifically, the exploration space consists of the union of the sets of transformations generated by stripmining three, two and one dimensions, and moving the strips to the inside of the loop nest. Selecting three loops for stripmining, means that we start by selecting 1 out of d not stripmined loops, then we select 1 out of the remaining $d-1$ not stripmined loops, and finally we select 1 out of the remaining $d-2$ not stripmined loops. This yields $\frac{d!}{(d-3)!}$ different transformations. These are called the k-permutations of d, with $k = 3$. The k-permutations are all permutations of the k-combinations of d, i.e. $k!\binom{d}{k}$. Taking the k-permutations with $k = 3$, $k = 2$ and $k = 1$ results in a total of $\frac{d!}{(d-3)!} + \frac{d!}{(d-2)!} + d$ transformations.

As an example, listing 3.5 shows an untransformed loop nest of three dimensions, two different transformations from the 2-permutations are shows in listings 3.6 and 3.7, and listing 3.8 shows one of the possible transformations from the 1-permutations.

In order to generate the transformations in this transformation space, a compact integer representation for the permutations was devised from which the proposed representation of a transformation can be easily generated. In this representation, the numbers in the range $[\ 0,\ \frac{d!}{(d-3)!} - 1\ ]$ represent the possible 3-permutations, the numbers in the range $[\ \frac{d!}{(d-3)!},\ \frac{d!}{(d-3)!} + \frac{d!}{(d-2)!} - 1\ ]$ represent the different 2-permutations and the numbers

Listing 3.5: Untransformed

```
for (i=0; i<16; i++)      //0,1
 for (j=0; j<16; j++)     //2,3
  for (k=0; k<16; k++)    //4,5
   y += x[i+j+k];
```

Listing 3.6: A transformation from the 2-permutations

```
for (i=0; i<4; i++)              //0
 for (j=0; j<16; j++)            //2,3
  for (k=0; k<4; k++)            //4
   for (k2=0; k2<4; k2++)        //5
    for (i2=0; i2<4; i2++)       //1
     y += x[4*i+i2+j+4*k+k2];
```

Listing 3.7: A transformation from 2-permutations

```
for (i=0; i<4; i++)              //0
 for (j=0; j<4; j++)             //2
  for (k=0; k<16; k++)           //4,5
   for (j2=0; j2<4; j2++)        //3
    for (i2=0; i2<4; i2++)       //1
     y += x[4*i+i2+k+4*j+j2];
```

Listing 3.8: Transformation from 1-permutation

```
for (i=0; i<16; i++)      //0,1
 for (j=0; j<4; j++)      //2
  for (k=0; k<16; k++)    //4,5
   for (j2=0; j2<4; j2++) //3
    y += x[i+4*j+j2+k];
```

in the range $[\ \frac{d!}{(d-3)!} + \frac{d!}{(d-2)!} - 1,\ \frac{d!}{(d-3)!} + \frac{d!}{(d-2)!} + d\ ]$ represent the 1-permutations.

3.6.2 Transformation verification

Since the transformations may alter the order in which loop statements are executed, it is necessary to check that the partial order specified by data dependences is not violated. As explained in chapter 2, a transformation θ transforms the original domain D into a scattering domain, which like the original domain is assumed to be ordered lexicographically.

Each dependence consists of a set of conditions *constr*, a pair of statements (S_{source}, S_{dest}), and a function $f(\bar{x})$ where $\bar{x}$ is a point in the original iteration domain of S_{dest} and $f(\bar{x})$ is a function from x to a point in the original iteration domain of S_{source}, and where $\bar{x}$ is further constrained by *constr*. For the partial order specified by the dependence to be satisfied by the transformation, $\theta f(\bar{x})$ must lexicographically precede $\theta\bar{x}$ for any point $\bar{x} \in D_{S_{dest}} \cap constr$.

For example, if we have statements S_{source} and S_{dest} which both have iteration domain $D = \{< i, j > | 0 \leq i \leq 8, 0 \leq j \leq 8\}$, and we have the function $f(\begin{bmatrix} i & j \end{bmatrix}^T) = \begin{bmatrix} i & j-1 \end{bmatrix}^T$ and the condition $j > 0$, figure 3.2 illustrates how a transformation is applied to a single point in case of a loop nest of two dimensions. In the illustration, point $\begin{bmatrix} i & j \end{bmatrix}^T = \begin{bmatrix} 6 & 3 \end{bmatrix}^T$ from the domain of S_{dest} reads data that is written at point $r\begin{bmatrix} 6 & 2 \end{bmatrix}^T$ by S_{source}. The transformation that is applied is $\theta(\begin{bmatrix} i & j \end{bmatrix}^T) = \begin{bmatrix} j & i \end{bmatrix}^T$. Since $3 > 0$, we know that $\theta f(\begin{bmatrix} 6 & 3 \end{bmatrix}^T) = \begin{bmatrix} 2 & 6 \end{bmatrix}^T$ should lexicographically precede $\theta \begin{bmatrix} 6 & 3 \end{bmatrix}^T = \begin{bmatrix} 3 & 6 \end{bmatrix}^T$ for the transformation to be valid. In fact, 2 is lexicographically smaller than 3, which tells us that the transformation is valid with regard to this dependence.

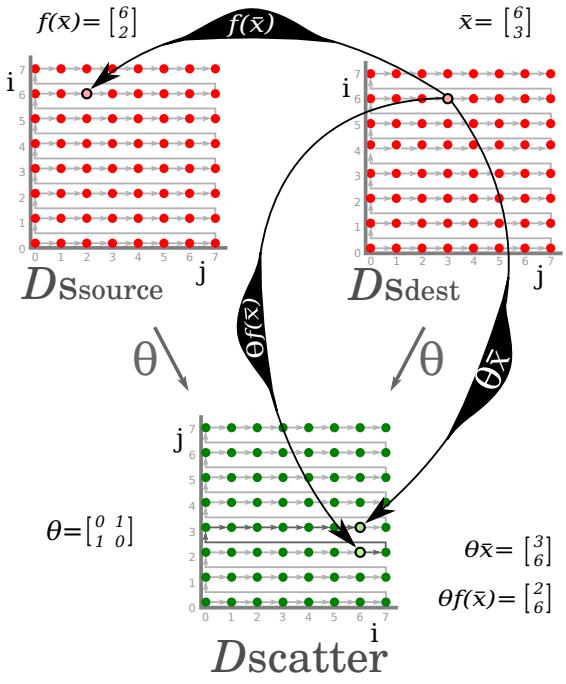


Figure 3.2: Illustration of transformation with regard to dependence

ILP formulation

Since parameters are not supported in the loop boundaries, it is possible to scan through the original iteration domain, and verify for each iteration point $\bar{x}$ that the lexicographical order in the scattering domain is still correct with regard to the dependence. However, specialized ILP solvers can solve this problem more efficiently by operating on polyhedra rather than single points. Some of these solvers are able to compute whether or not a system has a solution rather than finding the minimum or maximum solution. However, these solvers cannot implicitly determine lexicographical order of dimensions. This order can be made explicit by expressing the problem as a disjunction of subproblems that each compare the lexicographical precedence of a dimension. If we have points $\bar{w} = \begin{bmatrix} w_0 & w_1 & .. & w_k \end{bmatrix}^T$ and $\bar{z} = \begin{bmatrix} z_0 & z_1 & .. & z_k \end{bmatrix}^T$ in the scattering space, point $\bar{w}$ lexicographically precedes point $\bar{z}$ iff $(w_0 < z_0) \vee (w_0 = z_0 \wedge w_1 < z_1) \vee ... \vee (w_0 = z_0 \wedge w_1 = z_1 \wedge ... \wedge w_k < z_k)$. The conjunctions in this equation can be modeled in an ILP formulation, since they simply specify a set of equalities and inequalities. The disjunctions, however, require that the problem be subdivided in multiple formulations, since there is no way to model them in a single ILP formulation. Rather than showing that the dependence order is maintained for each pair of source and destination iteration points in the scattering domain, we formulate ILP problems that have solutions only if the dependence destination lexicographically precedes the source in the scattering domain.

An ILP formulation of the problem consists of a set of variables, a set of parameters, and a set of equalities and inequalities. We can subdivide the ILP formulation of these subproblems in three parts

1. The formulation of the variable $\bar{x}$ and $f(\bar{x})$

2. The formulation of the stripmining of $\bar{x}$ and $f(\bar{x})$

3. The formulation of the interchange and lexicographical precedence for a specific dimension

Note that parts 1 and 2 of the formulation remain the same for each subproblem, whereas part three changes according to the dimension that is to be verified.

In part 1 of this formulation, we implicitly represent the possible values of $\bar{x}$ as the set of variables and (in)equalities that represent the original domain of S_{dest}. The equalities from *constr* are then added to limit the possible values of $\bar{x}$ to the region of the domain of S_{dest} in which the conditions of the dependence are met. Then we introduce variables that represent the dimensions of $f(\bar{x})$, and establish the relationship between $\bar{x}$ and $f(\bar{x})$ through a set of equalities (one for each dimension in the domain of S_{dest}).

In part 2 of the formulation, we apply stripmining to every dimension of both $f(\bar{x})$ and $\bar{x}$, through the method explained in chapter 2. The variables representing the stripmined version of $\bar{x}$ are the dimensions of the scattering space, and the variables representing the stripmined version of $f(\bar{x})$ represent the same dimensions.

Finally in part 3 of the formulation, we add equalities and inequalities to relate the dimensions of the stripmined $f(\bar{x})$ and $\bar{x}$. Recall that the stripmined dimensions of the original domains represent the dimensions of the scattering space, and that loop interchange changes the lexicographical precedence of these dimensions. If the original destination and source domains have dimensionality k, the stripmined $f(\bar{x})$ and $\bar{x}$ both consists of $2k$ variables, which we will label $f(\bar{x})_0$ to $f(\bar{x})_{2k-1}$ and $\bar{x}_0$ to $\bar{x}_{2k-1}$. The interchange is specified by a sequence of the numbers 0 to $2k-1$, where each number refers to that dimension in both the stripmined source and destination domains. The interchange transformation is then represented in the ILP formulations by the order in which the stripmined dimensions are compared. If this sequence is $[\ d_0, d_1, d_{..}, d_{2k-1}\ ]$ where each d_x corresponds with one of the dimensions of the stripmined domains, 0 to $2k-1$, the ILP problem we define to find dependence violations at scattering dimension $2k-1$ is $f(\bar{x})_{d_0} = \bar{x}_{d_0} \wedge f(\bar{x})_{d_1} = \bar{x}_{d_1} \wedge .. \wedge f(\bar{x})_{d_{2k-1}} > \bar{x}_{d_{2k-1}}$. If the solution to this system is non-empty, there is an iteration point in the original domain of the dependence destination statement for which the transformation causes the dependence destination statement to be executed before the dependence source statement, which means that the transformation is not valid.

The algorithm used to verify the validity of a transformation w.r.t. a dependence is as follows:

Algorithm 1 verify_dependency_order

```
for dim = 1 → 2 * depth(D) do
  gt ← has_solutions(D, conditions, dim, θ, f, gt)
  if gt ≠ ∅ then
    return false
  end if
  eq ← has_solutions(D, conditions, dim, θ, f, eq)
  if eq = ∅ then
    return true
  end if
end for
return true
```

where $has\_solutions(domain, conditions, dim, transform, f, relation)$ generates and solves an ILP system that specifies that $x \in domain \cap conditions$, and for all dimensions in the scattering space that are lexicographically smaller than dim, $\theta\bar{x} = \theta f(\bar{x})$ and for dimension dim $\theta\bar{x} = \theta f(\bar{x})$ if $relation = eq$ or $\theta\bar{x} > \theta f(\bar{x})$ if $relation = gt$.

Example

To provide an example of this procedure, let us verify the correctness of a transformation on the loop nest of listing 2.4 with regard to the dependence $S1\ \delta^f\ S1$ which has the condition that $j > 0$ and the dependence vector is $[i, j\ -\ 1]^T$. The transformation θ we will verify is represented by the list $[0, 2, 3, 1]$, which means that both the dimensions i and j are stripmined, and that the strip of dimension i is interchanged so that it is the inner loop. We will use a stripmining size of 4, and we will assume that $N\ =\ 24$ and $M\ =\ 128$. To verify the validity, we will first define an ILP formulation ILP_{base} consisting of a set of variables, inequalities and equalities. This system will consist of part 1 and 2 of the formulation.

In part 1 of the formulation, we add the original iteration domain of the dependence destination ($\bar{x}$), consisting of the variables i and j, the constraints $i\ >=\ 0$, $i\ <\ 24$, $j\ >=\ 0$ and $j\ <\ 128$. To this, we add the extra constraint $j\ >\ 0$. Then we introduce the variables i' and j' to represent the original iteration domain of the dependence source statement ($f(\bar{x})$). The dependence function is then added in the form of the equalities $i' = i$ and $j' = j - 1$.

In part 2 of the formulation, we stripmine i, i', j and j'. To do so, we introduce the variables i_p, i_s, i'_p, i'_s, j_p, j_s, j'_p and j'_s to represent the control loops and the strip loops. For each variable x that is to be stripmined, we add the equations $4 * x_p \leq x$, $4 * x_p\ >=\ x - 3$, $x_s \geq 0$, $x_s \leq 3$ and $x = 4 * x_p + x_s$. The variables i_p,i_s,j_p and j_s together represent the scattering/scheduling space, whereas the variables i'_p,i'_s,j'_p and j'_s represent the same dimensions in the scattering space, as the scattering

space is the same for every statement.

To establish the lexicographical precedence, we need to specify part 3 of the formulation for each dimension, starting with the lexicographically strongest dimension.

In the transformation, the first and therefore lexicographically strongest entry is 0, which corresponds to the control loop of original dimension i, namely i_p and i'_p. To verify that there is no violation of the dependence at this dimension in the scattering space, we can create an ILP system consisting of ILP_{base} and the inequality $i'_p > i_p$, and if this system has a solution, there is a point in the transformed domain of the destination for which the corresponding source statement is scheduled in a later iteration than the destination statement, which means that the transformation is invalid. In this specific case, this system has no feasible solutions. If there is also no solution to the system ILP_{base} with the added inequality $i'_p = i_p$, then we know that the transformation is valid with regard to the dependence.

Since there is a solution, we continue to check for violations in the next dimension, namely 2, which corresponds with the variables j_p and j'_p. We then try to solve the system ILP_{base} with the added constraints that $i'_p = i_p$ and $j'_p > j_p$, which again has no feasible solutions. The system ILP_{base} with the added constraints that $i'_p = i_p$ and $j'_p = j_p$, does have solutions, which indicates that we need to proceed to the next dimension.

The next dimension is 3, and corresponds to the variables j_s and j'_s. We find that ILP_{base} with the added constraints $i'_p = i_p$ and $j'_p = j_p$ and $j'_s > j_s$ has no solutions, and neither does ILP_{base} with the added constraints $i'_p = i_p$ and $j'_p = j_p$ and $j'_s = j_s$. This means that the transformation is valid with regard to the dependence, as there is no instance of the dependence destination statement that is executed before the corresponding dependence source statement is executed.

3.6.3 Cost estimation

In this subsection, we introduce a transformation cost estimation algorithm, which is used to find the transformation of minimum estimated cost. The algorithm functions by determining per statement if it is simdizable dependency wise, if the memory strides are simdizable, if the sub expressions are simdizable, and whether or not misalignment is static. The high level cost estimation for a specific transformation is as follows

$$CE_{transformation} = C_{overhead} + \sum_{S_i \in Ss} \mathrm{CE}_{S_i}$$

where Ss is the set of all statements in the loop nest, not including the control, increment and test statements, and $C_{overhead}$ represents the cost overhead of the transformation compared to the original loop nest. This overhead consists of the estimated cost of the strip loops that are added to the loop nest. CE_{S_i} is the transformation specific cost

estimate for statement S_i, and defined as

$$CE_{S_i} = \begin{cases} CE_{S_{iSIMD}} & \text{if simdizable}(S_i) \\ CE_{S_{iscalar}} & \text{if } \neg\text{simdizable}(S_i) \end{cases}$$

where $CE_{S_{iSIMD}}$ and $CE_{S_{iscalar}}$ are the SIMD and scalar cost estimates for statement S_i. simdizable(S_i) represents whether or not S_i can be simdized after application of the transformation. The statement is considered simdizable if dependences, strides and the operation used in the subexpressions do not preclude simdization

$$\begin{aligned} \text{simdizable}(S_i) = \quad & \text{dependences\_simdizable}(S_i) \quad \wedge \\ & \text{strides\_simdizable}(S_i) \quad \wedge \\ & \text{subsexpressions\_simdizable}(S_i) \end{aligned}$$

Dependence simdizability

A set of operations can only be executed in a single SIMD operation if there are no interdependences within that set, unless a special vector instruction exists which can account for the dependence (e.g. a vector instruction that selects the minimum value element of the vector).

In order to determine if a statement can be simdized, we must determine if there are no dependences that block simdization of that statement. Simdization-blocking dependences can be either direct self dependences (e.g. a statement that depends on the previous iteration of itself) or indirect – a dependence on another statement that either directly or indirectly depends on the statement under examination.

Since we are trying to determine whether a statement as a whole is simdizable, there must be VS consecutive independent iterations of that statement in the inner loop, where VS is the number of elements in a SIMD vector. In determining whether or not dependences block simdization of a statement, we make the assumption that the statement must be parallelisable for the entire iteration sequence of the inner loop.

A very conservative approach to determine simdizability is to consider any dependence between different iterations of the same instance of the inner loop a simdization blocking dependence, and accepting that there may be false negatives. This can be done by using the same ILP base system was used in checking the validity of the transformation, and solving the base system with the constraint that all dimensions from the lexicographically strongest up to the lexicographically second weakest dimension of $\theta\bar{x}$ and $\theta f(\bar{x})$ are equal. If this has a solution, there is a dependence carried by the inner loop. Since the outcome of this analysis is the same for each statement in the loop nest, it need not be done for each statement individually. A false negative will occur if there is a dependence carried by the inner loop that is not part of a cycle, i.e. if there is a statement S_d that depends on another statement S_s, while S_s does not implicitly (through the transitive property of dependences) or explicitly (by direct dependence) depend on an instance S_d or an instance of itself in the same instance of the inner loop.

A less conservative approach is to create a graph of only the dependences between iterations in the inner loop, and applying an algorithm to find the strongly connected components. If the statement under examination is not in a strongly connected compo-

nent, then it is not part of a cycle in the graph, and there is no dependency prohibiting parallel execution of that statement.

Due to time constraints at the time of implementing the prototype, we have opted for the conservative approach that does allow for more false negatives.

Memory stride simdizability

As discussed in chapter 1, most SIMD ISEs have restrictions on memory strides for vector load and vector store operations. Commonly, only a stride of 1 is natively supported for vector load and store, and strides of 0 can be achieved with specific scalar expansion instructions. In order to determine if a statement can be simdized, each memory access in the statement is analysed to determine if the memory stride of the transformed iteration order is supported by the ISE.

In some cases it may be beneficial to use scalar instructions to pack data elements with unsupported strides into a data structure with supported stride, which can then be loaded into a vector register. Such operations are unsupported by the back-end, and hence this possibility has not been taken into account for memory stride simdizability.

In order to determine the strides of indexed memory accesses, we can exploit the fact that memory access functions are linear combinations of the loop variables and constants. Because of this property, the memory stride between any two consecutive iterations of the same instance of the inner loop is constant, since the only element that changes in the input of the memory indexing function is the loop variable of the inner loop. Thus, in order to find the stride of an access characterized with the function $f(\bar{x})$, all that needs be done is to calculate

$$f(\begin{bmatrix} c_0 & c_1 & c_2 & c_3 & .. & c_k \end{bmatrix}^T) - f(\begin{bmatrix} 0 & 0 & 0 & 0 & .. & 0 \end{bmatrix}^T)$$

where c_i is 1 if dimension i is the inner (lexicographically weakest) dimension in the transformed loop nest and the inner dimension is a strip loop, *stripsize* if i is the inner dimension, and it is a control loop and otherwise 0. This can be generalized for any dimension in the scattering domain:

$$stride_{dim}(f) = f(\bar{x}) - f(\bar{y})$$

where f is the indexing function, dim is the dimension (in the scattering space) for which we calculate the stride, and

$$x_i = \begin{cases} 1 & \text{if } i == (dim/2) \\ VS & \text{if } i == (dim/2) + 1 \\ 0 & \text{otherwise} \end{cases}$$

$$y_i = 0$$

where / represents integer division.

For example, if we have a statement $S1$ with iteration domain $D_{S1} = \{< i, j > | 0 \leq i < 8, 0 \leq j < 8\}$, a memory read of element $A[i * 8 + j + 16]$ and a transformation characterized as $[0, 2, 3, 1]$, the lexicographically weakest dimension in the scattering space is identified by the identifier 1,

which corresponds to the strip loop of loop variable i. The memory indexing function is $f(\begin{bmatrix} i & j \end{bmatrix}^T) = i * 8 + j + 16$, In this case, the inner loop stride is $f(\begin{bmatrix} 1 & 0 \end{bmatrix}^T) - f(\begin{bmatrix} 0 & 0 \end{bmatrix}^T) = (1*8+0+16) - (0*8+0+16) = 24 - 16 = 8$.

A statement is considered stride simdizable w.r.t. a transformation if all memory reads have a stride of 0 or 1, and all memory writes have a stride of 1, i.e. if both of the following conditions are met:

$$\forall r \in S_i \mid (stride_{innerdim}(r) = 0) \vee (stride_{innerdim}(r) = 1)$$

$$\forall w \in S_i \mid stride_{innerdim}(w) = 1$$

where r is the indexing function of an indexed read in the statement under consideration, w is the indexing function of an indexed write (of which there can be at most one per statement, due to the CCMIR specification), *innerdim* represents the lexicographically weakest dimension in the scattering domain and stride represents the memory index increment (in units of the size of the data type of the accessed element) between two consecutive iterations of a dimension (in this case, *innerdim*) in the scattering domain.

Misalignment analysis

While we were unable to generally determine if memory accesses in the inner loop were aligned or misaligned based on the polyhedral representation, we can relatively easily determine if the misalignment varies with the values of the outer loop variables. For example, an array access like $A[i + j]$ cannot be statically aligned if we have a 2-dimensional iteration domain with unit strides, since the alignment in the inner loop varies with the iteration in the outer loop. Since misalignment may make the difference between a speedup and a performance degradation, this information should be taken into account when doing cost estimation.

To determine this, we analyse per memory access function whether the strides of the outer loops as determined with the method described in the preceding subsection are dividable by the vector size without remainder. If they are not, the memory access is considered to be dynamically misaligned. The algorithm to determine if a memory access is dynamically aligned is

$$\text{dynamically\_misaligned}(f) = \begin{cases} false & \text{if } \forall dim_i \in D_\theta \setminus innerdim \mid stride_{dim_i}(f)\%VS = 0 \\ true & \text{otherwise} \end{cases}$$

where f is the indexing function of the memory access, $D_\theta \setminus innerdim$ is the set of dimensions of the scattering domain excluding the inner dimension, and VS is the number of elements in a vector.

Statement cost estimate

In order to issue a per-statement cost estimate, each statement in the loop nest is analysed expression by expression. Recall that the code generator uses rules that match patterns in the CCMIR to the LIR. Through code generator feedback, the analysis will determine per expression if the operation is simdizable, i.e. that a SIMD match rule exists for that node

in the CCMIR representation. Furthermore, it checks if the element size is compatible with the element sizes of the other subexpressions, e.g. we cannot combine operations on 8 bit integers with operations on 32 bit integers in the same statement unless we make specific provisions in the code generator for such constructs (which we have not done). If the expression is simdizable, a (possibly) operation specific constant $C_{op_{SIMD}}$ is added to the simdized statement estimate, while $C_{op_{scalar}}$ is added to the scalar statement estimate. These constants represent the estimated cost of executing the operation as SIMD instructions or scalar instructions respectively. This cost does not represent any actual unit, and is solely intended for comparing different transformations. For supported operations, we might expect that the cost of a SIMD instruction is approximately $\frac{1}{VS} * C_{op_{scalar}}$, where VS is the number of elements in the vector, which depends on the size of the data type. However, certain operations may require multiple SIMD instructions, such as unaligned loads, stores and scalar expansions.

If any of the subexpressions is found not to be simdizable, the scalar statement estimate is used as the statement estimate, otherwise the simdized statement estimate is issued. In case of memory load and store operations, the estimate also depends on stride and misalignment analysis.

The statement SIMD and scalar cost estimates are calculated as follows

$$CE_{S_{i_{SIMD}}} = \sum_{\exp_i \in S_i} C_{op_{SIMD}}$$

$$CE_{S_{i_{scalar}}} = \sum_{\exp_i \in S_i} C_{op_{scalar}}$$

where each exp_i represents a node in the expression tree of S_i, not including expressions that are part of an indexing function.

Transformation cost estimate

The total cost estimate of the transformation is the sum of the cost estimates of the statements, to which a stripmining overhead estimate $C_{overhead}$ is added for each loop that is stripmined in the final transformation. Note that as long as the cost estimation lacks any cache locality metric, this constant will always lead to selection of a transformation in which zero or one dimensions are stripmined.

Now let us apply this cost estimation for two different transformations on the example of listing 2.4, with $N = 16, M = 16$. We will take as our target architecture the AltiVec ISE. The transformations for which we will issue cost estimates are $\theta_1 = [0, 2, 3, 1]$ and $\theta_2 = [0, 2, 1, 3]$, where θ_1 is an example of stripmining one dimension and interchanging it towards the deepest level of the loop, and θ_2 is an example of stripmining both dimensions, and interchanging the strips toward the deepest level. Note that $2, 3$ of θ_1 will be considered as a single dimension, since the identifier of the control loop and strip loop appear as consecutive elements in the transformation, which means that there is no reason to perform the stripmining of the original dimension. We will use a stripmining size of 4 in this example. Since

cost estimation is only performed for valid transformations, validity of the transformations has already been established. The loop nest contains the single statement $S_i = \{x[i] = x[i]+y[j];\}$ The statement contains 3 indexed memory accesses: a read from element $x[i]$, a read from element $y[j]$ and a write to element $x[i]$.

We will start with the cost estimation for θ_1. A dependence analysis shows that if we apply transformation θ_1, there are no dependences between two iterations of the same instance of the inner loop, so the inner loop can be considered simdizable with regard to dependences, therefore dependences_simdizable(S_i) evaluates to *true*. Stride analysis shows that after the transformation, the write to $x[i]$ has a stride of 1, the read from $x[i]$ has a stride of 1, and the read from $y[j]$ has a stride of 0, each of these strides is considered simdizable, therefore strides_simdizable(S_i) evaluates to *true*.

Since the read from $y[j]$ has a stride not equal to 0 or the vector size (which in this case is 4) in loop dimensions other than the inner loop, this memory access is considered dynamically misaligned; alignment varies per instance of the inner loop.

The statement consists of the sub-operations $OP_0 = \{write : x[i] : int\}$, $OP_1 = \{int + int\}$, $OP_2 = \{read : x[i] : int\}$ and $OP_3 = \{read : y[i] : int\}$. Each of these sub-operations is supported by the SIMD ISE, and each of the sub-operations has the same data type and width. Let us take the following operation cost estimates: $cost(write : int) = 150$, $cost(read_{stride_0,misaligned} : int) = 400$, $cost(read_{stride_1} : int) = 100$, and $cost(int + int) = 100$, where we take into account that the access of OP_3 requires scalar expansion from a load that is know to be misaligned. This gives us the statement cost estimate $CE_{S_i} = 150+400+100+100 = 750$. The transformation θ_1 introduces one extra dimension, so we will add $1 * C_{overhead} = 150$ to this cost estimate. This gives us the estimate: $CE_{\theta_1} = 900$

Now let us consider the second transformation, θ_2. Dependency analysis instantly show us that there is a dependence from an iteration of the inner loop to the previous iteration; this blocks simdization, and therefore dependences_simdizable(S_i) evaluates as *false*. Since the statement is not simdizable, the scalar estimate for the statement will be issued. Let us take the following operation scalar cost estimates: $cost(write : int) = 400$, $cost(read : int) = 400$, $cost(read_{stride_1} : int) = 400$, $cost(int + int) = 400$. This gives us the statement cost estimate $CE_{S_i} = 400 + 400 + 400 + 400 = 1600$. Since two dimensions are stripmined, we add $2 * C_{overhead} = 150$, giving us the estimate: $CE_{\theta_2} = 1900$.

Since $CE_{\theta_2} > CE_{\theta_1}$, the cost estimation predicts that θ_1 is a better transformation than θ_2.

3.7 Transformation and simdization

After the cost estimation of the valid transformations, the transformation that has the lowest cost estimate is applied, after which the pre-existing simdization flow can proceed.

3.7.1 Transformation

This transformation is then translated into a scattering function for each statement, and CLooG is invoked to generate the code that scans the scattering domain according to the lexicographical ordering specified by the transformations. This scanning code is in the form of a CLooG abstract syntax tree (clast) data structure, which is then translated into CCMIR, which is the middle end IR of the CoSy compiler development system. As part of the clast, functions to compute the iteration point in the original domain from an iteration point in the scattering domain are also generated, which are used to transform any memory indexing functions. The original statements are copied into the new loop nest, array indexing functions are rewritten with the transformed indexing functions, and finally, the original loop nest is replaced by the new loop nest.

3.7.2 Loop versioning

Because of the stripmining transformation, there is a large probability that the inner loop of the rewritten loop nest no longer has constant loop boundaries that can be statically determined at compile time. A known iteration count for the inner loop is a requirement for successful simdization in the remainder of the simdization flow. For this reason loop versioning is applied to gain one version of the innermost loop which runs for a constant number of iterations that is known at compile time. To achieve this, the loop versioning engine searches for inner loops that are marked to be versioned, if the loop initialisation expression or loop test expression contain a loop invariant select expression with one or more constant leaves, it will generate an if-then-else construct which checks for the conditions to get both a constant loop init value and a constant loop test value. Then, the loop is copied to the then branch of the if statement, and the original is moved to the else branch. The init and test values of the copy are then updated to the constant values. As loop dependency analysis becomes too conservative for simdization if applied after loop versioning, the loop dependency analysis of the rewritten loop nest is run before the versioning, and the loop dependency graph is then updated along with the loop code as part of the loop versioning.

For example, consider a loop nest which has been stripmined and interchanged. In the transformed code, the innermost dimension is represented by the variable $c3$, and it is a strip loop with the parent loop being represented by variable $c1$. Each loop level has a lower bound and a upper bound. For this example, we shall take a lower bound of

$$\mathtt{LB}_{c3} = \begin{cases} 0 & \mathtt{if}\ 0 > -4 * c1 \\ -4 * c1 & \mathtt{otherwise} \end{cases}$$

and an upper bound of

$$\mathtt{UB}_{c3} = \begin{cases} 3 & \text{if } 3 < 1023 + -4 * c1 \\ 1023 + -4 * c1 & \text{otherwise} \end{cases}$$

where we know that $c1$ is invariant in the inner loop, which means that the number of the iterations of the loop depends only on a variable that does not change within the loop. Both the lower bound and the upper bound have a conditional constant value and a non-constant value. Since $c1$ is known not to change in the inner loop, we can lift these conditions out of the loop, and create a version with lower bound 0, upper bound 3, that is entered only if $0 > -4 * c1 \ \wedge \ 3 < 1023 + -4 * c1$, and leave the old version of the loop in the case that the condition is not met. Note that in all real tests we have performed, the lower bound condition is in fact unnecessary if $c1$ is known to stay positive, however due to time constraints, we have opted not solve this through range based analysis. This will yield a loop nest as shown in listing 3.9.

Then, we must update the dependences to account for the changes we have made by versioning the loop. In this case, we are dealing with direction vectors rather than precise dependences, which means that a dependence is be characterized by a vector of directions, e.g. $[=, <, =]$ or $[=, =, <]$. For each dependence featuring one statement in the inner loop and one statement outside of the inner loop, we add a copy of the dependence with new version. For each dependence of which both sides are inside the loop, we copy and adjust the dependence if it is carried only by the inner loop, and if it is also carried by one of the outer loops we add the necessary dependences between the versions. We assume that we have no specific knowledge of the order in which versions may be executed, which means that we add a dependence from the new version to the new version, one from the new version to the original version and one from the original version to the new version.

Listing 3.9: Versioned inner loop

```
...
if (0 > -4 * c1 && 3 < 1023 + -4 * c1)
{
  for (c3=0; c3 <= 3; c3++)
  {
    S0; // version 2
    S1; // version 2
  }
} else {
  for ( c3 = (0 > -4 * c1 ? 0 : -4 * c1);
        c3 <= (3 < 1023 + -4 * c1 ? 3 : 1023 + -4 * c1);
        c3++ )
  {
    S0; // version 1
    S1; // version 1
  }
```

```
}
...
```

3.8 Conclusion

In this chapter, we have explained our proposed changes to the CoSy simdization flow both in high level and in detail. We have described a transformation space consisting of combinations of stripmining and loop interchange, an algorithm to verify that a transformation does not violate any data dependences, and an algorithm by which we can provide a cost estimate per transformation. To estimate the cost of different transformations, we perform an analysis of dependences, memory strides and subexpressions in order to determine per statement if it can be simdized and to estimate the cost of simdizing the statement. Furthermore, we have described how we apply loop versioning to gain a version of the inner loop that has constant loop bounds, allowing for simdization by the CoSy simdization flow.

4 Evaluation

4.1 Introduction

In chapter 3, we have presented a new simdization flow for the CoSy framework. In this chapter we will evaluate this new flow and its parts through a series of measurements. In section 4.2, we describe the test programs and the experimental setup with which the measurements were performed. In section 4.3, the cost model is evaluated by relating the transformation cost estimates to the actual performance for each transformation for a small number of tests. In section 4.4, measurements of the old flow, the new flow and its parts are presented, analysed and evaluated.

4.2 Experimental setup

For our evaluation we have selected two target systems, viz. AltiVec and SSE2, because they are widely available and they each have distinct characteristics.

The PowerPC/AltiVec execution time measurements were performed on a system with a PPC970FX processor and 4GB of internal memory. The execution time results have been measured using the perf tool, and represent the execution time of the entire binary. perf is a performance measurement and profiling tool that is part of the performance counters subsystem in Linux. Apart from application execution time, it can count and measure a host of other things, including cycles, branches, branch misses, cache references, cache misses and page faults. The X86/SSE2 execution time tests were performed on a system with an *AMD Athlon(tm) 64 X2 5600+* processor and 4GB of internal memory. As the perf tool was not available on the system, the timing measurements were done by invoking the standard POSIX function *gettimeofday* before and after invoking the optimized test function. Since there is no need to compare the AltiVec execution times to the SSE2 execution times, this difference in measurement method is not an issue for our evaluation purposes.

To evaluate the transformation exploration strategy, the cost estimation model and the actual simdization flow, we have selected a small set of perfectly nested computational kernels. This set consists of one-dimensional convolution, two-dimensional convolution, a pre-fissioned Sobel filter – which is an application of two-dimensional convolution – and a matrix multiplication. Convolution was selected because it is a very common kernel in the DSP domain, both in the one-dimensional and two-dimensional form. Matrix multiplication was selected because it is a common operation in scientific computing and it also has some applications in image processing. These specific tests were selected because when implemented in a straightforward way, none of the tests could be simdized using the old flow. In the case of the matrix multiplication and both convolution tests,

only the deepest loop nest was included in the test, whereas in the Sobel filter, the entire implementation was pre-fissioned into perfectly nested loops.

The C code for each of these tests is included in Appendix A.

Each of the tests was implemented using floating point arithmetic for the reason that the floating point functionality of both the AltiVec code generator and the SSE2 code generators is more complete than the integer arithmetic.

In addition to these tests, a small subset of tests from the PolyBench/C [22] version 3.2 was used. PolyBench/C is a suite of benchmarks specifically created for benchmarking of polyhedral tools; each benchmark contains a computational kernel with static control flow (SCoP). The following algorithms were selected[1]:

3mm performs 3 matrix multiplications

2mm performs 2 matrix multiplications

bicg BiCG Sub Kernel of BiCGStab Linear Solver

doitgen Multiresolution analysis kernel

fdtd-2d 2-D Finite Different Time Domain Kernel

floyd-warshall finds the weight of the shortest path in a weighted graph

mvt Matrix Vector Product and Transpose

syrk Symmetric rank-k operations

Because of an unresolved problem in our implementation of the interface to CLooG, transformations on non-rectangular iteration domains cause a critical failure. For this reason, only kernels with a rectangular iteration domain were selected. Note that the kernels were fissioned to obtain perfectly nested loops, the loop boundaries were changed from function parameters to static numbers. Furthermore, data objects are not passed to the kernel as parameters, but are allocated as globals, whereas in the original benchmarks they are allocated on the heap. This was done because the object alignment analysis and aliasing analysis are out of the scope of this project. For these kernels, the execution time was measured using the tooling of the benchmark suite.

In each case, the program was compiled to an assembly file using an architecture specific CoSy compiler, and assembled and then linked on the target system into an executable together with a main function that invokes the test. All global data objects are assumed by the compiler to be 128-bit aligned.

4.3 Cost model validation

A critical part of the proposed simdization flow is the selection of the best transformation from the transformation exploration space. The cost estimates for the individual points in this exploration space are relied upon to indicate which transformation will yield the

[1] Most benchmark descriptions are from PolyBench website[22]

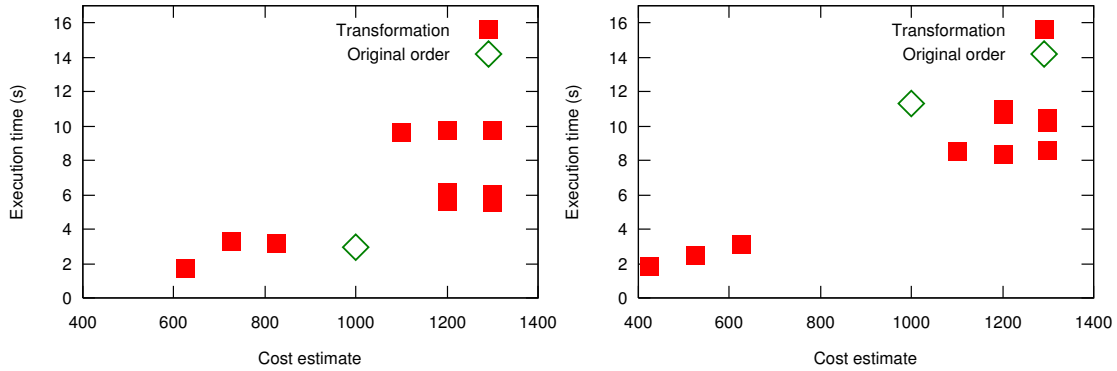
best execution time. To determine the quality of the cost estimation as a heuristic for selecting a transformation to perform, we have compared the cost estimation number per transformation with the actual execution time on the AltiVec target system. Figures 4.1a and 4.1b show the cost estimates and actual execution times of various valid transformations of two-dimensional convolution and matrix multiplication respectively. Note that in both these cases, the transformation with the lowest cost estimate is also the transformation with the shortest execution time. Ideally, there should be a strong correlation between the estimated cost and the actual execution time.

Note that much of the discrepancy between differences in the estimations and the actual execution times may be explained by the fact that cache locality is not explicitly taken into account in the cost estimation; it is only included insofar as it coincides with simdizable access patterns. This means that the cost estimation is unable to differentiate between a transformation that allows for efficient use of caching and a transformation that does not. Another cause for discrepancies between cost estimates and execution time is the fact that register re-use is not taken into account, while the full support for vector register reuse is still incomplete in the current simdization flow. It may happen that a vector register is written to memory in one iteration of a loop, and that same value is read into a vector register in the next iteration. While this is in accordance with the cost estimation for simdizable statements, the scalar compiler flow does exploit register reuse, which is not represented in the scalar cost estimates. Vector value re-use is a feature which is to be implemented in the near future, however it's applicability may be limited because of the loop versioning. This is because due to the loop versioning, it is unknown whether the previous and next iterations of the innermost loop are the simdized version or the scalar version of the loop.

In both the matrix multiplication and 2-dimensional convolution, the depth of the original loop is greater than or equal to 3, which means that in both these cases, the transformation space that is explored is limited to stripmining up to three loops, and moving those strips to the inside of the loop nest in every possible order. As might have been expected, the cost model will always issue a lower cost estimate for any simdizable transformation that stripmines one dimension than an equally simdizable transformation which stripmines more than one dimension. This is to be expected because cache and register re-use are not taken into account. In both of the tests, the selected transformation is a case of outer loop simdization, i.e. stripmining one dimension and interchanging it in order to obtain a simdizable inner loop. Had the option of interchange without stripmining been available, the lack of cache locality metrics would have led to that transformation being selected over one consisting of stripmine and interchange, which might have resulted in a degradation.

Figure 4.2 shows an example of the cost estimation causing the selection of a suboptimal transformation. In the case of the one-dimensional convolution, the larger exploration space is selected because of the limited loop depth. This exploration space does not only feature loop blocking, but the full set of permutations of the loops and the strips mined from the loops, and hence also the case of a full interchange. Since the full interchange has less loop overhead than stripmining, and the cost model does not include any cache metric, full interchange is chosen over outer-loop simdization (or stripmine and interchange).

Although the cost estimation heuristic correctly picks the best transformation in some cases, there is certainly room for improvement, especially in the areas of cache optimization and some form of estimation of iteration counts. Note that a cache locality metric like reuse distance could be added to the current cost model.



(a) 2D convolution (b) Matrix multiplication

Figure 4.1: Cost estimation vs Execution time

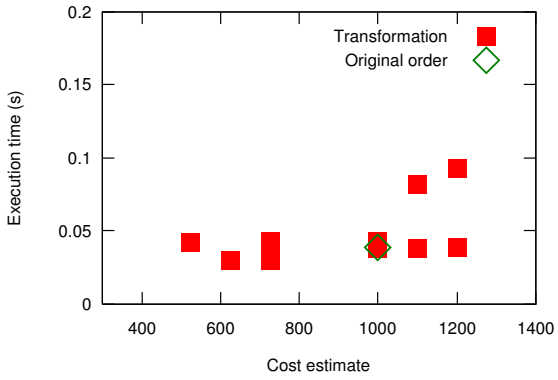
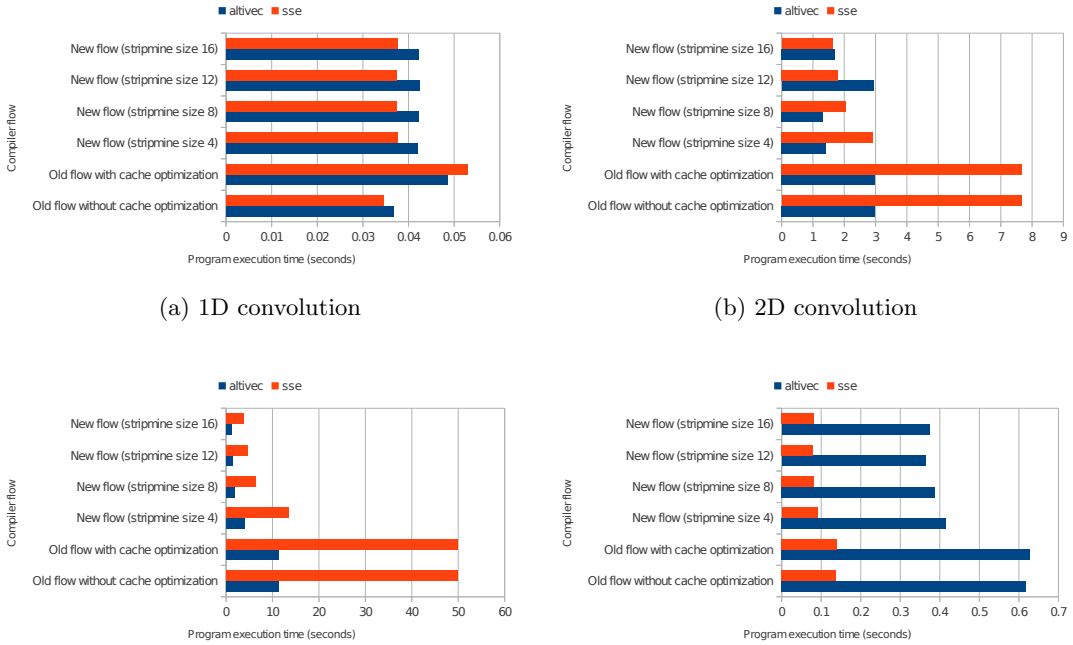


Figure 4.2: Cost estimation vs Execution time for 1D convolution

4.4 Performance measurements

In the previous section, we have shown that the cost model can to some extent predict which transformation will lead to the best performance. In this section we will focus on the performance in comparison with the old transformation flow, using execution time measurements on both the SSE2 and AltiVec systems. Leaving aside cache effects, ideally we would expect a simdization speedup equal to the number of operations that can be done in parallel; 4 in the case of floating point operations and 128 bit vector size. In practice, we find that such a speedup is often not attainable because of data alignment, scalar expansions and overhead introduced by the loop transformations that are required in order to simdize.

Figure 4.3 shows the execution times of the tests, compiled with both the old flow and the new flow. Note that the old flow was unable to issue SIMD instructions for any of the tests. These measurements show that large speedups can be attained over the



(a) 1D convolution

(b) 2D convolution

(c) Matrix multiplication

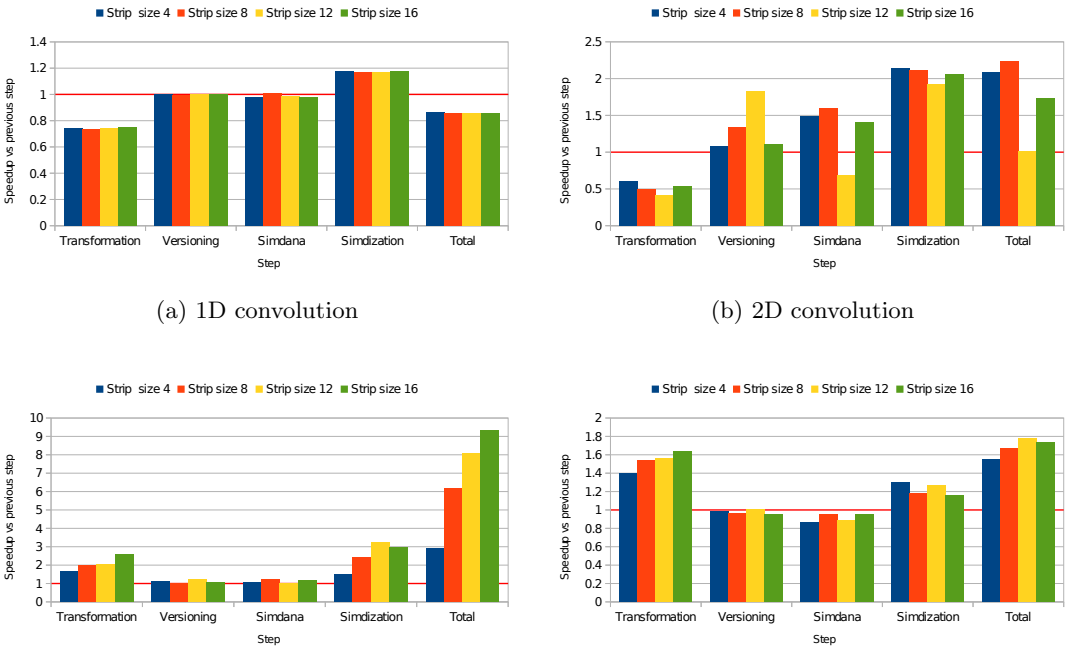
(d) Sobel

Figure 4.3: Execution times of the compiled kernels

old flow, although in the case of the one-dimensional convolution test, performance is slightly degraded. This degradation can be partially attributed to the fact a suboptimal transformation is selected because the cost model does not include any cache locality metric. For the matrix multiplication test however, an improvement in cache locality results in a speedup greater than 4.

Figure 4.4 shows the speedups for the AltiVec target system associated with each step in the simdization flow. The speedup of each step was measured by running the simdization flow up to and including that step, skipping the remainder of the simdization flow, and comparing the run-time to the same flow without the step. For example, the speedup of *Versioning* is calculated by dividing the execution time of a simdization flow consisting of only *Transformation* by the execution time of a simdization flow consisting of *Transformation* and *Versioning*. The first step, labeled as *Transformation* is the exploration of different loop transformations and the selection and application of the transformation with the lowest cost estimate. The second step, labeled as *Versioning*, creates a version of the inner loop with a constant number of iterations. The third step, labeled as *Simdana*, takes care of unrolling, loop peeling and scalar expansion. The final step, labeled as *Simdization*, is the basic block simdization. The total speedup is the product of the speedup of each step. In these figures, we can see that in most cases a

larger strip size yields better performance. This can be attributed to the fact that the innermost strip will be versioned, and the versioning overhead will be lower if the loop runs for more iterations. On the other hand, by making the strip size larger, we run the risk of damaging cache locality, and the chance of entering the non-constant version of the loop will be increased as well. Ideally, the strip size should be determined through analysis.



(a) 1D convolution

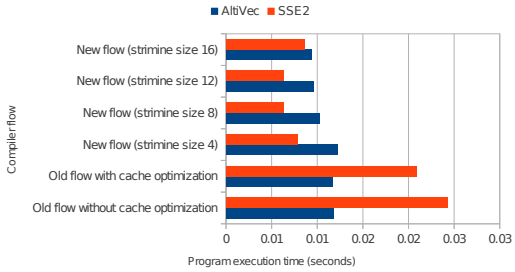
(b) 2D convolution

(c) Matrix multiplication

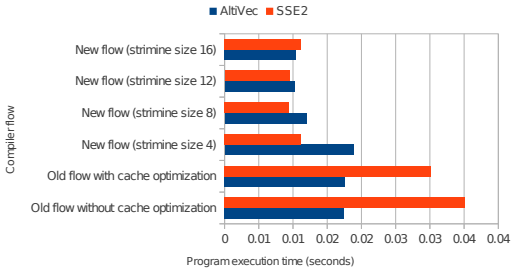
(d) Sobel

Figure 4.4: Execution speedups per pass in the simdization flow

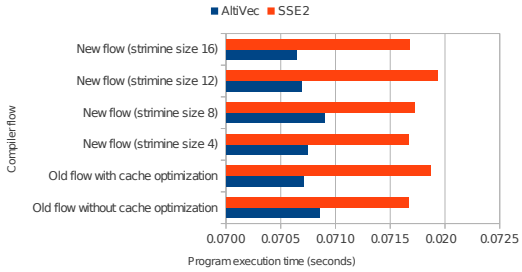
Figure 4.5 shows the execution times of selected benchmarks from the PolyBench suite. Figure 4.5a shows the execution time measurements of the 2mm benchmarks. Note that there are marginal speedups ranging from 1.14 to 1.25 for the AltiVec measurements with strip sizes 8, 12, 16, and a speedup of 0.96 for strip size 4. The SSE2 measurements show speedups ranging from 2.4 to 3.32. Figure 4.5b shows similar results for the 3mm benchmark, with speedups ranging from 0.93 to 1.7 for the AltiVec, and speedups ranging from 2.69 to 3.18 for the SSE2 measurements. Figure 4.5c shows the measurements for the bicg benchmark. Note that the new flow results in the exact same transformation as the old flow in this example, and the slight differences in execution time are in fact measurement noise. The doitgen benchmark results are shown in figure 4.5d, and are similar to the results of the 2mm benchmarks. In the AltiVec measurements, again strip size 4 results in a degradation with a speedup of 0.856, where the larger strip sizes result in speedups ranging from 1.44 to 1.69. In the SSE2 measurements, speedups range from 2.7 to 3.22. Figures 4.5e and 4.5f show the results for the fdtd-2d and floyd-warshall benchmarks respectively. In both cases, the end result of the old flow and



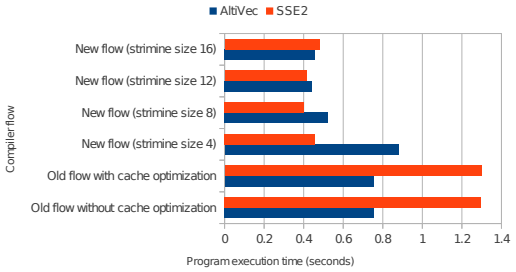
(a) 2mm



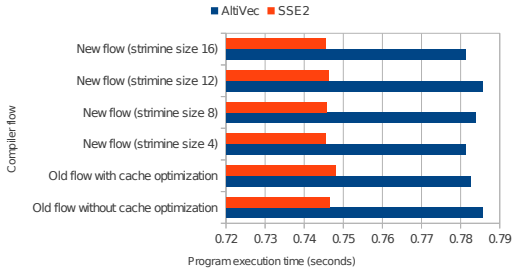
(b) 3mm



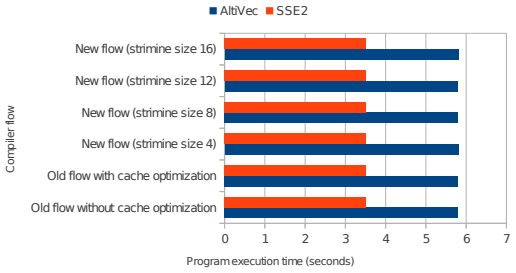
(c) bicg



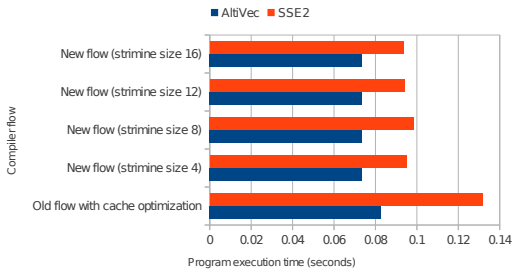
(d) doitgen



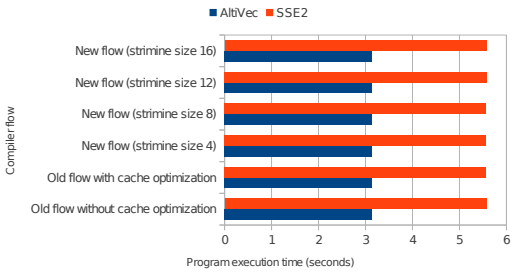
(e) fdtd-2d



(f) floyd-warshall



(g) mvt



(h) syrk

Figure 4.5: Execution times of the compiled kernels

the new flow is the same. Figure 4.5g shows the results for the mvt benchmark. Note that the measurement for old flow without cache optimization was omitted because for the AltiVec measurements, both the new flow and the old flow with cache optimization approximately achieve a speedup of 50 over the old flow without cache optimization. The new flow achieves a speedup ranging from 1.123 to 1.124 for AltiVec, and a speedup ranging from 1.33 to 1.4 on SSE2. Finally, the results of the syrk benchmark are shown in 4.5h. The new flow results in the same code as the old flow.

These benchmarks have served to show that the strip size must be included in the cost model, as it can make the difference between a speedup and a slowdown.

kernel	Strip size 4	Strip size 8	Strip size 12	Strip size 16
conv1d	1.15	1.15	1.14	1.15
conv2d	2.09	2.24	1.01	1.74
mm	2.92	6.18	8.11	9.35
sobel	1.51	1.62	1.73	1.68
2mm	0.96	1.15	1.22	1.25
3mm	0.93	1.47	1.72	1.69
bicg	1.00	1.00	1.00	1.00
doitgen	0.86	1.44	1.70	1.65
fdtd-2d	1.00	1.00	1.00	1.00
floyd-warshall	1.00	1.00	1.00	1.00
mvt	1.12	1.12	1.12	1.12
syrk	1.00	1.00	1.00	1.00
average	1.32	1.76	1.89	2.06

Table 4.1: AltiVec speedups per benchmark compared to old flow with cache optimization

kernel	Strip size 4	Strip size 8	Strip size 12	Strip size 16
conv1d	1.41	1.42	1.41	1.41
conv2d	2.62	3.73	4.29	4.73
mm	3.71	7.75	10.89	13.06
sobel	1.54	1.70	1.75	1.70
2mm	2.66	3.32	3.30	2.41
3mm	2.71	3.18	3.18	2.69
bicg	1.00	1.00	1.00	1.00
doitgen	2.84	3.23	3.13	2.71
fdtd-2d	1.00	1.00	1.00	1.00
Floyd-warshall	1.00	1.00	1.00	1.00
mvt	1.39	1.33	1.40	1.41
syrk	1.00	1.00	1.00	1.00
average	1.99	2.61	2.94	3.01

Table 4.2: SSE2 speedups per benchmark compared to old flow with cache optimization

Tables 4.1 and 4.2 shows the speedups compared to the old simdization flow with cache optimization for AltiVec and SSE2 respectively.

4.5 Conclusion

In this chapter we have presented an experimental evaluation of our proposed changes to the simdization flow. We have shown that the cost estimation allows for selecting the best transformation in most cases, and that the new flow results in a speedup for a large number of the tests we have run, and most of the cases where it does not, it does not result in a degradation. The tests have exposed a number of flaws in the cost model, the most important of which is that cache locality is not accounted for in the cost model. While in many cases, improvement in simdizability coincides with favorable cache locality, this is not always the case. Furthermore, the test results have shown that the strip size can make the difference between speedup and degradation, while it is not accounted for in the cost estimation.

On the AltiVec system, the speedups measured over the old flow range from 0.85 in the worst case, to 9.3 in the very best case. On the SSE2 system, the speedups ranged from 0.92 in the worst case to 13.06 in the best case.

5 Conclusion and future work

An analysis of the simdization flow in CoSy has determined that certain loop nests require additional transformations before they are simdizable. The exact nature of these transformations depends on properties of the loop nest and the statements contained within it. To address this, we have proposed a prototype for a simdization flow in the CoSy compiler which exposes SIMD parallelism through application of such transformations. This simdization flow consists of exploring a transformation solution space, heuristically estimating the cost of each transformation, applying the best transformation, and finally versioning the loop to obtain a simdizable inner loop.

To achieve this, a model and representation of stripmining and loop interchange have been constructed, as well as a heuristic per-transformation cost estimation.

Through tests with both the SSE2 ISE and the AltiVec ISE, this new flow is shown to be beneficial for some of the tests, with in the best case a speedup of 9.3 compared to the old simdization flow. For a few tests, a performance degradation was measured, with a worst case speedup of 0.85, showing that certain features that influence program performance are not accounted for in the cost estimation. The cost estimation was able to select a transformation that resulted in simdization of the statements. Due to restrictions on the supported input, the number of tests is rather limited. A full validation of the cost model should also make use of the loop fission and scalar expansion transformations to allow a broader range of input programs. The results have also shown that the polyhedral model may be an invaluable tool in achieving improved cache optimization, as significant speedups attributable to improvement of cache were measured.

Future work

While the prototype only supports perfectly nested loops, there is no such limitation in the polyhedral analysis or in the capability of the polyhedral tools. Support for loop fission as a transformation that enables interchange in imperfectly nested loops is certainly feasible, and would greatly increase the number of applications that can be optimized. Similarly, scalar expansion could also be included as an enabling transformation.

Certain properties of certain binary operations, such as the addition operation on unsigned integers, allow the order of the operations to be altered when they are applied in a reductive manner. For example, the statement $x[i] += y[j]$; takes two operands, $x[i]$ and $y[j]$, performs the associative and commutative operation + and overwrites $x[i]$. If the properties of associativity and commutativity hold for the type of the operation and the specific type of the operands, the order in which elements from $y[j]$ are added to $x[i]$ is unimportant for the end result. Such properties are also exploited in parallel computing and computer arithmetic, e.g. in the prefix sum algorithm. By recognizing these properties, potentially beneficial transformations might become valid where otherwise

they were not.

Since we already decide on a statement level whether or not a particular statement is simdizable, we might consider going one step further, and performing the actual simdization at the polyhedral level. An advantage of doing simdization at statement level is that it is possible to estimate the cost of partially simdizing the statement in cases where one ore more loads have unsupported strides, or where unsupported subexpressions are encountered, and decide whether or not to partially simdize based on this estimate.

Another shortcoming of the current prototype is its lack of awareness of iteration counts in the transformed polyhedral domain. While the used representation of a transformation is convenient to extract information about the order in which loop dimensions are iterated, there is no information on the number of iterations that is executed in each loop dimension. While acquisition of such information is complicated for the general case (the number of iterations in a dimension may be dependent on the iteration in another dimension), there are cases where such information is easily determined. If, for example, the original iteration domain is of (hyper)rectangular shape, the number of iterations in the original domain is easy to determine, and should be easy enough to track through stripmining and loop interchange. Such rectangular shapes are in fact very common, and all the tests in chapter 4 feature rectangular iteration domains.

The number of iterations per dimension is especially important if non-perfectly nested loops are to be considered, because in that case not every statement is necessarily executed the same number of times, which should be accounted for in the cost estimation.

In the current implementation, stripmining uses a fixed strip size, which can be overridden by passing an option to the compiler. Furthermore, every stripmined dimension is stripmined with the same size. This stripmining size should be automatically determined per dimension, perhaps based on properties of the cache and on the vectorization requirements, e.g. it should be at least as large as the number of data elements in a vector. Having information on the number of iterations per dimension could also be useful in determining the best stripmining size.

Finally, the evaluation has shown that there is a definite need for a cache metric to be added to the cost estimation, since cache locality can make the difference between a speedup and a degradation. Similarly, register value re-use should be accounted for.

Bibliography

[1] Randy Allen and Ken Kennedy, *Automatic translation of FORTRAN programs to vector form*, ACM Transactions on Programming Languages and Systems **9** (1987), 491–542.

[2] R. Bagnara, P. M. Hill, and E. Zaffanella, *The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems*, Science of Computer Programming **72** (2008), no. 1–2, 3–21.

[3] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar, *Efficient selection of vector instructions using dynamic programming*, Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (Washington, DC, USA), MICRO '43, IEEE Computer Society, 2010, pp. 201–212.

[4] Alexander Barvinok and James E. Pommersheim, *An algorithmic theory of lattice points in polyhedra*, 1999.

[5] Cédric Bastoul, *Code generation in the polyhedral model is easier than you think*, Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (Washington, DC, USA), PACT '04, IEEE Computer Society, 2004, pp. 7–16.

[6] Uday Bondhugula, Albert Hartono, and J. Ramanujam, *A practical automatic polyhedral parallelizer and locality optimizer*, In PLDI '08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, 2008.

[7] Philippe Clauss, Vincent Loechner, V. Taylor, and K. Parhi, *Parametric analysis of polyhedral iteration spaces*, Journal of VLSI Signal Processing **19** (1998), 179–194.

[8] Albert Cohen, Sylvain Girbal, and Olivier Temam, *A polyhedral approach to ease the composition of program transformations*, in: Euro-Par'04, no. 3149 in LNCS, Springer-Verlag, 2004, pp. 292–303.

[9] Sander de Smalen, *A solution to misaligned data access in a vectorizing compiler framework*, Master's thesis, Technische Universiteit Delft, 2011.

[10] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien, *Vectorization for simd architectures with alignment constraints*, Proceedings of the ACM SIGPLAN 2004 Conference on Programming language design and implementation (New York, NY, USA), PLDI '04, ACM, 2004, pp. 82–93.

[11] Paul Feautrier, *Parametric integer programming*, RAIRO Recherche Op'erationnelle **22** (1988).

[12] ———, *Dataflow analysis of array and scalar references*, International Journal of Parallel Programming **20** (1991).

[13] Liza Fireman, Erez Petrank, and Ayal Zaks, *New algorithms for simd alignment*, Proceedings of the 16th International Conference on Compiler construction (Berlin, Heidelberg), CC'07, Springer-Verlag, 2007, pp. 1–15.

[14] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam, *Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies*, Intl J. of Parallel Programming **34** (2006), 2006.

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

[16] Intel Corporation, *Intel ® cilk™ plus language specification*, 2010.

[17] Ken Kennedy and John R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[18] Samuel Larsen and Saman Amarasinghe, *Exploiting superword level parallelism with multimedia instruction sets*, SIGPLAN Not. **35** (2000), 145–156.

[19] Samuel Larsen, Rodric Rabbah, and Saman Amarasinghe, *Exploiting vector parallelism in software pipelined loops*, Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (Washington, DC, USA), MICRO 38, IEEE Computer Society, 2005, pp. 119–129.

[20] Y. Ökmen, *Simd floating point extension for ray tracing*, Master's thesis, Technische Universiteit Delft, 2011.

[21] Khronos Opencl and Aaftab Munshi, *The OpenCL specification version: 1.2*, 2011.

[22] Louis-Noel Pouchet, *PolyBench/C the polyhedral benchmark suite*, `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`.

[23] Ivan Pryanishnikov, Andreas Krall, Technische Universität Wien, and Nigel Horspool, *Pointer alignment analysis for processors with simd instructions*, In Proceedings of the 5th Workshop on Media and Streaming Processors, 2003, pp. 50–57.

[24] Jaewook Shin, Mary Hall, and Cha Jacqueline, *Superword-level parallelism in the presence of control flow*, Proceedings of the International Symposium on Code Generation and Optimization (Washington, DC, USA), CGO '05, IEEE Computer Society, 2005, pp. 165–175.

[25] Jan Sjödin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, and Antoniu Pop, *Design of graphite and the polyhedral compilation package*, Proceedings of the 2009 GCC Developers' Summit (Montre'al Canada), 06 2009, pp. 33–45.

[26] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen, *Polyhedral-model guided loop-nest auto-vectorization*, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (2009), 327–337.

[27] Sven Verdoolaege, *isl: An integer set library for the polyhedral model*, Mathematical Software (ICMS 2010) (Andres Iglesias and Nobuki Takayama, eds.), LNCS 4151, Springer-Verlag, 2010, pp. 299–302.

[28] Michael E. Wolf and Monica S. Lam, *A data locality optimizing algorithm*, Proceedings of the ACM SIGPLAN 1991 Conference on Programming language design and implementation (New York, NY, USA), PLDI '91, ACM, 1991, pp. 30–44.

[29] Michael Joseph Wolfe, *High performance compilers for parallel computing*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Appendices

Test programs

A

Listing A.1: prepared code from one-dimensional convolution

```
#define N 1024*1024
#define M 8
extern float x[N+M];
extern float b[M];
extern float y[N];
void convolve1d()
{
  int i, j;
  for (i = 0; i < N; i++)
  {
    for (j = 0; j < M; j++)
    {
      y[i] += b[j] * x[i+j];
    }
  }
}
```

Listing A.2: prepared code from two-dimensional convolution

```
#define N 2048
#define C 16
extern  float a[N*N];
extern  float y[N*N];
extern  float c[C*C];
void
convolve2d_bounded()
{
  int i,j,k,l;

  for (i = (C/2); i < N-(C/2); i++)
  {
    for (j = (C/2); j < N-(C/2); j++)
    {
      for (k = 0; k < C; k++)
      {
        for (l = 0; l < C; l++)
        {
          y[i*N+j] +=
            a[i*N+k*N+j+l - (C/2) - (C/2)*N]
            * c[k*C+l];
        }
      }
    }
  }
}
```

Listing A.3: pre-fissed code of sobel implementation

```
#define N 800
float data[N*N];
float sob[N*N];

float kx[3*3] = {       -1, 0, 1
                ,       -2, 0, 2
                ,        -1, 0, 1
                };
float ky[3*3] = {       -1, -2, -1
                ,       0, 0, 0
                ,       1, 2, 1
                };

float resx[N*N];
float resy[N*N];
int sobel()
{
  int x,y,k,l;
  for (x = 1; x < N-1; x++)
  {
    for ( y = 1; y < N-1; y++)
    {
      resx[x+N*y] = 0.0;
      resy[x+N*y] = 0.0;
    }
  }
  for (x = 1; x < N-1; x++)
  {
    for ( y = 1; y < N-1; y++)
    {
      for (k = 0; k < 3; k++)
      {
        for (l = 0; l < 3; l++)
        {
          resx[x+N*y] +=
            data[x+y*N + k -1 + (l-1)*N]
            * kx[k + 3*l];
          resy[x+N*y] +=
            data[x+y*N + k -1 + (l-1)*N]
            * ky[k + 3*l];
        }
      }
    }
  }
  for (x = 1; x < N-1; x++)
  {
    for ( y = 1; y < N-1; y++)
    {
      sob[x+y*N] = sqrtf
        (  resx[x+N*y]*resx[x+N*y] +
        resy[x+N*y]*resy[x+N*y]
        );
    }
  }
}
```

Listing A.4: prepared code from matrix multiplication

```
#define N 1024
extern float    A[N*N];
extern float    B[N*N];
extern float    X[N*N];
void mm()
{
  int i,j,k;
  for (j = 0; j < N; j++)
  {
    for (i = 0; i < N; i++)
    {
      for (k = 0; k < N; k++)
      {
        X[i+j*N] += A[k+j*N]
          * B[i+k*N];
      }
    }
  }
}
```

Curriculum Vitae

Earned a diploma in multimedia design at Grafisch Lyceum Utrecht, went on to achieve a Bachelor of ICT (with honours) at Hogeschool Utrecht. After working for one year, he decided to enroll in the MSc Computer Engeneering program at Technische Universiteit Delft. His hobbies include cycling, reading and computers.
Pepijn Westen