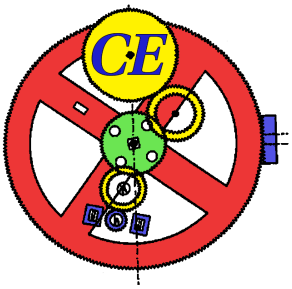
2009



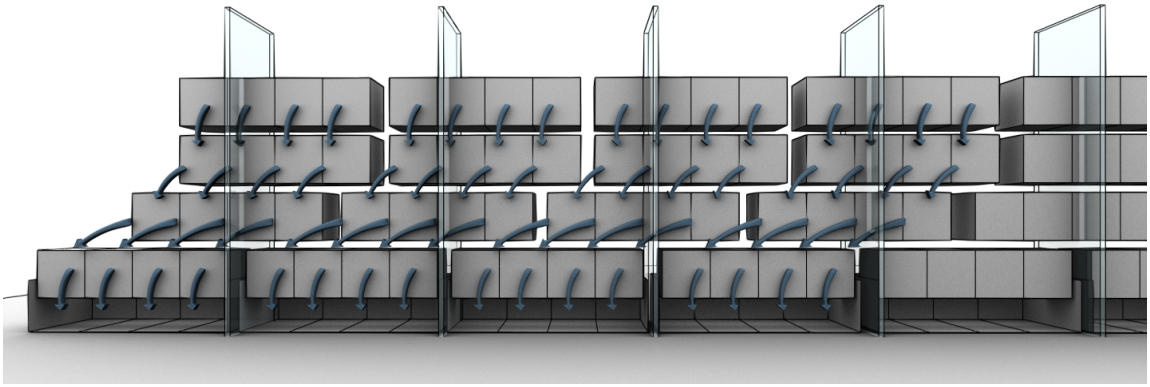
Computer Engineering
Mekelweg 4
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

ACE Associated Compiler Experts
De Ruyterkade 113
1011 AB Amsterdam
The Netherlands

MSc THESIS

A solution to misaligned data access in a vectorizing compiler framework

Sander de Smalen



Faculty of Electrical Engineering, Mathematics and Computer Science

A solution to misaligned data access in a vectorizing compiler framework

THESIS

submitted in partial fulfillment of the
requirement for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Sander de Smalen
born in Alkmaar, The Netherlands

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

A solution to misaligned data access in a vectorizing compiler framework

by Sander de Smalen

Abstract

Vectorizing code for short vector architectures as employed by today's multimedia extensions comes with a number of issues. The responsibilities of these issues are moved to the compiler in order to keep hardware simple. One of those issues is memory-alignment, which requires the compiler to guarantee loading and storing vectors at aligned addresses.
Previous work that covered this issue proposed a mechanism to reorder vectors at runtime to ensure proper alignments, while other work has focussed on finding a minimal number of reorderings. We combined these subjects into an in-depth research and implemented the optimization for the retargetable CoSy® compiler framework. Instead of solely focussing on the minimal number of reorderings, we also considered dynamic (runtime) properties which may enable latency-hiding of reordering operations. Furthermore, we performed a comparison of the presented reordering-techniques and researched the impact of other compiler optimizations on the proposed transformation. Finally, we placed our results into perspective with unaligned load/store operations supplied by our target architecture.
With our implementation, we were able to vectorize a number of applications for SSE and SSE2 vector extensions where alignment-issues were involved. For randomly generated loops we were able to achieve between 50% and 80% of the speedup obtained by unaligned memory instructions. (Our targeted architecture is less strict on memory alignment and supplies instructions that can handle misalignments by hardware). As for the benchmarks, we were able to achieve speedup factors of about 2.25x for a block-matching algorithm (combined with loop versioning to avoid runtime alignment), 1.6x for the SPEC95 Swim benchmark and a factor 4x for a Sobel FIR filter.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-31

Committee Members :

Advisor:	Ben Juurlink, CE, TU Delft
Chairperson:	Koen Langendoen, PDS, TU Delft
Member:	Marcel Beemster, ACE Associated Compiler Experts bv.
Member:	Hans van Someren, ACE Associated Compiler Experts bv.

Contents

Chapter 1

Introduction

This thesis covers a compiler optimization that serves as a solution for alignment issues that are found when automatically vectorizing applications for SIMD architectures. It follows from a research that was performed at ACE© Associated Compiler Experts, and finally the proposed solution was implemented for the CoSy® compiler framework. Developing the proposed optimization for a vectorization framework that is still in development, proved to be a challenging task. Our work involved more than the proposed optimization alone, as we encountered a number of details in the framework that needed to be reviewed in order to get proper results.

We will begin this thesis with an introduction to Short vector instructions (SIMD), auto-vectorization and the corresponding issues in Chapter 2. This chapter will also present an overview of the research that has been done on the field of auto-vectorization. Finally, Chapter 2 will elaborate on alignment issues and give the motivation for this thesis. This will be followed by Chapter 3, which will try to establish a conceptual understanding of our proposed solution. Chapter 4 will elaborate on the details of this solution, along with a more formal description of our approach. The actual implementation of the algorithm will be discussed in Chapter 5, as well as problems that we encountered during the implementation process. Chapter 6 will present the results of benchmarking our proposed optimization and will also elaborate on our benchmarking platform and several artifacts we encountered while benchmarking. We draw our conclusions in Chapter 7, followed by Chapter 8 which elaborates on areas of research that could extend our work. Finally Appendix A displays the source code of our benchmarks, and Appendix B [1] presents a report comparing the auto-vectorizing capabilities of several compiler frameworks.

[1] These pages may be left out, as it contains confidential information disclosed under an agreement with ACE Associated Compiler Experts bv.

Chapter 2

Background

Over the past few decades, the presence of vector architectures has resulted in a lot of research on the use of parallel vector instructions. However, the developed techniques are only partially valid for newly developed architectures that allow a restricted form of vector instructions. These restrictions typically include small vector lengths, alignment restrictions and restrictions on the used data types. This chapter will give an overview of the research that has been done on automatic vectorization for Short vector architectures, and shows the perspective and context in which this thesis should be placed.

```
#define SIZE    (1024*1024)

/* a,b,c are arrays representing
 * greycolor images              */
float a[SIZE];
float b[SIZE];
float c[SIZE];

void blend_images(float alpha)
{
     int   i;
     float      C1;
     float      C2;

     C1 = alpha;
     C2 = 1.0 - alpha;

     for(i=0; i<SIZE; i++)
     {
          a[i] = C1*b[i] + C2*c[i];
     }
}
```

(a) C-code

```
     ...
     ...
.SCALARKERNEL:
     movss     c(,%eax,4),%xmm5     // load scalar c[i]
     movss     %xmm1,%xmm4
     movss     %xmm0,%xmm6
     mulss     %xmm5,%xmm4          // mutiply C2*c[i]
     movss     b(,%eax,4),%xmm5     // load scalar b[i]
     mulss     %xmm5,%xmm6          // multiply C1*b[i]
     addss     %xmm4,%xmm6          // add scalar results
     movss     %xmm6,a(,%eax,4)     // store scalar
     incl %eax                 // i++
     ...
     ...
.SIMDKERNEL:
     movups    c(,%eax,4),%xmm4     // load vector from c
     movups    b(,%eax,4),%xmm5     // load vector from b
     mulps     %xmm3,%xmm4          // multiply with C2-vector
     mulps     %xmm2,%xmm5          // multiply with C1-vector
     addps     %xmm4,%xmm5          // add result vectors
     movups    %xmm5,a(,%eax,4)     // store vector
     addl $4,%eax              // i+=4
     ...
     ...
```

(b) Assembly-code for SSE

Figure 2.1: Example of alpha blending two grey-scale images

2.1 (Short) Vector instructions

Most programming paradigms are conceptually based on performing scalar operations with scalar data values. As physical limits for creating smaller and faster hardware are starting to appear on the horizon, the industry is beginning to focus more on parallelism. Programs often iteratively perform the same operations on large data collections. To exploit data parallelism in applications, instructions to handle multiple data values at a time have been proposed.

Traditional vector architectures try to implement this by performing a single instruction on a series of values (i.e. vectors). Traditionally, these vector architectures operate on large vectors, and often have support for scatter and gather operations to support non-sequential data access. Gather operations load multiple scalar values from several places in memory into a single vector, while scatter operations store scalar values from a single vector register to multiple locations in memory. But implementing these complicated operations in hardware is very expensive, and therefore not feasible for every application.

Short vector architectures try to bridge the gap between complicated vector architectures and scalar execution. By imposing restrictions on the memory architecture, the responsibility of properly exploiting data parallelism is shifted more towards the programmer and compiler. Scatter and gather operations will need to be programmed explicitly by the programmer. Streaming SIMD vector Extensions (SSE) for Intel processors are an example of such short vector architectures. SSE has SIMD (Single Instruction, Multiple Data) instructions that operate on vector registers of 16 bytes in length, containing either floating point or integer valued data. Supporting short vectors can have several advantages as opposed to larger vectors supported by traditional vector architectures. Short vectors do not require large numbers of arithmetic units, keeping the hardware cost manageable. The imposed restrictions on the use of these instructions (like supporting only aligned vector loads at consecutive addresses in memory), keeps the hardware simple, and therefore the latency of vector operations to acceptable proportions. This way, large stalls in the processor pipeline are prevented, causing SIMD operations to be used interchangeably with scalar operations, thus utilizing both scalar and vector units on the chip.

As the rest of this chapter is devoted to explaining the restrictions of short SIMD architectures, we will not go into further details here. We will end this section however, with an illustration of short vector instructions, which is displayed in Figure 2. The figure displays a program that blends two grey-scale images with a given blending factor. The assembly code of the loop-kernel is displayed on the right, showing both code for scalar instructions as well as SIMD instructions. Vector instructions have the 'ps' suffix in their name, meaning they operate on *Packed Single precision floating point* data, whereas scalar instructions use the 'ss' suffix denoting *Scalar Single precision FP*. Apart from the instruction names, the semantics are different as the vector instructions operate on 4 data values simultaneously.

2.2 Applications suited for SIMD

When performing SIMD operations on vectors, these vectors need to be read from and written to the memory respectively. When the vector elements are at adjacent addresses in memory, a single memory operation can load or store the vector. Otherwise, instructions that combine scalar values into a vector, or extract a vector into scalar values need to be issued. Therefore, the way data-structures are laid out in memory can be decisive whether an algorithm can be efficiently vectorized. Depending on the memory layout and the access pattern (see section 2.5.2) some algorithms are

```
struct
{
   float a[N], b[N], c[N];
}obj;
```

(a) Structure of Arrays

```
struct
{
   float a, b, c;
}obj[N];
```

(b) Array of Structures

```
struct
{
   float a[4], b[4], c[4];
}obj[N/4];
```

(c) Hybrid

Figure 2.2: Memory Layouts

more suited for SIMD execution than others. Before discussing applications suitable for SIMD execution, some definitions on memory layout are given.

Structure of Arrays (SoA) The layout shown in Figure 2.2.a shows a structure of arrays. The example shows that vectors can be accessed in one instruction since the objects are adjacent in memory. A drawback of this approach arises when there are a large number of members in a structure. All memory systems have a limit on the number of pages that are cached for quick access. When the number of members is too large, the memory system will have to reopen pages, causing unwanted overhead.

Array of Structures (AoS) Figure 2.2.b shows an array of structures, which is an intuitive paradigm for programmers. With an AoS, the elements are not adjacent which complicates compiler-analysis for vectorization. Depending on the algorithm, a matrix as formed by multiple vectors must be transposed before they can be executed by SIMD instructions. Compared to an SoA however, there are no caching and paging issues, but there may be the additional cost of transposing the values before being able to perform a batch of similar operations using SIMD instructions.

Hybrid A hybrid approach as shown in 2.2.c combines the advantages of AoS and SoA. The caching issues are overcome by having smaller member arrays while the property of adjacency is preserved.

The memory layouts as explained above form an important fundament on data parallelism which is exploited with SIMD instructions. The next section will describe some applications that exhibit the property of data parallelism.

Image Processing is one of the most obvious uses for SIMD operations. Most of these applications access the memory sequentially, transforming the data by constants or by a linear transformation of neighboring values. Color conversions are a good example of the former since they multiply a constant conversion matrix with each pixel, while FIR filters are a good example of the latter, since they perform a linear transformation of a pixel with its surrounding pixels. For image processing applications, the memory layout is important. Having an AoS (i.e. array of {R, G, B, A} values) or SoA (i.e. structure containing R, G, B, A arrays) can have an impact on the performance, since an algorithm can either apply their operations to each color layer separately, or on multiple color layers simultaneously.

Integer based calculations suffice for the precision in basic image processing algorithms. However, for more complex image processing algorithms where a higher precision is required, floating point operations are needed. An example of such an application is bilinear interpolation, where each pixel value is calculated by interpolating between its neighboring pixel

values. The precision requires floating point arithmetic, causing two data type conversions. One from integer (which is the image data type) to floating point before the transformation, and one back from floating point to integer after the transformation. Most SIMD architectures support this functionality.

A class of algorithms that is less suitable for SIMD instructions are algorithms that use the concept of histograms. With histograms, the value of a pixel is used as index in a table. This requires the SIMD architecture to support instructions where data can be used as an address for table lookup, which may not be available on any platform. Intel's SSE instruction set provides specific instructions for this purpose ([2]).

3D Applications perform many transformations on coordinate vectors. To transform a vector to another coordinate space, a vector must be multiplied with a transformation matrix. This transformation has to be performed on all the vectors in a 3D mesh by a single transformation matrix, so instead of performing additions and multiplications for each coordinate separately, these operations can be performed on vectors. Again, the memory layout is important, since having a AoS requires a transposition of the loaded vectors using permutation operations.

A transformation takes the form:

$$\underline{v}' = A\underline{v}$$

For coordinate space transformations, this becomes:

$$\begin{bmatrix} x' \\ \vdots \\ w' \end{bmatrix} = \begin{bmatrix} a_{00} & \cdots & a_{03} \\ \vdots & \ddots & \vdots \\ a_{30} & \cdots & a_{33} \end{bmatrix} \begin{bmatrix} x \\ \vdots \\ w \end{bmatrix}$$

When execution is performed on scalars, each element value is calculated as:

$$x' = x * a_{00} + y * a_{01} + z * a_{02} + w * a_{03}$$

When executed in parallel this becomes:

$$\begin{bmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} a_{00} \\ a_{00} \\ a_{00} \\ a_{00} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} a_{01} \\ a_{01} \\ a_{01} \\ a_{01} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} + \begin{bmatrix} a_{02} \\ a_{02} \\ a_{02} \\ a_{02} \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} + \begin{bmatrix} a_{03} \\ a_{03} \\ a_{03} \\ a_{03} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

Assuming the vector length is 4, this will calculate the transformation for 4 vertices at the same time, instead of one. As can be noted in the equation above, each element in the transformation matrix should be duplicated into a vector. Intel's SSE extensions can perform this operation in one instruction. An additional perspective divide for each element with $\frac{1}{w}$ can be performed by multiplying with a vector of values resulting from a reciprocal instruction (Section 2.3.1).

Video Applications are another example of data parallel applications, where MPEG is one of the most often used algorithms to code and decode image data. MPEG contains several steps that can truly benefit from vectorization. These steps include motion estimation, motion compensation, DCT and variable length encoding.

Motion estimation calculates the sum of absolute differences (SAD) of 16x16 blocks. The SAD is computed for a number of offsets in x and y direction from the source block. The offsets with the minimum SAD value (i.e. best match), are combined into a direction vector. By performing the SAD row-wise, the memory can be accessed sequentially. Matching for each offset is a computationally intensive process, as a result of which hierarchical subsampling is often performed as this decreases the size of the blocks and thus simplifies the search. This does however require some additional subsampling operations. For more on subsampling and motion estimation for SIMD instructions, see [2].

The Discrete Cosine Transform (DCT) and its inverse (IDCT) are computationally intensive parts in video compression and decompression respectively. The DCT transforms 8x8 blocks of spatial image data into a block of values in the DCT-domain. A two-dimensional (2D) DCT is performed efficiently by a one-dimensional (1D) DCT in the horizontal dimension, and a 1D DCT in vertical direction afterwards. [18] describes an algorithm that can implement the 1D DCT with only 11 multiplications and 29 additions for an 8 point DCT.

The first stage of the DCT/IDCT algorithm consists of additions/substractions of a combination of 8 values. By loading the values into vectors and reshuffling the order of the elements, the operations can be performed in parallel. The other stages of the DCT/IDCT can also be performed in parallel, but that requires some more reordering operations. The interested reader is referred to [23, 18].

DSP Applications perform their operations on one-dimensional arrays of data. Image processing applications are actually a specific (two-dimensional) case of DSP applications. DCT and Fourier transformations are widely used in DSP applications. As noted above, FIR filters and DCT coding and decoding are well suited for SIMD instructions. An example of a basic FIR filter can be seen in Figure 2.6.

2.3 Hardware architecture support

2.3.1 SSE extensions

Intel's Streaming SIMD Extensions (SSE) aim to optimize multimedia applications. SSE has evolved from the older MMX extensions, which provided integer SIMD instructions on 64-bit vectors (which were actually the x87 FPU stack registers). SSE however supports eight 128 bit dedicated vectors. It also provides a more extensive set of operations and supported data types (floating point SIMD instructions were introduced with SSE).[1] The SSE extensions have SIMD units that can process single and double precision floating point values, and word, double-word, quad-word integer values, and also provides corresponding conversion instructions. SSE supports several data reordering operations for elements within the vectors.

Since SSE is specifically suited for multimedia applications, it contains some instructions that are often used in these applications. Some example vector instructions are the reciprocal ($\frac{1}{x}$) which is used for perspective divides, square-root ($\sqrt{x}$) for calculating distances, and an SAD instruction (Sum of Absolute Differences) for motion estimation.[2] The square-root and its reciprocal (i.e. $\frac{1}{\sqrt{x}}$) are computationally intensive operations. Therefore, the processor obtains the value from a table of pre-calculated values, instead of calculating the value at runtime. To keep the tables manageable, the precision of the mantissa is limited to 11 bits, while 23 bits precision is offered by scalar single

precision instructions. To double the accuracy to 22 bits, the Newton-Raphson method could be used, which performs the operations as shown in equations 2.1 and 2.2.

$$rcp'(a) = 2 * rcp(a) - a * rcp(a)^2 \tag{2.1}$$

$$rsqt'(a) = 0.5 * rsqt(a) * (3 - a * rsqt(a)^2) \tag{2.2}$$

The above example shows the non-uniformity of SIMD processors, as several operations may have different precisions, instead of complying to the standard IEEE 754 floating point standard for each floating point instruction.

When it comes to alignment issues, SSE does not impose high constraints on the alignment of data, since it provides both an optimized aligned vector move (i.e. load/store) instruction, as well as an unaligned move instruction. The unaligned instruction imposes overhead compared to the aligned instruction (see Section 6.2), but allows some flexibility for the compiler. It is not allowed to give an unaligned address to the aligned vector move instruction, as this causes an exception.

2.3.2 Altivec extensions

Altivec extensions do not support unaligned memory access, even though the processor does not generate an exception when a vector from an unaligned address is requested. The low-order n bits (where $n =_2 log(vectorlength)$) of the address are discarded such that the address is always accessed at an aligned boundary. This restriction is compensated by an extensive unit for data reordering operations, such that explicit realignment operations can be efficiently implemented. The *vperm* instruction provides for any possible element reordering of a vector while allowing runtime operands for the reordering (i.e. the way the vector is reordered does not need to be a constant known at compile-time, but can be a varying value at runtime).

2.3.3 Hardware vs Compiler considerations

The simplicity and non-uniformity of the available SIMD architectures result in more complex compilers. By shifting the responsibility more towards the compiler, this allows for cheaper, faster, more energy-efficient hardware. However, these restrictions limit the set of code segments that can profit from vectorization, and the compiler will have to incorporate complicated analysis and techniques to deal with these restrictions to try and fully utilize the SIMD capabilities. The non-uniformity of the architectures is formed by a large variety in the set of supported data types, the architecture-specific constraints for alignment, variety in precision for different instructions and the set of provided instructions (Altivec has extensive reordering instructions, while SSE provides some multimedia specific instructions).

2.3.4 Support for irregular and unaligned data access

A hardware solution as proposed by [7] solves alignment issues and irregular array access patterns. Alignment issues are solved by implementing an *align and shift*-unit in hardware which extracts the unaligned vector from two buffers. One possible implementation is to have a split line buffer with a single-bank cache. A single cache line is buffered and shifted into the result of the other load, thus requiring two consecutive loads. Another implementation would be to have a dual-bank cache, separating the cache lines by their odd and even lines respectively. When a load crosses two cache-lines, these two lines can be accessed in parallel.

Irregular array access is implemented with a small multi-port memory unit. The unit is given the base address and array-indices, and returns the resulting vector by combining multiple loads. This operation requires that the unit is embedded in the memory architecture, which can be done transparently by implementing it as a cache, or more explicit by implementing it as on-chip memory. As a cache, this will require additional tagging logic on top of the packing logic. To keep the packing-cache coherent with the cache-memory, the packing-cache will need to be accessed for each (ordinary) memory access. This causes unnecessary overhead which may not be desirable (possibly due to power constraints). As an on-chip memory, the packing-buffer will have its own memory space. The compiler will have to insert operations to handle the dynamic memory management such as copy/copy-back functions.

2.3.5 Graphics hardware

Graphic cards rely heavily on the data-parallelism in graphical models and computations. Identical computations need to be performed on a large number of pixels or vertices, which is perfect for SIMD execution, having only one instruction for multiple data elements. This hardware can be found in game consoles like XBox and Playstation, which require extensive processing power trying to create life-like experiences.

Graphic cards have different SIMD processors called shader units for each step in the graphics pipeline [21]. Vertex and geometry shaders operate on multiple vertices simultaneously to perform geometrical and lighting computations. Fragment and texture shaders perform clipping, culling, texturing and other fragment related operations. Increasing the vector size of the SIMD unit increases the number vertices that can be handled simultaneously. However, the nature of SIMD is an issue when state-changes between model-primitives cause vertices to be processed differently. Therefore, the trend of shader-units in recent graphics hardware is shifting more toward the MIMD (Multiple Instructions Multiple Data) execution model, which often results in a hybrid of both MIMD and SIMD execution models [6].

2.4 Automatic vectorization

The purpose of automatic vectorization is that sequential code sequences can be automatically transformed into vector sequences. By performing general transformations for vectorization on sequential code, no architecture dependent details need to be embedded in the source program, keeping it portable for other architectures. A vectorizing compiler needs to find consecutive scalar operations to transform them to vector operations. Since 'loops' iteratively apply the same operations, they are very suitable for this transformation.

2.4.1 Loop Carried Dependences

Vectorization of code can be hindered by dependences, either between consecutive statements or between loop iterations. Dependency analysis needs to determine which dependences are present in the loops. If dependences exist or independence cannot be proven, the compiler should be conservative and assume dependence. Reading from and writing to the same memory locations in one statement or basic block can cause dependences, possibly preventing the use of vector instructions. When writing to a memory location that is read some iterations later, there is a loop carried (flow-) dependence between the statements. If the distance between the iterations is greater than the

```
/* global data */
float a[N], b[N];

for(i=0; i<N; i++)
{
    S1:    a[i+1] = a[i] + Const;
    S2:    b[i+5] = b[i] + Const;
    S3:    c[i] = c[i+1] + Const;
}
```

Figure 2.3: Reading and writing from/to the same address can cause dependences

vector length, vectorization can still proceed, since there will be no dependences between elements within the vector itself. To illustrate, let us review the dependences as depicted in Figure 2.3.

Statement S1 contains a loop carried (flow-) dependence of distance 1. The (backward) flow-dependence makes S1 recursive:

$$a[i] = Const * (i) + a[0], \forall i > 0 \tag{2.3}$$

When S1 is performed with vectors, the vector $\underline{a}$ from array a will be loaded from memory, and the vector $\underline{const} = [C_k, .., C_k]$ is added to $\underline{a}$. Equation 2.3 shows that the values within the vector should accumulate. Assuming the vector length is 4, the problem as described above does not hold for S2, even though the (backward) flow-dependence remains. In this case the distance is greater than the vector length, and therefore doesn't result in inter-vector dependences. Statement S3 contains a (forward) anti-dependence. Anti-dependences form no problem for vectorization since the value of c[i] is not dependent on the result of the previous iteration, so a vector can be safely loaded before the operation is performed. Do note however that anti-dependences cannot be discarded in general.

To be sure whether dependences exist, the compiler is required to know which memory is accessed. When the memory is accessed from a pointer (given for example as a function argument), the compiler has no knowledge about the layout of the memory. It is possible that the memory-blocks overlap, resulting in data-dependences. The compiler could try to check every use of the function, and try to determine whether the given pointers reference an aliased memory block. However, when the function is declared globally, it can be accessed outside of program scope, and the analysis does not hold. Another solution to this situation is to explicitly tell the compiler whether the function arguments have overlap in memory. The C99 standard uses the 'restrict' keyword for this purpose. When the compiler knows that the memory referenced by the pointers is not aliased, vectorization can proceed safely. An example of this can be seen in Figure 2.4.

2.4.2 Code and loop transformations

There are some code and loop transformations that can provide a solution when dependences are involved. Below we will describe some of these techniques. When dependences still remain despite the transformations, vectorization should be disallowed.

If-conversion

Control-flow by *if-else*-constructs cause control dependences withinin a loop body. As the control flow is calculated on a scalar value as opposed to a vector, this may result in different control

```
float array[N];

// compiling foo will result in erroneous output
void foo(void)
{
    bar_restrict(array+1, array);     // error
    bar_nonrestrict(array+1, array); // ok
}

void bar_restrict(restrict float* a, restrict float* b)
{  /* Guaranteed to have no dependences => vectorizable */
    for(i=0;i<N-1;i++)
        a[i+1] = b[i];
}

void bar_nonrestrict(float* a, float* b)
{  /* Should not be vectorized, as there
        are no guarantees about pointers */
    for(i=0;i<N-1;i++)
        a[i+1] = b[i];
}
```

Figure 2.4: Using the restrict keyword

```
/* before */
for(i=0; i<N; i++)
    if(a==0)
        b[i] = 0;
    else
        b[i] = c[i];

/* after */
if(a==0){
    for(i=0; i<N; i++)
        b[i] = 0;
}else{
    for(i=0; i<N; i++)
        b[i] = c[i];
}
```

(a) Loop Unswitching

```
/* before */
for(i=0; i<N; i++){
    a[i] = b[i] + 1;
    /* dependence a[i-1] */
    b[i] = a[i-1] + constant;
    c[i] = d[i];
}

/* after */
for(i=0; i<N; i++)
    c[i] = d[i];
for(i=0; i<N; i++){
    a[i] = b[i] + 1;
    b[i] = a[i-1] + constant;
}
```

(b) Loop Distribution

```
/* before */
for(x=0; x<N; x++)
    for(y=0; y<M; y++)
        a[y][x] = constant;

/* after */
for(y=0; y<M; y++)
    for(x=0; x<N; x++)
        a[y][x] = constant;
```

(c) Loop Interchange

Figure 2.5: Several loop transformations

flows for values within a vector. As the control flow should be the same for all values within the vector, this cannot be allowed. Some of these conditional statements can be rewritten using if-conversion by converting an *if-else*-construct into a series of *and* and *or* operations, that manipulate the outcome of the expressions, and transform a flow dependency into a data dependency [5, 3]. However, *if-conversion* is not always possible or feasible since both the *if* and *else* part need to be computed. If the expressions are large, the overhead imposed by computing both the if and else part may prove to be too high. Side-effects may limit the use of *if-conversion* as this can invalidate the transformation. Both edges of the condition are executed, so possible side-effects that had been previously guarded by the condition will be executed as well. Alternatively, some hardware architectures provide predicated instructions which are only executed when their corresponding condition is evaluated to true,

Loop unswitching

Another solution to control-flow dependences in basic blocks is loop unswitching (Figure 2.5(a)). If the test-expression of the statement is invariant, the compiler could create two versions of the loop. One loop when the condition is *true*, and one for which the condition is *false*. Now the control flow dependence is moved outside the loop, so the loop can be optimized for parallel execution.

Loop distribution

When there are multiple dependences in a loop body, loop distribution as depicted in Figure 2.5(b) can help to divide loop bodies by distributing statements coupled by dependences to separate loops. By moving statements unaffected by dependences into a new loop, these loops can be effectively vectorized, while the other loops are executed with scalars.

Loop interchange

Depending on the access pattern of the inner-loop, it may be beneficial to interchange the inner-loop with the outer-loop if this results in continuous memory access (types of access patterns are further elaborated in section 2.5.2). An example of loop-interchange can be seen in Figure 2.5(c). [14] provides a more detailed discussion on loop-interchange. [20] describes a similar approach that vectorizes outer-loops instead of inner-loops, using an unroll-and-jam technique (see Section 2.4.4). Depending on how the loop-nest is 'unrolled-and-jammed' it can be similar to performing loop interchange. However, unroll-and-jam is a more direct and more importantly, broader approach to vectorizing loop-nests.

2.4.3 Cost model

Even though the use of vector instructions could theoretically speed up the program, it is not always beneficial. The restrictions imposed by most SIMD architectures, can cause data reordering operations to be inserted to keep the vector-orderings and data-sizes consistent with each-other. This could occur when the program accesses the memory in a non-consecutive order while the SIMD-architecture does not support indexed vector loads/stores. Another situation that causes the compiler to produce overhead is when basic blocks can only be partially vectorized. The compiler generates instructions to extract/compact vectors from/to scalar values, to switch between vector and scalar execution. The produced overhead might render the execution with vector instructions

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

```
/* unroll outer loop 4x */
for(i=0; i<N; i+=4){
    s = 0;
    for(j=0; j<M; j++)
        s += x[i+0+j] * c[j];
    y[i+0] = s;
    s = 0;
    for(j=0; j<M; j++)
        s += x[i+1+j] * c[j];
    y[i+1] = s;
    s = 0;
    for(j=0; j<M; j++)
        s += x[i+2+j] * c[j];
    y[i+2] = s;
    s = 0;
    for(j=0; j<M; j++)
        s += x[i+3+j] * c[j];
    y[i+3] = s;
}
```

```
/* jam */
for(i=0; i<N; i+=4){
    s = {0, 0, 0, 0};
    for(j=0; j<M; j++){
        s[0] += x[i+0+j] * c[j];
        s[1] += x[i+1+j] * c[j];
        s[2] += x[i+2+j] * c[j];
        s[3] += x[i+3+j] * c[j];
    }
    y[i+0] = s[0];
    y[i+1] = s[1];
    y[i+2] = s[2];
    y[i+3] = s[3];
}
```

Figure 2.6: Unroll and Jam enhances performance of vector reductions for typical FIR filter

less efficient than when executed with scalars. The report in Appendix B reveals this effect for several benchmarks performed on the GCC and Intel compilers.

These undesirable situations can be prevented by using a cost-model that indicates whether or not vectorization is feasible. The cost-model could either be applied before or after vector-specific code transformations. By applying the cost-model before transformations, the estimated cost may prevent transformations from taking place. An example of such a cost model is described by [15]. By performing the cost-model afterwards, the costs can be calculated accurately and depending on the results, the compiler may choose to revert the vector-specific transformations.

Since vectorization generates overhead, it gains a benefit when the number of loop iterations crosses a certain threshold. Since the number of loop iterations can often not be determined at compile time, profiling the compiled program could help to determine the dynamic properties of the loops.

2.4.4 Generating SIMD instructions

Several techniques to generate SIMD instructions for statements in a loop body have been proposed. One widely used technique is based on Super-word Level Parallelism (SLP)[25, 17, 24]. With SLP, an innerloop is unrolled several times, and the algorithm aggregates statements that could form a SIMD instruction. The criteria to aggregate statements are based on the memory access and the performed operation. The number of times the loop is unrolled should at least correspond to the vector-length. However, when the vector-length is too large, the problem of finding a suitable grouping may become unmanageably large for a compiler. Therefore, this technique is specifically suited for short-vector SIMD architectures.

SLP forms an initial set of pairs by aggregating statements that access consecutive memory addresses with isomorphic operations. This initial set of pairs is then extended with other pairs by following their def-use/use-def chains. Pairs can only be added if their statement accesses an adjacent memory location, contain no dependences, are isomorphic and do not cross alignment boundaries. In a next step, these pairs are combined into groups of isomorphic statements. When

no dependences are violated, the statements in the groups are scheduled, and SIMD instructions are emitted.

By concentrating on the inner-loop in the loop-nest, only information about the memory access pattern of the inner-loop is known. A technique called *unroll and jam* unrolls the loop that encloses the inner-loop, and combines the statements in the inner-loop body (i.e. the *jam*). By vectorizing the outer-loop, the SLP algorithm has more information available on the memory access pattern, and could aggregate the statements differently than it would if only the inner-loop would be concerned. Especially reductions from an inner loop can benefit from this transformation, as can be seen in Figure 2.6. When reductions are present and only the inner loop is vectorized, this requires an additional loop that sums all the scalar values within the vector. Using the unroll-and-jam transformation however, a more natural vectorization of reduction is performed, without the need of an additional summation loop. Note that this also holds for other reduction operations like min and max.

Another technique by [26] uses *virtual vectors*. Instead of unrolling the loop and aggregating statements in a basic block, this technique performs its aggregation on unaltered loop bodies. This does not require unrolling of the loop or other transformations which may need to be reverted if vectorization appears unfeasible. A virtual vector of arbitrary length is created by aggregating isomorphic operations that operate on consecutive memory. At a later stage these virtual instructions are mapped onto actual instructions of the architecture vector length. [26] also implements recognition for reductions in short loops (loop bounds need to be known at compile-time).

Another common occurrence in loops are the presence of induction variables, that are used in computations. Inductions are difficult to vectorize for compilers, as they require a special construction. Expressions containing an induction variable need an *induction vector* that is updated each iteration. An induction vector (for an induction variable of form $ax + b$) is initialized with

$$[0 * v_{incr}, 1 * v_{incr}, ..., n * v_{incr}]$$

where v_{incr} is the increment-value of the induction variable. This vector needs to be updated each iteration with the vector

$$[n * v_{incr}, n * v_{incr}, ..., n * v_{incr}]$$

This does however require that the iteration step-size is a compile-time-known constant.

2.5 Loading and storing of vector data

2.5.1 Data types

There is a large variety in data type support on the various SIMD architectures. Some architectures allow only a small set of data types, while other architectures support a wide range of data types and operations on these types. SIMD operations in GPUs (see Section 2.3 for example), require a specific data type in each stage of the pipeline. The compiler needs to know whether certain conversions between one data type and the other are supported by the target platform. Mapping scalar code to vector code is not always possible or profitable due to architecture restrictions that might prevent data type conversions, or have limited support for packing and unpacking of data.

Issues could arise for code that operates on multiple data types within a single statement, thus requiring data size conversions within expressions. When the data types have different sizes, length conversion is required. Different data sizes result in vectors with a different number of elements.

```
/* global data */
float a[N], b[N], c[N]; // all are 16-bytes aligned

/* original loop */
for(i=1; i<N; i++){
   a[i] = b[i+1] + c[i+2]; // Three unaligned accesses
}

/* loop unrolled for first 4 iterations */
a[1] = b[2] + c[3];   // loads vector from:
a[2] = b[3] + c[4];   //     'a' at offset 4.
a[3] = b[4] + c[5];   //     'b' at offset 8.
a[4] = b[5] + c[6];   // 'c' at offset 12.
...
```

Figure 2.7: Example of unaligned vector access

Packing and unpacking instructions will need to be issued to convert these vectors to have identical data sizes and number of elements.

2.5.2 Memory accesses

Accessing the memory can be done in several ways. The accesses can be either consecutive or non-consecutive, where the latter can be strided or non strided. Below we list some definitions.

Unit-stride memory access: This is the simplest case and is supported by all vector architectures. In case of unit-stride access, values with consecutive memory addresses are accessed. This is also known as continuous, consecutive or stride-1 memory access.

Strided memory access: Accesses to the memory where the addresses are not consecutive, but are spaced with a fixed step size (the stride), causing the memory-addresses to be linear. Having strided memory accesses often prevents the compiler from vectorizing the code, since the overhead of permutation instructions needed to combine/extract the vector often outweights its advantage with respect to scalar execution. [19] describes a technique that effectively vectorizes strided memory accesses having a power of 2. When the address is required to be a power of 2, the compiler always knows the location of the value within the vector for each access. This enables the use of the simple virtual instructions, 'extract_even' and 'extract_odd', which can be efficiently mapped onto actual instructions for most platforms.

Indexed memory access: When the memory is accessed in a non-linear order, than this is called an indexed memory access. To access an indexed vector from the memory, scatter and gather operations must be supported by the hardware. For vectorization to be beneficial, hardware units must implement these operations, requiring complex and expensive scatter/gather units.[7] proposes a hardware solution that performs the task of packing and unpacking scalar values into/from vectors respectively (See section 2.3 for more details).

2.6 Alignment issues

The simplified memory architectures of SIMD processors often impose constraints on the alignment when accessing vectors in the memory. They require that the vectors are accessed at aligned

```
/* Assuming a, b and c have same alignment */
void foo(restrict float* a, restrict float* b, restrict float* c, int size){
   /* scalar execution */
   for(int i=0; mod(&a[i],VL) != 0 && i<size; i++)
      a[i] = b[i] + c[i];

   /* parallel exection, loop is now aligned to a[i] */
   for(; i<size; i++)
      a[i] = b[i] + c[i];
}
```

Figure 2.8: Dynamic loop peeling

boundaries in the memory, which means that the memory addresses must be a multiple of the vector length.

An example can be seen in Figure 2.7. Here the vectors are loaded at byte offsets 4, 8 and 12 for variables a, b and c respectively. Since the base addresses of all the memory blocks are aligned, the memory at these offsets (assuming $VL = 4$) will be unaligned.

While some architectures strictly disallow accesses at unaligned memory addresses, other architectures access memory only at aligned boundaries. Intel's SSE extensions relax the alignment constraints by providing specialized instructions that can access vectors at unaligned boundaries, but the use of these unaligned instructions imposes significant overhead, which should be avoided whenever possible. If no realignment techniques are used, vectorization might either not be possible or may not be profitable. Therefore, several techniques have been proposed to solve these issues, which will be elaborated on in this section.

2.6.1 Implicit realignment techniques

Several realignment techniques have been proposed to deal with misalignments for SIMD processors. Implicit as opposed to explicit realignment techniques, do not realign the data at runtime using specific instructions. It either aligns the data at compile time, or inserts code to select the load/store instructions depending on the alignment of the data. Each of the proposed implicit techniques has its pros and cons, and none of them can be seen as an optimal solution. A thorough comparison between the proposed solutions is given in [22]. We will discuss each of the techniques below.

Static Loop peeling This technique can be used to align one of the memory accesses in the loop, by peeling off a fixed and compile-time-known number of iterations from the loop. When multiple misalignments are involved, only one of them can be aligned by loop peeling, since aligning one of the memory accesses can misalign another formerly aligned access. Therefore, static loop peeling is only profitable when all memory accesses have the same misalignment. This technique is often limited by the restrictions that the alignments need to be known at compile time, which is a rather strict limitation. In that case dynamic loop peeling can prove itself useful.

Dynamic Loop peeling When alignment information is not known at compile-time, this technique can peel off several loop iterations from a loop until one of the memory addresses is aligned. When there are multiple misalignments in the loop body, dynamic loop peeling can be used in conjunction with other realignment techniques. An example of Dynamic loop peeling can be seen in Figure 2.8.

The algorithm introduces a small overhead, since the first few iterations need to be checked for their alignment. After that the loop is executed in parallel without any overhead. This causes the algorithm to be beneficial when the number of iterations is more than the vector length.

Multidimensional Array Padding When the innermost dimension of a multidimensional array is not a multiple of the number of elements in a vector, the alignment of the accessed values becomes dependent of the indices, and cannot be predicted at compile time. This effect can be seen in the following example, where the inner dimension of the array is one element short of being a multiple of the vector length.

```
/* assume VL of 4 elements */
/* global data */
float a[N][M][31];

for(i=0; i<N; i++)
  for(j=0; j<M; j++)
    for(k=0; k<31; k++)
      a[i][j][k] = constant_value;
```

The alignment of the array values of 'a', cannot be computed at compile time, since the alignment depends on the values for i and j. This prevents the loop from being properly vectorized. A solution to this problem would be to pad the inner-dimensions of the array with 'dummy' elements, causing the inner-array to have a size that is a multiple of the vector-length, thus resulting in compile time known alignments.

A problem with this approach is the need of support from the language standard. If the language provides no support, viewing the multidimensional array as a one-dimensional array after padding (for example when using a type-cast in C) would have undefined results. Accessing the elements in the one-dimensional array will not correspond with the elements in the two-dimensional array, since the padded elements are accessed as normal elements. To prevent this from happening, the compiler will have to perform an analysis to see if this situation occurs somewhere throughout the program. When the array is defined globally, and is accessible to code outside of the compiled program, the scope of the analysis is insufficient, as a result of which multi-dimensional array padding should not be allowed at all.

Multi-version code A relatively straightforward way to solve the problem of runtime-known alignments, is to use multi-version code. A dynamic check is performed to test the alignment of the memory access, and the corresponding code-version that is optimized to deal with that particular set of misalignments is executed. Depending on the number of possible misalignments, the combinations of these misalignments, and extensiveness of the implementation, the code size may 'explode'. Depending on the application, the compiler will have to decide between code size and performance. For desktop applications code size may be less important than for embedded devices. Profiling of the program could provide a better assessment [5].

Duplication of constant tables Applications that use constant tables in their calculations like DSP or Image processing applications (see section 2.2) could benefit from having multiple versions of their constant tables, one version for each misalignment. Often, constant tables are small in size (like most FIR filters for example), and duplicating the tables does not add too much overhead in memory size. By duplicating the constant tables, the algorithm can

```
/* original loop */
for(i=0; i<N; i++)
   a[i] = b[i+1] + c[i+1];

/* realign the expression */
for(i=0; i<N; i++)
   a[i] = realign_to_offset_0(b[i+1] + c[i+1]);
   /* or similarly (but using two realign operations) */
   a[i] = realign_to_offset_0(b[i+1]) + realign_to_offset_0(c[i+1]);
```

Figure 2.9: The realign-operation explicitly realigns the expression from element-offset 1 and 2, to offset 0

dynamically check the runtime misalignment, and use the corresponding aligned table. This is actually a special case of multi-version code.

2.6.2 Explicit realignment techniques

Even though the implicit realignment techniques as mentioned in the previous section can improve the alignment properties of the expressions, statements or data elements, it does not cover all situations where alignment issues are involved. The alignment issues as depicted in Figure 2.7 cannot be handled by using an implicit realignment technique. For this situation, loop peeling would only be able to align one of the accesses, while the other two expressions remain unaligned. In situations where the alignment of expressions are unknown at compile time (when pointers are involved, or runtime address offsets), only multi-version code would be able to form a solution.

[9, 10] Proposes a technique to create aligned accesses for each vector load, by inserting explicit instructions to extract the misaligned data from two consecutive aligned vectors. These permutation instructions realign the data at runtime by conceptually *shifting* the data-streams from one offset to another. A simplified example can be seen in Figure 2.9, while a thorough explanation is given in Chapter 3. Extracting misaligned data requires support for permutation instructions from the targeted platform. Profitability of vectorization depends on the number of required permutations and the cost of a permutation instruction on the targeted platform.

Figure 2.9 already shows that there are multiple ways to realign expressions, so finding the configuration with a minimal number of permutation operations is important. An algorithm as proposed by [9] uses heuristics to find a configuration for the permutation instructions. [11] claims that the problem of finding an optimal arrangement for realignment instructions resembles the proven NP-Hard problem in [8], but is different in some important details. The SIMD alignment problem is therefore presumed to be an NP problem, but no NP-hardness or NP-completeness proof has been given so far.

The heuristics given by [9] (so called 'shift-policies') each have its own characteristics. The least efficient policy (zero-shift) can be used in situations where unknown misalignments are present, while the other policies focus more on suppressing the number of permutations. These policies are further elaborated in Chapter 3.

Two algorithms as proposed by [11] try to find the minimal number of permutations for two specific situations. One being a statement with no common subexpressions, the other being a statement with only two misalignments and where common subexpressions are allowed. Both algorithms only focus on the placement of the shifts, and assume that each shift operation is performed with equal costs. [16] proposes an approach that is based on an ILP solver, to find an

optimal shift configuration.

2.7 Focus of this thesis

For this thesis project, the focus is on explicit realignment techniques. Explicit realignment techniques provide for a complete solution to alignment issues. The overhead of the permutation instructions is the result of a trade-off between performance, code-size and functionality when compared to the previously presented implicit realignment techniques. When the overhead caused by these instructions is minimized due to a minimal realignment configuration, the use of short vector instructions is still a great benefit compared to scalar execution (see Chapter 6). This thesis project is performed at the company ACE, which is known for its compiler framework, CoSy. As ACE is currently developing components that can optimize a compiler for short vector architectures, alignment issues are something they see as an improvement to their compiler framework. As both the interests of the university, as well as ACE needs to be conceded, our approach will show a trade-off between theoretical and pragmatical solutions.

This thesis covers both heuristics as well as an algorithm to find an optimized shift configuration based on the number of realignment and permutation instruction characteristics. Common subexpressions are not considered as we believe that this is outside the scope of the subject. Furthermore, an algorithm to insert the shift operations is covered, as well as an implementation of the proposed algorithms.

Chapter 3

Explicit Realignment

This chapter will will try to establish a conceptual understanding of explicit realignment and the solution that is presented in later chapters. Section 3.1 will begin with an example that illustrates the problems with multiple misalignments. Section 3.2 will present some definitions that are used throughout this report. The section that follows, 3.3, will explain how the misalignments can be resolved. This section is divided in two parts, one part that explains how to find a configuration to explicitly realign the expression (3.3.1), and another part that explains the generation of operations to perform this realignment (3.3.3).

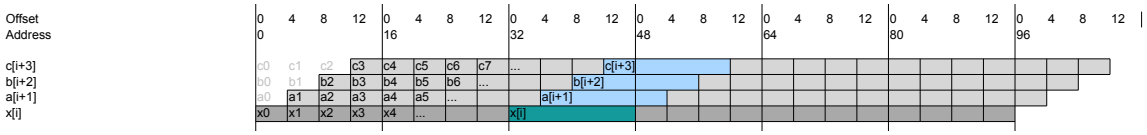
3.1 Introduction

Expressions containing memory operations with multiple alignments cannot be handled properly by implicit realignment techniques, as these can only remove one of the misalignments in an expression (see Section 2.6.1). To allow vector operations to be used for these expressions, explicit realignment techniques are required.

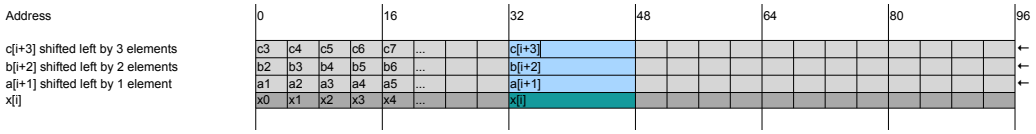
To illustrate this situation with an example, let's take a look at figure 3.1a, which shows the memory for operations of the expression *x[i] = a[i+1] + b[i+2] + c[i+3]*. The store for *x[i]* is aligned, as the vector to be stored does not cross any alignment boundaries. However, the vector load for *a[i+1]* is not properly aligned, as the load crosses the alignment boundary by one element. The same holds for b and c, where the loads cross the alignment boundary by 2 and 3 elements respectively. Note that in this specific case the store itself is already aligned by the statement definition itself. We assume that vector-store operations are not allowed to cross alignment boundaries. When the store is misaligned as is the case in the above-mentioned example, several loop-iterations should be 'peeled off' (see Section 2.6.1) to assure an aligned store. Note that peeling does not affect the relative offsets between the streams, and as a result only the loads are misaligned.

3.2 Definitions

In order to find a systematic approach to this problem, we first need to introduce the concept of 'streams' and 'stream offsets'.



(a) Unshifted streams



(b) Shifting stream a, b and c

Figure 3.1: Shifting the expression $x[i] = (a[i+1]^{0\leftarrow 1} + b[i+2]^{0\leftarrow 2} + c[i+3]^{0\leftarrow 3})$
(the $^{0\leftarrow 1}$ indicates a left shift operation from offset 1 to offset 0 on the expression node)

Memory Stream is the sequence of scalar values obtained by accessing an array during the lifetime of a loop. Note that in this research we only focus on memory streams having a unit-stride access pattern. For the subscript expression b[i+2] in Figure 3.1a, the memory stream results in the value-sequence: $\{\mathbf{b_2}, \mathbf{b_3}, \mathbf{b_4}, \mathbf{b_5}, \ldots\}$.

Register Stream is the sequence of vector registers produced by a single operation during the lifetime of a loop. This operation can be either a logical, arithmetic or memory-load operation. The stream of vectors resulting from such an operation is denoted as the register stream. For load operations, we assume that vector registers are loaded at aligned boundaries. This may cause the memory stream to be different from the resulting register stream. This can be seen in Figure 3.1a, where the register stream resulting from b[i+2] is the sequence $\{[b_0, b_1, \mathbf{b_2}, \mathbf{b_3}], [\mathbf{b_4}, \mathbf{b_5}, \mathbf{b_6}, \mathbf{b_7}], \ldots\}$. Note that the first two elements of the register stream are different from the first two elements of the memory-stream.

Shifted Stream is the sequence of vector registers produced after 'shifting' the elements in the register stream to a different offset (see Section 3.3). Shifting a stream can be seen as moving all (scalar) elements within the (vector) register stream several elements to the left or right, depending on the shift-direction. For the register stream b[i+2] as shown in Figure 3.1a, shifting the register stream two elements to the left results in the register stream $\{[\mathbf{b_2}, \mathbf{b_3}, \mathbf{b_4}, \mathbf{b_5}], [\mathbf{b_6}, \mathbf{b_7}, \mathbf{b_8}, \mathbf{b_9}], \ldots\}$.

Stream Offset can be defined as the byte offset of the first *desired* element in the register stream. For expression b[i+2], the first desired value is b_2, having an offset of 8.

Vector Length (VL) is the length of a vector in bytes.

Blocking Factor (BF) is the number of elements within a vector.

3.3 Shifting register streams

As demonstrated above, the sequence of values obtained from a memory stream can be different from a resulting (vector) register stream. A consequence of this is that operations defined on

scalar values cannot be directly mapped to vector operations when the elements within the register streams have different offsets. This would cause incorrect elements within the vector to be used for the operation. Therefore, vector operations can only be performed on register streams when they have similar offsets. As the memory-stream itself is fixed (i.e. the contents of the memory may not be changed as a side-effect to an expression), the discrepancy of offsets will have to be compensated in the register stream.

An example of this can be seen in Figure 3.1a, where the register stream of c cannot be added to the register stream of b, as the former stream is shifted right by one element compared to the latter stream. The same holds for stream b with respect to a, and a with respect to x. By conceptually *shifting* all streams to the same offset, the vector operations can be performed. The result of shifting all streams to offset 0 can be seen in Figure 3.1b. Note that shifting is conceptually simple, but requires actual instructions to perform the streamshift-operation at runtime. Section 3.3.3 explains what constitutes such an operation.

3.3.1 Finding a shift configuration

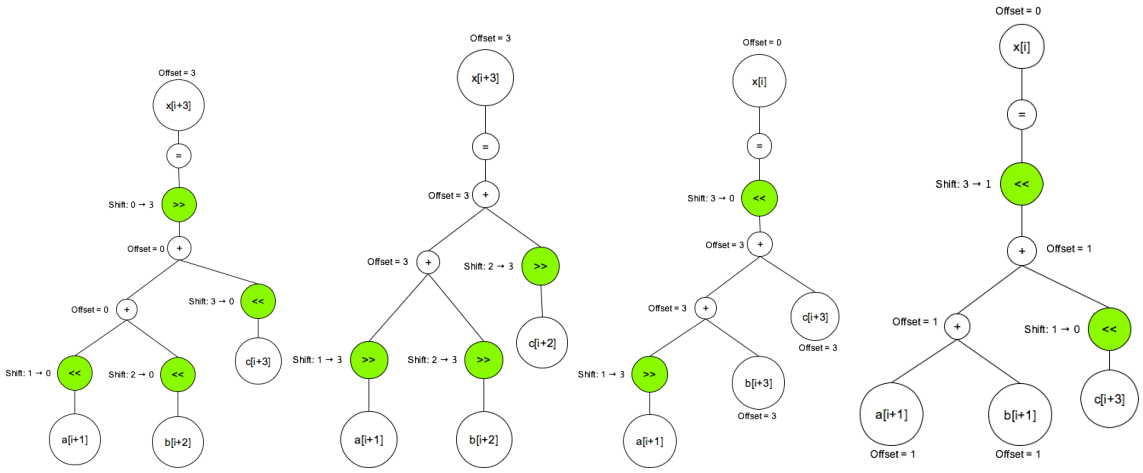
The previously mentioned example showed one possible way to shift the register streams and perform the vector operations, but multiple orderings are possible, where some orderings may be more efficient than others. Therefore, inserting shift-operations requires calculations of shift positions within the expression as well as the desired offsets.

The number of shifts has several important consequences on the program. For one, shifting a register stream to a different offset requires one or more permutation instructions (depending on the Instruction Set Architecture). When the overhead of inserting these instructions is significant, the speed-up which is obtained by the use of vector calculations may be diminished by the overhead produced by stream shift operations. Second, a shift-operation is performed on two consecutive vectors. Only one of those vectors has to be loaded on each vector iteration, as long as the other vector is kept from the previous iteration. Depending on the architecture and the number of shift operations in the expression, the program may be caused to spill some of these registers to memory if there are an insufficient number of available registers to store all the intermediate results. This may have a serious impact on performance as well. Intel's SSE2 extensions for example, provide only 8 vector registers. These 8 registers can contain at maximum three shifts, as three intermediate results can be stored for the next iteration, three vectors are calculated in the next iteration, and might require an additional 'scratch' vector-register for the shift operation itself.

Finding a configuration for the shifts within an expression can be a computationally intensive process as the problem itself is believed to be of the class of NP-complete problems ([11]). [11] proposes two algorithms to find the minimal set of shifts. These algorithms only focus on the positioning of the shifts and assume similar costs for each shift operation. As this assumption does not appear to be valid for our targeted platform (see Chapter 5), we will present an algorithm that tries to find a minimal solution that also involves individual costs for the shift operations. However, we will start with the heuristics as proposed by [9], also referred to as shift-policies.

Zero-shift policy This shift policy shifts the offset of every register-stream to zero, after which all the vector operations can be performed since all the registers now have the same stream offset. Finally, the resulting register-stream is shifted to the offset of the store in order to perform the store operation.

Eager-shift policy This policy is similar to the zero-shift policy, with the exception that the



(a) Zero shift (b) Eager shift (c) Dominant shift (d) Lazy property

Figure 3.2: Several Shift Policies

register-streams are shifted to the same offset as the store, instead of zero. This may save one shift operations compared to the zero-shift policy, as the additional shift to the store-offset may be omitted.

Dominant-shift policy This policy determines the most common stream-offset within the statement, and shifts the register-stream to that offset.

A useful property that can further reduce the number of shift-operations is the *Lazy shift property*, which causes the algorithm to postpone shifting when two connected nodes in the expression tree have the same offset, meaning that the result of an operator is shifted, instead of it's operands. This is allowed, since operations can be performed when two register streams share the same stream-offset. However, when the offsets are distinct, the shift-policy needs to decide the desired offset.

As described above, the zero-shift policy produces the least optimized results, and one can wonder whether the policy is useful. However, the zero-shift policy contains a property that can be useful when the offsets are unknown at compile-time but are runtime constants. The property that this policy inhibits, is that the shift-directions are known at compile-time, as opposed to the value of the constants. First, all offsets are shifted *left*, to offset 0, after which the offsets are shifted *right* to offset O_{src}. In theory only the direction of the shift needs to be known at compile time, but some hardware support to permute data with runtime-defined orderings is required as well. If supported by hardware, zero-shift policy is the only possible alternative when offsets are runtime defined.

3.3.2 Optimal shift configurations

As the number of misaligned accesses within an expression increase, chances of finding an optimal configuration using the simple heuristics decrease. The heuristics only use global properties from

the expression tree to calculate the offsets, disregarding local properties. Apart from looking solely at the number of shift operations within an expression tree, the above-mentioned heuristics cannot take the cost of the shift operations into account, while Section 5.4.2 points out that the shift-costs are not uniform for different shift offsets.

A dynamic programming algorithm can be used to find an optimal shift configuration. The algorithm generates an optimal offset-labeling for every inner-node in the expression tree while minimizing the cost of shifting from one offset to another. It does this by calculating partial (and optimal) solutions for each subtree in the expression. This process is started by considering the leaves of the expression. In each iteration, a larger subtree is considered, and the partial optimal solution for this subtree is calculated from the partial solutions of the child-nodes. This results in a list of costs (a cost for every possible alignment offset) for each of the inner-nodes. When all partial solutions have been calculated, a final walk through the expression tree chooses the optimal labeling given the predefined offset from the expression's left hand side.

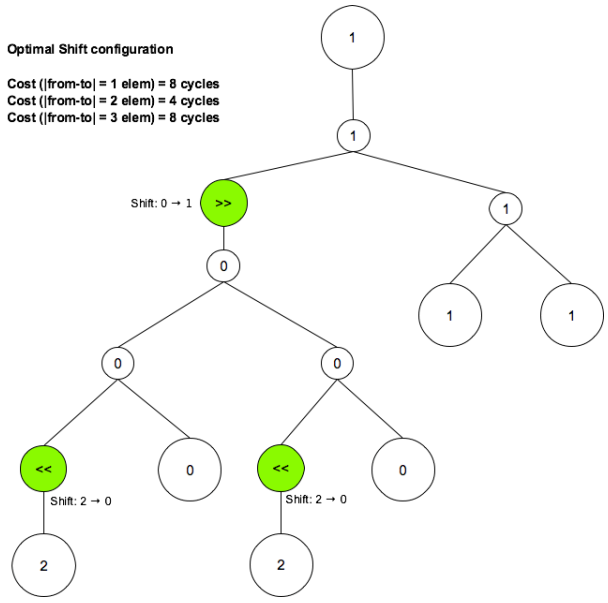
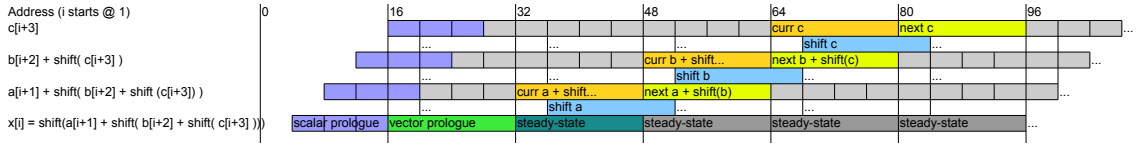
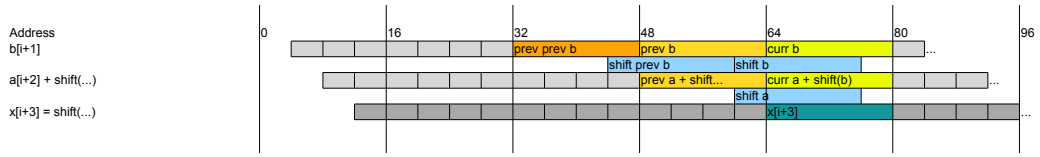


Figure 3.3: Optimal Shift configuration

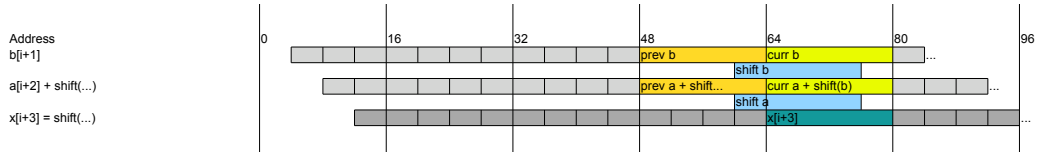
To illustrate the possible performance gain, Figure 3.3 shows the offset labeling for an expression that is vectorized for Intel's SSE platform. The costs for each shift is shown in the upper-left corner of Figure 3.3. Using the zero-shift heuristic (with lazy-property) would result in a shift-cost of 24 cycles { 2 x (2 → 0), 1 x (1 → 0), 1 x (0 → 1) }, while the eager-shift and dominant-shift heuristic would both result in a shift-cost of 32 { 2 x (0 → 1), 2 x (2 → 1) }. An optimal labeling however, only requires 16 cycles { 2 x (2 → 0), 1 x ((0 → 1) }. This result is not achieved by any of the heuristics using only global properties.



(a) Three left-shifts (SP) for the expression $x[i] = a[i+1] +^{0\leftarrow 1} b[i+2] +^{1\leftarrow 2} c[i+3]^{2\leftarrow 3}$



(b) Two right-shifts (non-SP) for expression $x[i+3] = a[i+2] +^{2\rightarrow 3} b[i+1]^{1\rightarrow 2}$



(c) Same as (b) but 'software pipelined'

Figure 3.4: Several stream shifts, containing both left and right shift operations

3.3.3 Realignment transformation

When a configuration for the shift-operations have been found, code can be inserted into the program, so the realignment can be performed at runtime.

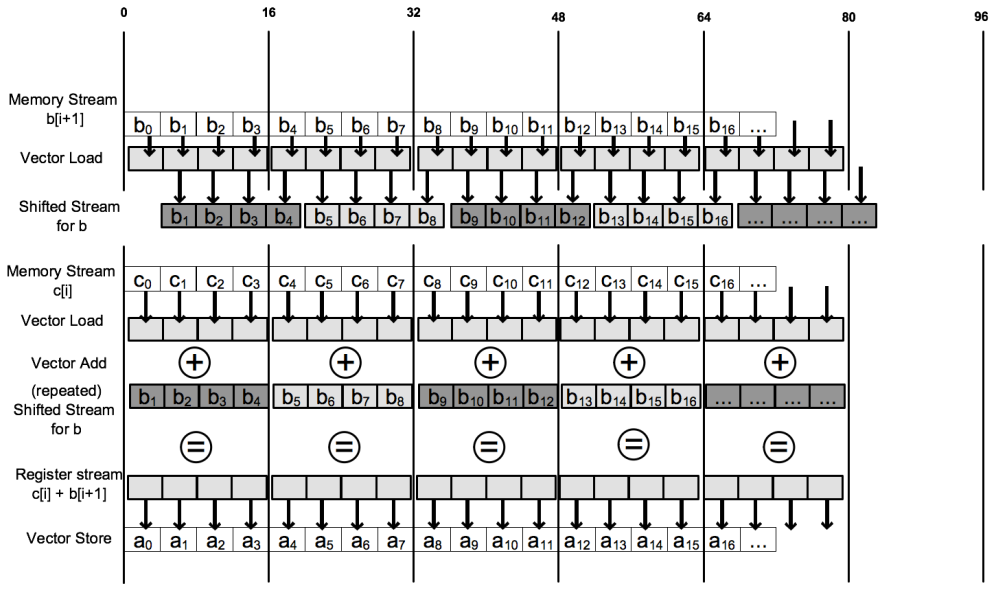
The concept of shifting a stream to another offset, can also be seen as extracting the desired (i.e. unaligned) register stream from two aligned and consecutive register streams. These streams can be denoted as *current* and *previous* register streams or *current* and *next* register streams for right and left shifts respectively. Combining these streams is done by extracting the desired values from the two vectors, and combining them into the new 'shifted' vector, which forms the shifted register stream. Extracting and combining the elements of a vector, can be done using permutation instructions that are available on most architectures.

Figure 3.5 illustrates the procedure with a simple example, by showing both (pseudo) code and graphical representation of the stream-shifting procedure. As can be seen, an aligned register stream is extracted from the misaligned register stream resulting from the vector load operation b[i+1]. Extracting is done by taking two subsequent vectors and selecting the required values to combine them into a new vector, thus forming a new (shifted) register stream. The pseudo-code in Figure 3.5a shows how this is performed element by element using scalar operations, but this can be mapped to specific permutation instructions for vectors in the back-end. As the resulting stream now has similar offsets as the other streams in the expression, it can be used to calculate b[i+1] + c[i]. Finally, the resulting vectors can be stored in *a*. Note that only 1 new vector actually needs to be loaded each iteration, as v2 could be used as v1 in the subsequent iteration.

Figure 3.4a illustrates a more complicated example where stream-shifts are performed on results of other stream-shift operations. In this example, the vector *shift(a)* extracts elements from *curr(a)* and *next(a)*, the vector-registers of the current and next register stream respectively. However,

```
int a[N], b[N], c[N]; /* aligned arrays */

/* Original loop */
for(i=0; i<K; i++)
    a[i] = b[i+1] + c[i];

=>

/* vectorized (unaligned) */
for (i=0; i<K; i+=4) {
    a[i+0..i+3] = b[i+1..i+4] + c[i+0..i+3];
}

=>

/* vectorized, realigned */
for (i=0; i<N; i+=4) {
    /* vector registers */
    int v1[4], v2[4], v3[4];

    /* two aligned vector reads */
    v1[0..3] = b[i+0..i+3];
    v2[0..3] = b[i+4..i+7];

    /* shift left by 1 element */
    v3[0] = v1[1];
    v3[1] = v1[2];
    v3[2] = v1[3];
    v3[3] = v2[0];

    /* two aligned vector reads and one aligned store */
    a[i+0..i+3] = v3[0..3] + c[i+0..i+3];
}
```

(a) Realigning expression *b[i+1]* in statement *a[i] = b[i+1] + c[i]*



(b) Visualization of (a)

Figure 3.5: Pseudo-code and corresponding visualization of extracting an aligned register-stream from a misaligned register stream.

next(a) is dependent on the vector *shift(b)*, which again extracts elements of the vectors *curr(b)* and *next(b)*. *next(b)* on it's own turn is dependent on the vector *shift(c)*, which is composed of elements from *curr(c)* and *next(c)*. Note that the example only shows how to compute the *current* vector of each shifted register stream, assuming the *previous* vector has been calculated in a previous iteration. In literature ([10, 9]), this is referred to as the Software-Pipelined (SP) version of the procedure, which stores the *current* vector as the *previous* vector for the next iteration. A non-SP version of the procedure will have to (re)calculate the *previous* vector every loop iteration as well. Figure 3.4c and 3.4b show both the SP and non-SP version for an expression with two (right) stream-shift operations. Note however that this definition of software pipelining is different from the original definition, as the latter actually refers to a combination of loop-unrolling and scheduling to remove latencies between instructions by filling the pipeline with instructions of subsequent loop-iterations.

When the register-stream of the store is misaligned as well, loop peeling is required to handle the first iterations until the stream is aligned, or until an alignment boundary has been crossed (due to address truncation, see Chapter 4). Storing partial vectors is not supported by most instruction sets, and writing outside the bounds of an array is illegal. Therefore, the first few iterations must be peeled off and handled using scalar execution. The same holds for the last few iterations that will cause a partial vector store. Therefore, the original loop is divided into three parts; a *prologue* which calculates the first few iterations using scalar arithmetic until the store is aligned, the *steady-state* loop, which performs the vector calculations and shifting of the streams, and an *epilogue* that handles the last few iterations of the loop using scalar arithmetic.

The *steady-state* loop is performed efficiently when software pipelining is used. However, when using the SP-version of the shifting-process, the values of the *previous* iteration need to be calculated in advance. Therefore, another prologue is required that calculates the vector registers from the *previous* iteration. This vector-prologue is scheduled right after the original scalar prologue, and is similar to the steady-state loop, but it does not use software pipelining and only calculates a single iteration (i.e. one whole vector). Figure 3.4a shows this partitioning of the loop.

As well as writing outside the bounds of an array might be illegal, the same holds for reading from memory locations outside array bounds. The example as shown in Figure 3.4a shows that calculating the result of the current vector, requires loading a vector of register stream a from 3 alignment boundaries ahead. This number is dependent on configuration and direction of the shifts (this will be elaborated on further in Section 4.4.2). This effect needs to be taken into account, as accessing memory outside bounds of the array may be disallowed, and care must be taken when setting the bounds of the steady-state loop. This has an impact on the number of iterations that are performed in the epilogue. A similar situation arises when the shifts are directed the other way, but this will influence the prologue instead of the epilogue.

Chapter 4

Algorithm

In this chapter we will present an algorithm that performs a code transformation to explicitly realign register streams.

This chapter is partitioned in four sections. We will start in Section 4.1 by defining the input to the algorithm and several constraints that apply in order for the code transformations to be valid. This is followed in Section 4.2 by several definitions that are used throughout this chapter. Section 4.3 explains an algorithm to calculate the shift configurations, followed by Section 4.4 that presents an algorithm to transform a shift-annotated expression into an expression containing solely aligned memory references.

4.1 Input

Section 2.4 described that inner-loops are most likely candidates for vectorization. As register streams follow from these consecutive loop iterations, the input to the algorithm are inner-loops with iteration counter i, lower bound LB, upper bound UB, increment $incr$ and body basic block $bodybb$.

4.1.1 Constraints

To validate the algorithm, several constraints are imposed on the inner-loop. As the underlying reasons are not similar for each constraint, we will classify the constraints into three categories:

Fundamental Constraints (FC) As the name implies, these constraints are imposed to prevent further processing of loops having fundamental limitations for vectorization. These limitations make the application of the algorithm theoretically impossible.

Constraints due to Implementation (IC) Limitations of the implementation are another reason to impose constraints on the input.

Constraints due to Performance (PC) Compilers always have to make a trade-off between performance and functionality. The proposed code transformations may not be profitable when certain code constructs are present. Therefore, violating these constraints may result in a decrease of performance.

(C1) **All memory references are either loop invariant or stride one array references (PC).** To be able to load consecutive values into one vector, the array references must be of stride one. When the stride is higher than 1, scatter and gather operations are required to store and load vectors respectively. These operations are often not supported by short vector architectures, and performing them manually may completely diminish the effect of vectorization. For loop invariant memory references, the value of the operation can be assumed to be in a register during the loop. This constraint can be split up into several other constraints, as shown below.

(i) **Memory references are linear subscript expressions with stride one (i.e. *baseaddr[i + C]*).** When calculating the properties of a memory reference during compilation, we can only take one variable dimension into account. The loop variable that is most applicable to span this dimension, is the iteration counter, which itself is subject to constraints (C1ii). To simplify the analysis, we prohibit the use of other (induced) loop variables.

(ii) **The loop Increment, *incr*, is a constant equal to 1.** To be able to determine whether memory references are of unit-stride, the increment of the loop must be constant. When the constant is equal to 1, determining the stride of a linear subscript expression is straight-forward.

(C2) **The alignment of each memory-reference is calculable at compile-time (FC).** To calculate the relative offsets between memory-streams, we need to determine the alignment of memory-references. As the algorithm is based on these relative offsets, this constraint is fundamental (see also Section 4.4.5). The alignments are determined by all elements that make up a linear subscript expression. The constraints below require some of these elements to be either compile-time constants, or require their modulo behavior with respect to the Blocking Factor to be known (i.e. value *mod* BF). A loop-versioning transformation can *bypass* the below-mentioned constraints by generating separate loops where each of the constraints are met.

(i) **C *mod* BF in *baseaddr[i+C]* is known at compiletime.**

(ii) ***LB mod* BF is known at compile time.**

(iii) **Base alignment of each array is known at compile time.** This constraint is important for arrays that are referenced by pointers and arrays that are allocated on a non-aligned stack frame.

(C3) **The body of the loop contains solely unconditional assignment statements (IC / FC).** Conditional statements (like if-else constructs or conditional assignments) might create inter-vector dependences or change the control flow of the program. A thorough analysis of the loop body might be able to determine whether these dependences disallow vectorization, but as these analysis are currently not reliable enough, we chose to prohibit conditional statements within the loop body. For the same reason, this constraint also applies to unconditional branches and function calls.

(C4) **There are no loop-carried dependences that hinder the vectorization process (FC).** As explained in Chapter 2, dependences can influence vector calculation. This means that

no loop-carried flow dependences with a length smaller or equal to the *blocking factor* are allowed. Note that this also covers reductions (like *x += a[i];*), as there is a loop carried flow dependence between the value in the previous iteration and the current iteration. To still allow vectorization of reductions, a manual unroll-and-jam scheme (see Section 2.6) could be applied.

(C5) **All memory references write and access data of the same length (IC).** No type conversions between data of different lengths are allowed due to implementation issues as explained in Chapter 5.

(C6) **Iteration counter only appears in address computation of stride-one references (IC).** Vectorizing expressions containing the iteration counter (or other loop induction variables) is possible but require the use of special induction-vectors. As this problem is outside the scope of this thesis, we will not elaborate on this any further.

(C7) **The loop must have an incrementing iteration counter (IC).** This is purely an implementation constraint to keep the implementation a bit more simple. When the iteration counter is decrementing, the calculations for loop boundaries and offsets are different. One could choose to perform a loop-reversal transformation before applying explicit realignment.

The constraints as presented above apply to both explicit realignment and to vectorization in general. Constraints C2, C5 and C7 are specific to our implementation of explicit realignment.

4.2 Definitions

The algorithms as presented below use the following data types.

(i) **AlignmentTuple** = { Expression, Alignment, Offset}
This data type is used to store alignment and offset information for each node in an expression tree. The Alignment field denotes the base-alignment of all referenced arrays in the subtree. The Offset denotes the position of the first desired element in the register stream.

(ii) **ShiftstreamTuple** = { Expression, From, To}
This data type represents the stream-shift operation as found by the analysis algorithm. Applicable nodes within the expression tree are annotated with this information. The register stream of Expression is shifted from byte offset From, to byte offset To.

4.3 Calculating shift configurations

The calculation of a shift configuration for an expression basically consists of three operations. First, the algorithm needs to be initialized by annotating the expression tree with AlignmentTuples. Then the alignment/offset information that is calculated for the leaves, needs to be propagated towards the root of the expression tree. When the propagation conflicts due to a multitude of possible offsets, a shift operation needs to be inserted. The type of shift chosen during this step is determined by the desired policy.

4.3.1 Calculating the peel factor

To facilitate the prologue loop, a peeling factor needs to be calculated. This is done to assure that for each statement, at least all partial stores have been handled by scalar operations, before vector instructions can handle the full vector stores. As the loop body may contain multiple assignment statements, the peeling factor depends on the store offsets of all assignments. When all store offsets are equal, the required peeling is equal to *BF - offset(stmt.LHS)*. Peeling to this amount will precisely align the store offset for each statement. When the store offsets are distinct, the peeling factor depends on the smallest offset. The reason for this is that the algorithm will *truncate* memory references to result in aligned addresses. Therefore, the store offsets should have been peeled the right amount of iterations, such that for each statement an alignment boundary has been reached. Therefore, the required peeling factor when offsets are distinct, is equal to:

$$\begin{aligned} Peel &= BF - min(offsets) \\ offsets &= \{offset(x.LHS) \quad | \quad stmt(x), x \in LoopBody\} \end{aligned} \tag{4.1}$$

This peeling factor is combined a step later with the value *ProPeel* (see Section 4.4.2). It is important that the peeling factor is determined before calculating the shift configuration, as the number of peeled iterations influences the starting offset of each memory reference expression (as these depend on the initial value of the iteration counter), and therefore influences the directions of the shifts.

4.3.2 Initialization

Initialization of the algorithm is done by calculating alignment and offset information from information already available in the program like the base address, index expression, and peeling factor, for every leaf node in the expression tree. Equations 4.5 and 4.6 show how to calculate the offset and (base) alignment of each memory reference respectively.

When the alignment of a base address is not known (like for instance when dereferencing a pointer), the Alignment value is defined as the natural alignment of the expression type, as this is the minimally required alignment for an array. For this case the offset is always set to 0 (as there is no offset with respect to a vector). For constants, no offset information applies, as shown in Equations 4.7. For non-leaf nodes in the tree (4.8), the offset cannot yet be determined.

$$AlignmentTuple(expr) = \{expr, Offset(expr), Alignment(expr)\} \tag{4.2}$$

$$Offset(a) = Address(a[0])\%VL \tag{4.3}$$

$$Alignment(a) = \begin{cases} VL & \textit{if Offset(a) = 0} \\ elementsize_{bytes}(a) & Otherwise \end{cases} \tag{4.4}$$

$$Offset(a[i+k]) = Offset(a) + elementsize_{bytes}(a) * (i_0 + k) \tag{4.5}$$

$$Alignment(a[i+k]) = Alignment(a) \tag{4.6}$$

$$Offset(C) = NO\_OFFSET \tag{4.7}$$

$$Offset(operation) = UNKNOWN\_OFFSET \tag{4.8}$$

4.3.3 Determining offsets and stream-shifts

When all the leaf-nodes have been supplied with offset and alignment information, the other nodes in the expression will need to be supplied with these annotations as well. Note that this only needs to be done for offset-annotations, as the alignment information is only required to impose the constraints (C2). The offset-annotations can be propagated upwards from the leafs toward the root of the tree. The shift-policies determine rules to do this. We will assume the property of lazy-propagation as this reduces the number of unnecessary shift operations. Lazy propagation implies that when a node has child nodes with similar offsets, it will be given the same offset as its children. Invariant child-nodes have no memory-offset or alignment. Therefore, the offset of the parent-node will depend on the offset of the non-invariant child-node. When both child-nodes are non-invariant and both have dissimilar offsets, the shift-policy determines the offset of the parent node. These shift policies differ in their choice of offset. For the zero-shift policy, the choice is constant, namely offset 0. The eager-shift policy takes the offset of the statement's LHS as its choice. The dominant-shift policy chooses the most common offset found throughout the statement as the offset. Finally, the child-nodes will be annotated with Shiftstream-tuples to indicate that the expression subtree needs to be shifted to another offset.

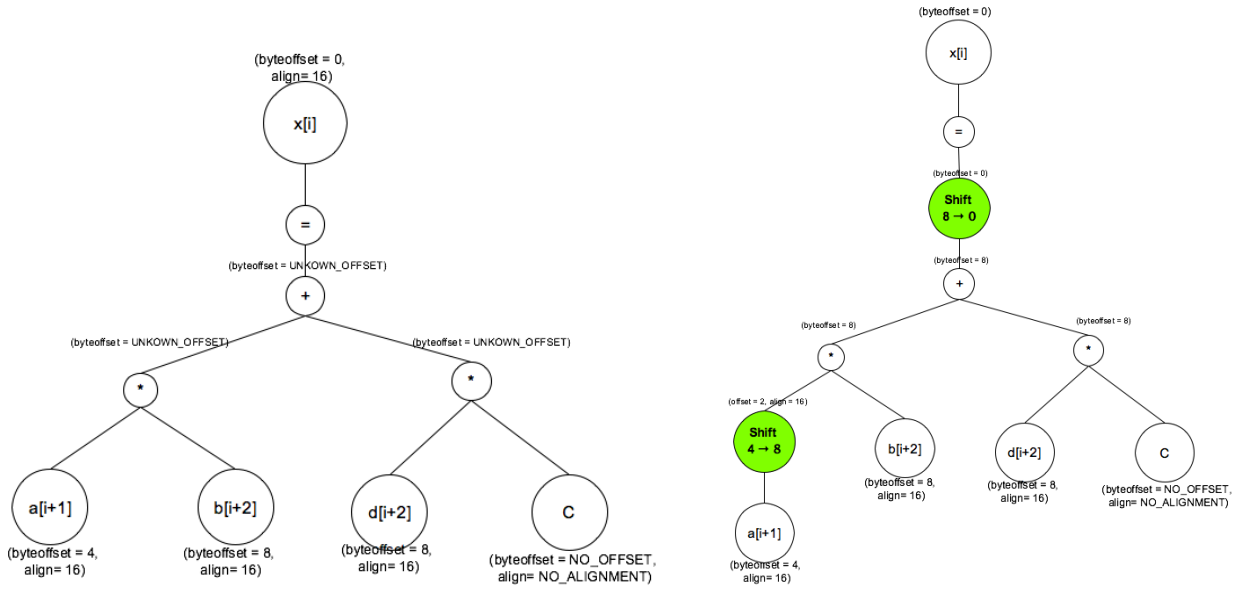
```
/************************************
 * This function calculates the shift-
 * configuration based on given policy
 ************************************/
calc_shifts(stmt:Statement, shift:Policy)
{
    switch(shift)
    {
        case zero_shift:
            desired_offset = 0;
            break;
        case eager_shift:
            desired_offset = stmt.Lhs.offset;
            break;
        case dominant_shift:
            desired_offset =
    max(histogram(get_offsets(stmt)));
            break;
    }


    postwalk(
        insert_shifts(stmt.Rhs, desired_offset)
    );

    create_shift(
        ShiftstreamTuple(
            stmt.Rhs,
            stmt.Rhs.offset,
            stmt.Lhs.offset
        )
    );
}
```

(a) Algorithm to align a statement

```
/************************************
* Walk the expression tree in post order,
* and call this function for each node
************************************/
insert_shifts(root:Node, desired_offset:int){
    if(root.offset != UNKNOWN_OFFSET){
        return;
    else if(is_unary(root)){
        root.offset = child(root).offset;
    }else if(is_binary(root)){
        if(left(root).offset == right(root).offset){
            root.offset = left(root).offset; // = right(...
        }else if(left(root).offset == NO_OFFSET){
                root.offset = right(root).offset;
        }else if(right(root).offset == NO_OFFSET){
            root.offset = left(root).offset;
        }else{
            create_shift(
                ShiftstreamTuple(
                    left_child,
                    left_child.offset,
                    desired_offset
                ));
            create_shift(
                ShiftstreamTuple(
                    right_child,
                    right_child.offset,
                    desired_offset
                ));
            root.offset = desired_offset;
        }
    }
}
```

(b) Insert shifts where necessary

Figure 4.1: Pseudo-code to determine shift-configuration with heuristics.



(a) Step 1 (b) Step 2

Figure 4.2: Example of algorithm

4.3.4 Example of applying heuristics

To clarify the above algorithm, we will end this section with an example. Figure 4.2a shows the annotated expression tree after initialization of the algorithm. Here, all memory references are accessing an array with an aligned base address. As the loop iteration counter i starts at 0 (we assume no peeling in this example), the offsets are equal to the index expressions. For the loop-invariant value C, no alignment and offset information is available. The offset values of all non-leaf nodes (multiplication and addition) are initialized as unknown.

As shown in Figure 4.2b the (byte)offset of the node *d[i+2]*C* is 8, because C has no offset causing the offset of *d[i+2]* to be propagated. For the multiplication node *a[i+1] * b[i+2]*, the offsets of *a[i+1]* and *b[i+2]* are distinct, and a shift-policy will have to determine an appropriate offset. In this example we have chosen for the dominant shift policy, which determines that (byte) offset of 8 is the most common offset in the statement. As a result, *a[i+1]* must be shifted to offset 8. For the addition node, the offsets of both child-nodes are equal causing the lazy-shift property to determine the same offset for their parent-node. Finally, the whole right-hand-side of the statement (now having offset=8) should be shifted towards the offset of the store (i.e. offset=0).

4.3.5 Optimal offset labeling

Calculating an optimal shift configuration follows from labeling the expression tree in a way that minimizes distinct offset between parent and child nodes[1]. This labeling is done using a dynamic

[1]We should note that this algorithm only applies to simple expressions without common subexpressions.

programming algorithm that consists of two stages. The first stage calculates the cost for each offset for every inner-node in the tree. For an expression tree with K inner-nodes, this means calculating all values of the K x BF cost matrix. The calculated cost for offset j is defined as the lowest calculated cost of moving from offset i to offset j for any i. The resulting cost for offset j of node k is assumed to be the minimal cost of the entire subtree when offset j is chosen. This assumes however that the costs of the offsets for the subtree of k are calculated similarly (and therefore display the minimal costs as well). The labeling cost for the leafs of the expression are initialized by setting the cost for the actual offset to 0, while the other offsets are initialized with infinity, as the leafs initially have a fixed offset. This step is performed bottom up, by considering a larger subtree every iteration.

```
/************************
 * Call in post-order
 ************************/
void step1(node){
   if(is_leaf(node)){
      foreach(offset in 0 to BF)
         if(node.offset == offset)
            cost[node][offset] = 0;
         else
            cost[node][offset] = INFINITY;
   }else{
      foreach(offset_p in 0 to BF){
         cost = 0;
         foreach(child in node.children){
            for(offset_c=0 to BF)
               min = MIN(min, cost[child][offset_c]
                  + calc_cost(offset_c, offset_p));
            cost = cost + min;
         }
         cost[node][offset_p] = min;
      }
   }
}
```

(a) Calculate and initialize cost for each offset and node

```
/*************************************
 * Call in pre-order
 * (statement.LHS is considered as the
 *  'root' of the expression tree )
 *************************************/
void step2(node)
{
   if(is_leaf(node))
      return;

   foreach(child in node.children)
   {
      if(! is_leaf(child)
      {
         foreach(offset in 0 to BF){
            cost[child][offset] +=
               calc_cost(offset,node.offset);
         }

         child.offset = ARGMIN(cost[child]);
      }
   }
}
```

(b) Choose alignment offset for nodes

```
void optimalshift(statement)
{
   int cost[size(get_nodes(statement))][BF];

   postwalk(statement, step1);
   prewalk(statement, step2);
}
```

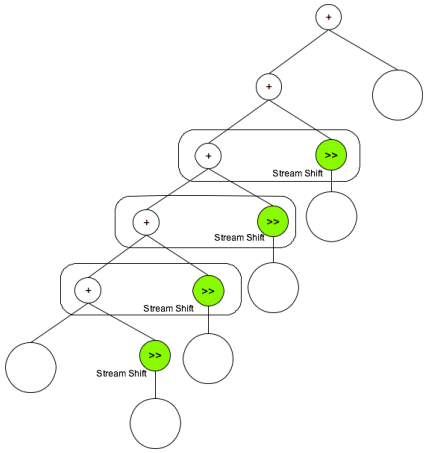
(c) Start of algorithm

Figure 4.3: Algorithm to calculate an optimal shift configuration

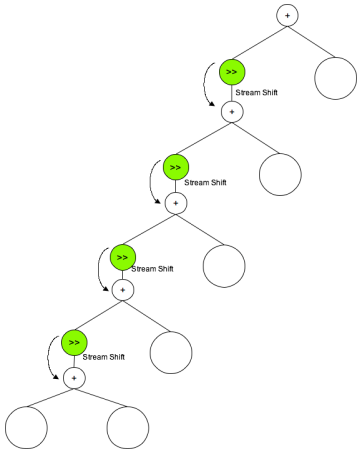
A second stage will actually label the nodes by choosing the cheapest offsets using offset-information of the left-hand side of the assignment. This is done in a top-down fashion, with the offset of the left hand side determining the choice of the root node for the right hand side. For each inner node, the costs are recalculated for every possible offset using the offset of its parent-node. The resulting offset will display the offset with lowest cost of choosing labeling j, given labeling i of the parent node. This step uses the cost of the entire subtree which has been calculated in the first stage. The pseudo-code for the algorithm is shown in Figure 4.3.

The above algorithm calculates the cost based on the individual cost of shift operations. How-

ever, instead of only considering the cost of shift operations, the dependence between shift operations may have an influence on performance as well. When considering a left recursive tree for example, generating the shift operations to be on the critical path may result in poor performance, as there is a data dependence between every step of the calculation from which the latency cannot be hidden. Generating the shift operations to be on the leafs of the expression, does allow for latency-hiding, as the shift operations can be scheduled along with other loads or arithmetic operations. Figure 4.4 displays this effect. To provide a mechanism to reduce this dependence the cost calculation can be expanded. This translates into adding additional cost for having a different labeling between the parent node, and a non-leaf node. This will encourage the algorithm to support shifting the offset of the leaf nodes instead of the inner-nodes. This prevents the algorithm from choosing a distinct labeling between two nodes on the critical path.



(a) Shift operations on leaf nodes (blocks around nodes indicate hiding of latency, as these can be performed simultaneously)



(b) Shift operations on inner nodes (arrows indicate data dependence between nodes, preventing latency hiding)

Figure 4.4: Latency hiding for left recursive trees

4.3.6 Example of finding optimal labeling

Figure 4.5 displays the expression tree for x[i+1] = (a[i+2]*b[i+0] + c[i+2]*d[i+0]) + (e[i+1]*f[i+1]). The first pass of the algorithm will try to calculate the cost for each offset, for every non-leaf node in the expression. This is done from the bottom up towards the root. Calculating the cost for offset 0 for the subtree *c[i+2]*d[i+0]* for example, the cost is defined as $min(C) + min(D)$ where

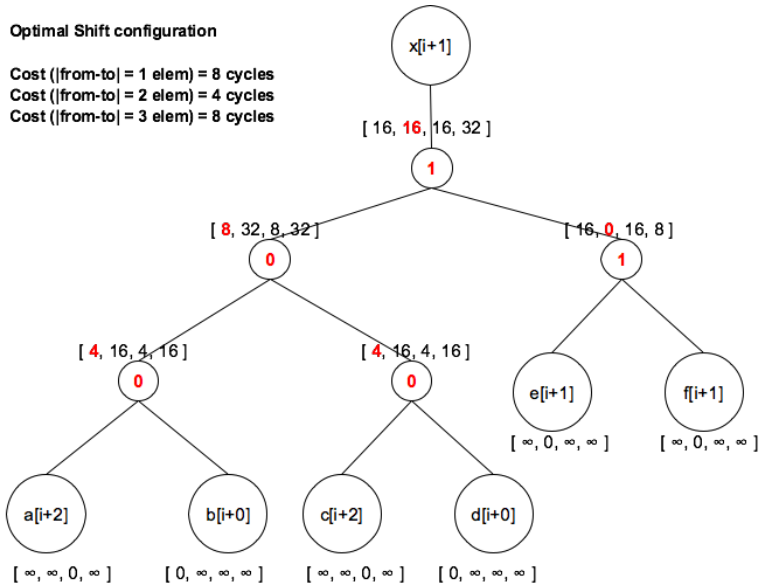
$$C = [(0 \to 0) : \infty + 0 = \infty, \quad (1 \to 0) : \infty + 8 = \infty, \quad (2 \to 0) : 0 + 4 = 4, \quad (3 \to 0) : \infty + 8 = \infty]$$
$$D = [(0 \to 0) : 0 + 0 = 0, \quad (1 \to 0) : \infty + 8 = \infty, \quad (2 \to 0) : \infty + 4 = \infty, \quad (3 \to 0) : \infty + 8 = \infty]$$

resulting in $min(C) = 4$ and $min(D) = 0$ resulting in a cost of 4. The resulting cost-vector of this subtree is used to calculate the cost of its parent node, *(a[i+2]*b[i+0] + c[i+2]*d[i+0)*. Finally, when the cost-vectors of all subtrees have been calculated, the labeling is performed using the LHS

of the statement. In this case, the LHS has an (element) offset of 1. For the root-node, the chosen label is defined by

$$argmin((0 \rightarrow 1) : 16 + 8, \quad (1 \rightarrow 1) : 16 + 0 \quad , (2 \rightarrow 1) : 16 + 8 \quad , (3 \rightarrow 1) : 32 + 4)$$

This offset is then used to determine the offsets of their child-nodes in the same way as shown above. By recursively labeling the nodes down towards the leafs, an optimal labeling for the expression-tree is chosen.



(a))

Figure 4.5: Finding an optimal offset labelling for: *x[i+1] = (a[i+2]*b[i+0] + c[i+2]*d[i+0]) + (e[i+1]*f[i+1])*

4.4 Towards vectorization

4.4.1 Important observations

In order to define the loop boundaries later this chapter, some observations need to be discussed.

To explain the first observation, let's review an example shown in Figure 4.6. The contents of register *shift (a)* depend on the content of registers *prev (a)* and *curr (a)*. And again, *prev (a)* depends on the *shift (prev (b))*, where *shift (prev (b))* depends on *prev (prev (b))* and *prev (b)*.

Note that in calculating *shift (a)*, only one element of *prev (a)* is used (marked in this example by *'xx'*). This value is the fourth element in the register of *prev (a)*. In order to calculate this value, only the last element in the shift register *shift (prev (b))* is needed. Now note that register *shift (prev (b))* depends on *prev (prev (b))*, while none of its values are used! Therefore, the vector *prev (prev (b))* is not relevant in calculating *shift (b)*. In this example, the shift-operations both shift

only by one element. But this is no different for other shift-operations with different parameters. Because, in order for the vector *prev (prev (b))* to contain useful information, this would require that the sum of the shift-parameters is larger than the vector length. As shifting more than the vector length makes no sense, this situation never occurs, and the observation remains valid. This same observation holds for *next (next (...))* as well.

Therefore we can conclude that:

(i) **The vector *prev (prev (...))*** is not used in calculating the vector *shift (...)*.

(ii) **The vector *next (next (...))*** is not used in calculating the vector *shift (...)*.

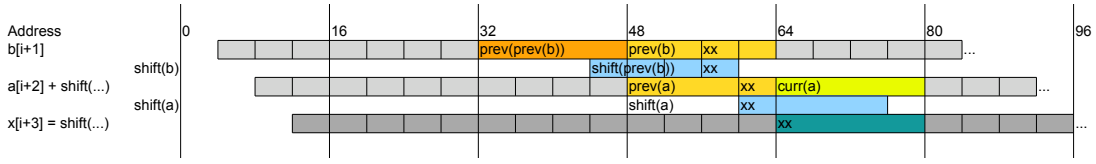


Figure 4.6: Two accumulated right shift-operations

4.4.2 Calculating Steady state loop boundaries; ProPeel and EpiPeel

As explained in Section 3.3.3, the whole stream is subdivided into four parts, a scalar prologue, vector prologue, steady-state and scalar epilogue. Calculating the precise bounds for these parts is important, as the goal is to achieve a maximum steady-state, while not loading and writing outside the bounds of the array as given by the program. We will begin by analyzing the 'range' of vector loads that is caused by the shift-configuration. This range influences the bounds of the vector prologue, steady state and epilogue.

The hierarchy of shifts resulting from some shift-configurations accumulates the 'boundaries' of the range of vectors that is calculated each iteration. For instance, the expression 'a[i+1] + b[i+2]', where the stream of b[i+2] is shifted to offset 1, and the stream a[i+1] + shifted(b[i+2]) is shifted to offset zero, contains two shifts to the left forming a hierarchy in vector operations. To calculate the current vector, this requires loading two vectors ahead (see the example in Section 4.4.1). Even though Section 4.4.1 tells us that next(next(...)) and prev(prev(...)) are not relevant in calculating the current vector, let's first concentrate on calculating these bounds.

To calculate the bounds of a statement, we will have to walk the expression tree of the statements right-hand-side in pre-order (i.e. the root is always visited first). We define for each node in the expression tree, a tuple *(Lbound, Ubound)* for Lower and Upper bound respectively. These values are initialized with (0, 0). For each node in the expression tree, if the node is annotated with a left shift operation, the Ubound is *incremented* by one. If the node is annotated with a right shift operation, the Lbound is *decremented* by one. The desired information is the minimum Lbound and the maximum Ubound found in the entire loop body, as these define the maximum range of vectors that need to be loaded in order to store a single vector. We will define the minimum Lbound as *ProPeel* (Equation 4.9) and the the maximum Ubound as *EpiPeel* (Equation 4.10).

Figure 4.7 shows the pseudo code of an algorithm that calculates these bounds.

```
int minlbound;
int maxrbound;

// find boundaries of expression
find_bounds(stmts:basicblock)
{
    minlbound = 0;
    maxrbound = 0;

    for(i=0; i<stmts.length; i++){
        find_bounds(stmts[i].RHS, 0, 0);
    }

    // truncate
    minlbound = max(minlbound, -1);
}
```

(a) Apply for each statement in loop body

```
find_bounds(root:node, lbound:int, rbound:int){
    lbound_local = lbound;
    rbound_local = rbound;

    if(is_left_shift(root))
        rbound_local += 1;
    else if(is_right_shift(root))
        lbound_local -= 1;

    maxrbound = max(maxrbound, rbound+1);
    minlbound = min(minlbound, lbound-1);

    // top down recursion
    find_bounds(root.left, lbound_local, rbound_local);
    find_bounds(root.right, lbound_local, rbound_local);
}
```

(b) Calculate ProPeel and EpiPeel recursively

Figure 4.7: Algorithm to calculate the boundaries of an expression

$$ProPeel = min(lbounds), \tag{4.9}$$
$$lbounds = \{Lbound(e) \quad | \quad subexpression(e, stmt), \forall stmt \in LoopBody\}$$
$$EpiPeel = max(ubounds), \tag{4.10}$$
$$ubounds = \{Ubound(e) \quad | \quad subexpression(e, stmt), \forall stmt \in LoopBody\}$$

Now that we have these bounds, we can look at truncating these bounds to minimize the required peeling factors. As discussed in Section 4.4.1, for expressions containing a hierarchy of right-shift operations, calculating the current vector requires loading at most one vector from a previous vector iteration. All the vectors before that point are irrelevant in calculating the current vector. Therefore, the left bound can be truncated to *(- Blocking Factor)*. For the right bound, this truncation cannot be performed, as for the software pipelined algorithm a 'previous' vector needs to be stored for each shift operation in the next iteration. For right-shift operations this only requires the previous vector from one vector back, but for left-shift operations this may require next iterations from several vectors in advance. Therefore, the final boundaries for the steady-state loop become:

$$ProPeel = max(ProPeel, -BF) \tag{4.11}$$
$$EpiPeel = EpiPeel \tag{4.12}$$

4.4.3 Adjusting vector offsets

Some vector architectures automatically perform address truncation to access address at aligned boundaries. As we can not assume address truncation in this thesis (the compiler framework used in this thesis, should be able to target *any* architecture), we will have to let the compiler insert code to truncate the addresses manually to create aligned memory streams. As the peeling factor has already been taken into account before the calculation of the shift operations, only truncating

the address calculation to an aligned boundary is required. The index of an address calculation like 'a[i+k]' thus becomes:

$$index(expr) = index(expr) - offset\_afterpeeling(expr) \tag{4.13}$$

4.4.4 Generating shift operations

After analyzing the expressions and finding shift configurations, the final thing to do is to actually generate the shift operations. The algorithm will have to be able to generate the shift operations in two ways, in a software pipelined fashion for the steady state loop, and a non-software pipelined fashion to initialize the steady state loop in a vector prologue. To simplify some terminology for the rest of this section, let's discuss the *previous*, *current* or *current*, *next* registers as the *old* and *new* registers respectively.

The generation of shift operations is done by walking the right-hand-side of each statement in the loop body, in pre-order, and generating code for each shift operation that is encountered. This is done recursively for each shift operation. When a stream-shift node is found, the algorithm will generate an expression for the vector-prologue, as well as for the steady-state. For the vector-prologue, the algorithm generates code to load both the *old* and the *new* vectors (i.e. Non-SP), and an additional shift-operation to return the desired result. For the steady state, the algorithm only generates code to load the *new* vector. It will then generate code to shift the already loaded *old* vector with the newly loaded *new* vector (i.e. SP). Finally, the *new* vector needs to be preserved for the next iteration by copying it into the *old* vector register. For the generation of the shift itself, the direction is important. When the found shift is a left shift, this means that the old iteration is loaded at *offset*, while the new iteration is loaded at *offset + VL*. When the found shift is a right-shift, the converse is true, as the old iteration is loaded at *offset - VL*, while the new iteration is loaded at *offset*. The algorithm to generate the code for shift instructions is shown in Figure 4.8.

As was explained in Section 4.4.1, the value *prev(prev(...))* is not required in calculating the current vector in the vector prologue loop. Therefore, instead of generating code to calculate that vector, one is free to choose another register for the shift operation. Being free to choose any register saves one or more registers and memory load operations. It also saves a number of iterations for peeling thus allowing a larger range for the steady-state loop (only when there is a hierarchy of shifts directed right). Note that this optimization can only be done for the vector prologue loop and will only have a noteworthy effect on performance for loops with a small number of iterations.

The mapping of shift-operations onto actual instructions is done in the code generation phase of the compiler. As code generation itself is implementation specific, it will not be discussed here, but in Chapter 5.

4.4.5 Runtime alignments

As has been previously mentioned in Constraint C1i, we require all parts of a memory address calculation to be constant (apart from the loop iteration counter). However, for expressions like *a[i+k]* where *k* is loop invariant but not a compile-time constant, this constraint could be considered too strict, as this disallows runtime misalignments. Section 3.3.1 mentioned that expressions with runtime misalignments can still be vectorized by using the zero-shift policy. This chapter has shown that only the direction of the shift (as opposed to the length of the shift) is important in determining the characteristics of the steady state loop, boundaries and the generation of the addresses of the load operations. For the zero-shift policy, the direction of the shift operations is always defined,

```
genSIMD_NOSP(root:Node, offset:int){
    if(contains_shift(root))    {
        shift = get_shift(root);

        if(shift.from > shift.to){ // shift left
            old = genSIMD(root, offset);
            new = genSIMD(root, offset + VL);
        }
        else if(shift.from < shift.to){  // shift right
            old = genSIMD(root, offset - VL);
            new = genSIMD(root, offset);
        }
        shift = genShift(old, new, offset);
        return shift;
    }else if(is_leaf(root)){// load operation
        return genLoad(root);
    }else if(is_OP(root)){  // other operation
        return genOp(root);
    }
}
```

(a) Non Software Pipelined

```
genSIMD_SP(root:Node, offset:int){
    if(contains_shift(root)){
        shift = get_shift(root);

        if(shift.from > shift.to){ // shift left
            new = genSIMD(root, offset + VL);
        }
        else if(shift.from < shift.to){  // shift right
            new = genSIMD(root, offset);
        }
        shift = genShift(old, new, offset);
        genStore(old, new);
        return shift;
    }else if(is_leaf(root)){// load operation
        return genLoad(root);
    }else if(is_OP(root)){  // other operation
        return genOp(root);
    }
}
```

(b) Software Pipelined

```
genSIMD(stmt:Statement, inPrologue:Bool){
    if(inPrologue)
        RHS = genSIMD_NOSP(stmt.RHS, 0);
    else
        RHS = genSIMD_SP(stmt.RHS, 0);
    genStore(stmt.LHS, RHS);
}
```

(c) Statement

Figure 4.8: Algorithm to generate SIMD from a single statement with shift operations

as all load operations are shifted left, and each store operation is shifted right. However, for our targeted platform no mapping is possible that maps shift operations with runtime shift-constants to actual instructions. As we will not implement any features that cannot be tested on our targeted platform, we will not consider runtime misalignments in the rest of this thesis.

Chapter 5

Implementation

This chapter will elaborate on the implementation details of the algorithms that were presented in the previous chapter. We will start in Section 5.1 by giving an overview of the CoSy compiler framework, and explain how we integrated our algorithms. Section 5.2 will present the implementation details of the analysis component. This will be followed in Section 5.3, by presenting the implementation details of the component responsible for performing the code transformations. Finally, Section 5.4 will explain how to map the shift-operations onto instructions for our targeted platform, and we will conclude in Section 5.5 with several anecdotes on difficulties that were encountered during this research.

5.1 The compiler framework

To explain the details of our implementation, we first need to give an overview of the CoSy compiler framework.

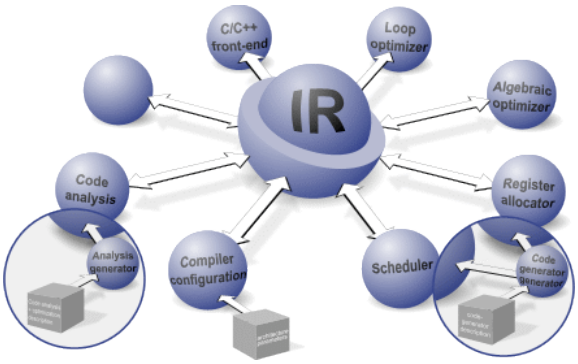


Figure 5.1: CoSy framework

5.1.1 General overview

The CoSy framework subdivides a compiler into several components, called engines. These engines perform analysis and transformations on the internal representation (IR) of the program. The IR used by the CoSy framework is a particular instance of a medium level intermediate representation (MIR), and is known as the CCMIR. As engines work on a single instance of the IR, engines can be easily 'plugged' into the compiler, depending on the transformations that are desired.

The engines follow a typical ordering, starting with a front end that transforms textual data (i.e. the program) into the CCMIR representation. This is followed by several high-level and architecture independent optimizations. These include for example engines for constant folding/propagation, if-conversion, function inlining and loop unrolling. When all high-level optimizations have been performed, the IR is *lowered* into an architecture dependent representation, known as the LIR. Finally, the back-end of the compiler will transform the exit-LIR into LIR, and finally into actual instructions for the targeted platform. Figure 5.2 shows the typical engine ordering of a compiler.

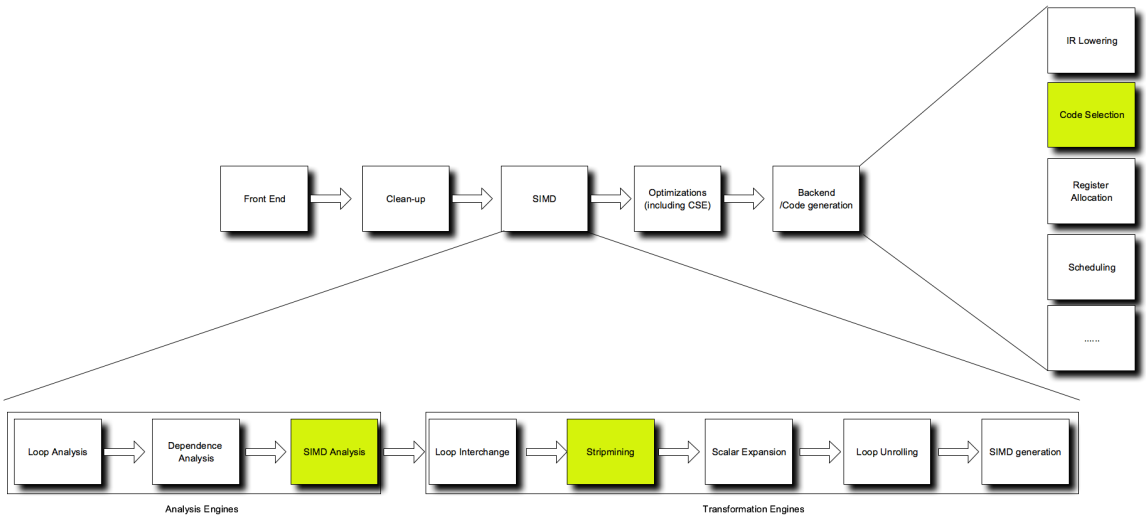


Figure 5.2: SIMD Engine Order

5.1.2 SIMD Optimization

For short vector architectures, the CoSy framework contains several engines that perform CCMIR-level analysis and transformations to optimize the program for vector execution. These engines are grouped together in a composite SIMD engine. The composite engine ordering can be roughly divided into two parts; The engines at the front analyze loop structures, annotating the loops that can be vectorized efficiently. The engines that follow transform the loop structure and generate vector operations.

The loop analysis engine is executed first. This engine regenerates loop information from the CCMIR by analyzing it's control flow graph, as the loop information is not retained by the compiler

front-end [1]. This is followed by a dependence analysis, which collects all (loop-carried) dependences for every loop structure. The SIMD analysis engine that follows uses this information to decide which loops are suitable for vector execution. Apart from dependences, loop characteristics such as memory access patterns, alignment issues and induction variables are taken into the consideration. When loops are annotated as vectorizable, a stripmining transformation is performed, which partitions the loop into vector-sized chunks (see Figure 5.4a). The stripmining transformation is followed by scalar expansion, which expands all loop invariant scalar values to a vector. Consecutively, a loop unrolling engine unrolls the stripmined loop, resulting in a series of operations that can be grouped together as vector operations. The aggregation is done by the SIMD generation engine, which aggregates groups of operations into virtual vector instructions. These vector instructions are virtual, as they are mapped onto actual instructions in the final phase of the compiler.

5.1.3 Integrating explicit realignment in CoSy

Our proposed optimization can be subdivided into four operations; imposing constraints on the input, determining a configuration of shifts, inserting code for loading the *old* and *new* vectors and the shift operation, and finally mapping shift operations onto instructions.

As the SIMD analysis engine already imposes most of the constraints from Section 4.1.1, it is convenient to add the additional constraints to this engine. Apart from imposing constraints, this engine also performs analysis on alignment and memory access schemes. As the functionality of our alignment analysis overlaps with the SIMD analysis engine, we agreed to combine our algorithms into this engine. When the alignment analysis fails due to alignment issues that cannot be resolved by explicit realignment, this will cause the entire SIMD analysis to fail, causing all subsequent SIMD engines to be skipped for the given loop.

Positioning the component that inserts the shift operations needs to be considered carefully. It is important to understand that a shift operation is performed on vectors, and that there is no semantical meaning of a shift operation on scalar values. Even though there is no semantical meaning of scalar shift operations, we did consider inserting 'scalar' shift operations on scalar elements, which can be grouped by a later engine as a vector shift operation on a vector register. However, doing this temporarily creates an invalid representation of the program, due to invalid dependences. These invalid dependences are a result from the fact that the shift operation extracts only a number of values from two vectors, to form a new vector.

To explain why this is an issue, Figure 5.3 shows an example code fragment in which the shift operation is inserted before the stripmining engine (and is thus performed on scalar elements). In statement *SHF1* a scalar shift operation is inserted, loading both the old and new element. Statements *UNR1-UNR4* show the statements after unrolling the stripmined loop. Statement *UNR1* implies that there is a dependence between the (scalar) element *a[i+0]* and elements *b[i+0] and b[i+BF+0]*, while this dependence is not true for the original loop. There is however a dependence between *a[i+0]* and *b[i+1]*, but this dependence is not represented by the expression! Pragmatically, one might not consider this an issue, as the scalar shift operations are mapped correctly onto a vector shift operation at the end of the composite SIMD engine. However, this would mean that the compiler temporarily has an incorrect and semantically different representation of the program.

[1] The actual reason for not retaining loop-information in the front-end is that C does not guarantee for/while-constructs to actually be loops, as a goto or break statement within a for/while-construct can result in non-looping behavior. Contrastingly, a sequence of goto-statements can form a loop without being explicitly defined as a loop. As the latter must be recognized as a loop and the former requires loop-analysis anyway, reconstructing loops becomes a good alternative to retaining loop-constructs by the front-end.

```
/* original loop */
for(i=0; i<N; i++)
ORG1:    a[i] = b[i+1];

/* after inserting shift */
for(i=0; i<N; i++)
SHF1:    a[i] = shift(b[i], b[i+BF], -1);

/* after stripmining */
for(i=0; i<N;     )
        for(j=0; j<BF; j++, i++)
        // STR1 represents nonexistent dependence between elements a[i] and b[i],b[i+BF]
STR1:           a[i] = shift(b[i], b[i+BF], -1);

/* after unrolling */
for(i=0; i<N; i+=4){
        // UNR1 does not represent dependence between elements a[i+0] and b[i+1]
UNR1:   a[i+0] = shift(b[i+0], b[i+BF+0], -1);
        // UNR2 does not represent dependence between elements a[i+1] and b[i+2]
UNR2:   a[i+1] = shift(b[i+1], b[i+BF+1], -1);
        // UNR3 does not represent dependence between elements a[i+2] and b[i+3]
UNR3:   a[i+2] = shift(b[i+2], b[i+BF+2], -1);
        // UNR4 does not represent dependence between elements a[i+3] and b[i+4]
UNR4:   a[i+3] = shift(b[i+3], b[i+BF+3], -1);
}
```

Figure 5.3: Inserting shift operations before stripmining is not allowed due to dependences

As it is possible to 'plug' engines into the compiler, it becomes possible that plugged-in engines will perform transformations that are not allowed. As this is totally unacceptable for a compiler, we quickly discarded this option.

Another option would be to add the shift operations after generating the virtual vector instructions (i.e. after the SIMD generation engine). There are several reasons why this option isn't attractive. First, code for the *prev* and *next* iteration will have to be generated. This is preferably done on the high-level IR, as this representation is easy to manipulate and review. These high-level expression trees are easier to manipulate than a series of statements of vector instructions (as the generated vector instructions is like 'virtual' assembly code). Another important aspect, is that the shift annotations are not yet supported by the rest of the compiler framework. As the SIMD engines transform the IR, the shift annotations should be kept in place until the SIMD engine has finished. This would mean adjusting each engine in the SIMD composite engine from the analysis engine up-to the SIMD generation engine.

Alternatively, we chose for a solution that is a compromise between the ideas presented above, and involves the insertion of shift operations right after the stripmining transformation. Performing the transformation at this point is allowed, because after a single stripmined iteration, all the operations are (element-wise) performed on vector registers, and it can safely be assumed that all the calculated values are in a vector register. To keep the distance between the analysis and alignment transformation as short as possible, we chose to combine our algorithm with the stripmining engine.

Finally, to map the virtual shift operations onto actual instructions, several matching rules need to be added to the code generator in the back end of the compiler.

5.2 Analysis engine

The alignment analysis closely follows the algorithms as presented in Chapter 4, and were implemented without raising any notable details. Therefore, we will not elaborate on the analysis engine any further.

5.3 Transformation engine

Before continuing on the details of inserting shift operations, let's begin by explaining the stripmining transformation, as this transformation precedes the insertion of shifts. The stripmining engine transforms the inner-loop of a loop-nest into a *stripmined* loop, by 'tiling' the loop into *strips* of $K * blockingfactor$ (also known as the *stripsize*). This is done by replacing the body of the inner-loop by a new loop, i.e. the *stripmine* loop, that ranges from 0 to the stripsize. The original body of the inner-loop is moved into the body of the new stripmine loop and the step size of the original inner-loop is set to the stripsize. The stripsize is the blocking factor of the smallest type found within the loop body, and can be defined as

$$stripsize = VL/min(sizes), \quad sizes = \{sizeof(x.LHS) \quad | \quad stmt(x), \forall x \in LoopBody\} \tag{5.1}$$

As the size of the original loop may not be a multiple of the blocking factor, the stripmining engine generates an epilogue loop, for the remaining iterations that are not serviced by the entire stripmined loop. The stripmining engine also calculates a peeling factor, to align one or more statements in the loop body. Figure 5.4a shows an example of a stripmined loop.

5.3.1 Implementing shift operations

Section 5.1 explained that the SIMD generation engine transforms a series of vectorizable expression trees into series of virtual vector instructions. These instructions are implemented as calls to Compiler Known Functions (CKF). These CKFs are mapped in the code generation phase onto actual instructions. We chose to take the same approach for implementing the vector shift operation, and created several compiler known functions that are used for shifting register streams. As function calls are natively supported by the framework, no adjustments to the IR or framework are required. The following part of this section will explain how to generate code for the *old* and *new* iterations, and how to insert the function call to the shift-CKF.

Creating registers

The reason for having a vector prologue is to initialize the *old* registers for the steady-state loop. The *old* registers are used in creating both the prologue and steady-state loop, and have a fixed mapping with respect to their corresponding shift operation. As they are used in creating both loops, this requires that they have to generated before performing the transformation.

For each shift annotation in the expression tree, a register is generated for the *old* iteration. A key-value map is then used to couple the shift operation with the generated *old* register. When generating code for the software pipelined algorithm, the compiler will look up the corresponding *old* register from the key-value map, and use it as the *old* value for the shift operation. For the vector prologue it is the other way around, as it will use the *old* register to hold the *new* value.

Loop splitting

When a stripmine loop has processed all it's iterations, it means that one vector iteration has been processed. As shifts operate on vectors, the operation itself will have to be outside the stripmine loop. But by placing the shift outside the stripmine loop, the result cannot be used within the loop itself. A solution to this problem is to split up the stripmine-loop every time a shift operation is encountered. Splitting up the stripmine loop into several loops is a valid transformation, as we assume that there are no hindering dependences within a distance of BF iterations, and we guarantee that the ordering of the statements is not changed by the transformation.

```
/*******************
 * Original loop
 *******************/
float a[SIZE], b[SIZE], c[SIZE];

for(i=start; i<end; i++){
  a[i] = b[i+1] + c[i+2];
}

/*********************************
 * After stripmining
 *********************************/
//prologue
for(i=start; i<propeel; i++){
  a[i] = b[i+1];
}

// steady-state
for(i=propeel; i<epipeel; )
{
   for(j=0; j<StripSize; j++, i++){
      a[i] = b[i+1] + c[i+2];
   }
}

// epilogue
for(i=epipeel; i<end; i++){
   a[i] = b[i+1];
}
```

(a) Stripmining transformation

```
/* Declarations of vector registers */
float bnew[BF], cnew[BF], bold[BF], cold[BF];
float bshiftres[BF], cshiftres[BF];

/***************************
 * After inserting the shift
 ***************************/
scalar_prologue();
vector_prologue(); // (initializes bold and cold)

// steady-state
for(i=propeel; i<epipeel; ){
   for(j=0; j<StripSize; j++, i++){
      bnew[j] = b[i+StripSize];
      cnew[j] = c[i+StripSize];
   }

   bshiftres = shift(bold, bnew, -4);
   cshiftres = shift(cold, cnew, -8);

   i = i-StripSize;
   for(j=0; j<StripSize; j++, i++){
      a[i] = bshiftres[j] + cshiftres[j];
      bold[j] = bnew[j];
      cold[j] = cnew[j];
   }
}

epilogue();
```

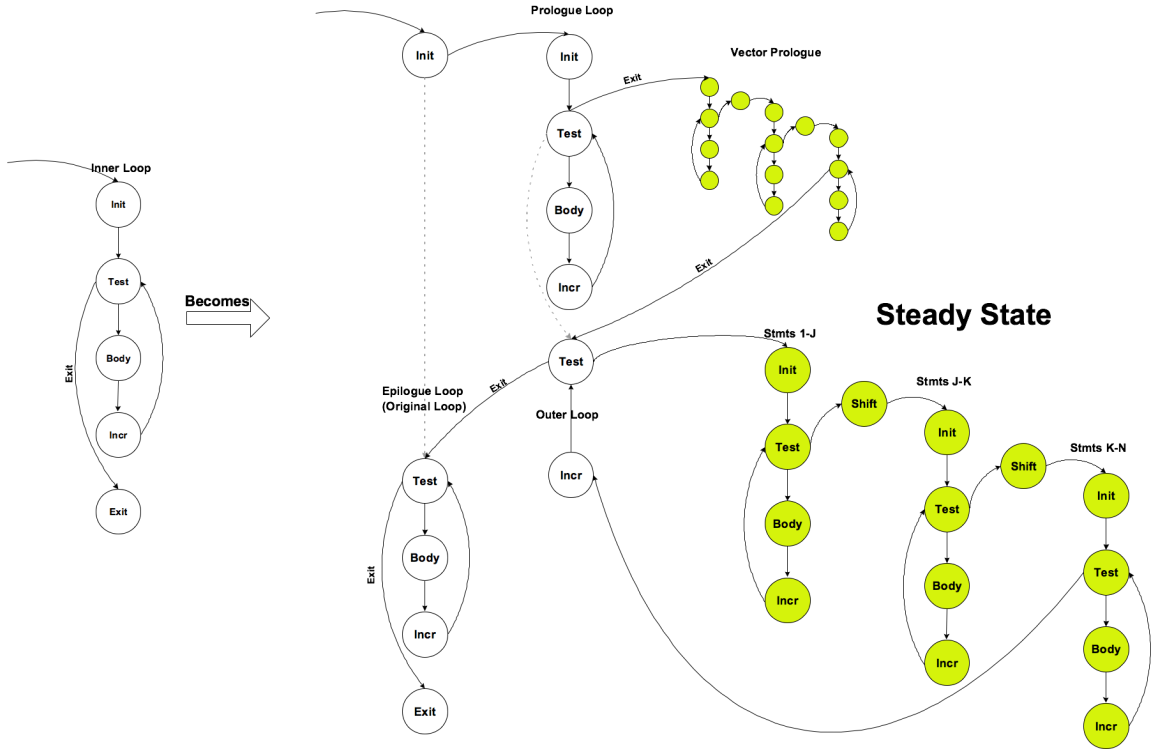
(b) Shift insertion

Figure 5.4: Example of stripmining transformation and shift insertion

The transformation follows the algorithm from Chapter 4, generating code for the *old* and *new* vector for each shift that is encountered. When both the *old* and *new* vector have been calculated, the shift operation itself must be inserted. As this must be done outside the loop, the loop is closed, and is followed by a statement that contains a function call to the corresponding shift-CKF. Finally, the original expression must substitute the shift-annotated expression with the result from the shift-CKF, and a new stripmine loop must be created to handle the remaining part of the statement and loop body that are currently being processed. As this process is performed recursively, loops may be split several times. Splitting the loop means closing the 'current' context (i.e. the stripmine loop), adjusting the (un)conditional jumps to exit-blocks, and finally creating the new context for the remaining loop body. This process involves quite some internal 'bookkeeping' to keep the control flow graph and jump targets correct.

Figure 5.4b shows a code-example of loop-splitting. The statement is annotated with two shift

operations to shift both b[i+1] and c[i+2] to offset 0. It first creates a statement to load the aligned *new* vector for *b*. As the shift is performed on vectors, the loop is closed, and a statement that shifts the stream using the *old* and *new* vectors of *b* is issued. A new stripmine loop is created, which stores the *new* vector of *b* for the next iteration. Also the *new* vector for *c* is loaded, and the process repeats itself. Finally, the unaligned memory references in the original expression are substituted by the shift-result, and the result of the addition is stored at a[i]. The accompanying control flow graph is shown in Figure 5.5. Here the colored nodes denote the basic blocks which operate on vectors, while the non-colored nodes denote the basic blocks operating on scalars.



(a) Stripmining transformation

Figure 5.5: Control flow graph of stripmined after inserting shift operations

Adjusting induction variables

As can be seen in Figure 5.4a, the iteration counter of the loop, *i*, has become a basic induction variable in the stripmine-loop. When splitting the loop into multiple stripmine-loops, this value will keep increasing several times during a single vector iteration, as the value of *i* is incremented each stripmine loop iteration. As a result, all subsequent stripmine loops that have been generated by loop splitting, will have an incorrect value for *i* (note that this also holds for all other induction variables in the original loop). Therefore, all induction variables that are incremented in the inner-loop should be adjusted by BF times the value of its increment expression before starting a new

stripmine loop. Figure 5.4b shows that the induction variable i is adjusted just before the new stripmine-loop with $i = i$ - BF.

5.3.2 Multiple statements in Loop Body

When the loop body consists of multiple statements with distinct offsets, several loop iterations are peeled off at the beginning of the loop, such that each vector store can safely start at offset 0. For the steady state loop, each store-offset is truncated to offset zero, such that only full vector stores are performed. When the steady-state loop ends, a scalar epilogue finishes the remaining iterations. But instead of continuing at offset 0 for each of the stores, each store will continue with it's original offset. When the original offset is greater than zero, the scalar epilogue will skip iterations for that statement. An example of this issue can be seen in Figure 5.6, where z[i+2] (with offset 1, after peeling) skips one scalar iteration. To resolve this issue, some iterations will need to be peeled off at the end as well. This number of iterations can be defined as the maximum store offset (in elements, not bytes) found in the loop body. The iterations are peeled off by subtracting this number from every induction variable in the loop body.

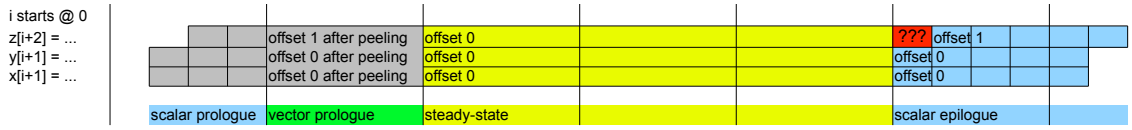


Figure 5.6: Peeling epilogue due to distinct store offsets

5.4 Code generation

The back-end of the compiler generates the actual instructions. As opposed to the sequential ordering of operations as presented earlier in the composite SIMD engine, code generation is performed iteratively. The intermediate results are refined each iteration until an optimal sequence of instructions is found. The code generator performs algorithms for pseudo register annotation, register allocation, code scheduling, pattern matching, and code selection. To target instructions, the CoSy framework requires a set of rules to map subexpressions onto instructions. By specifying a minimum set of rules, the compiler will be able to generate code for the given expressions. More advanced instructions can be targeted by adding more specific rules to the code generator. For each rule, a cost can be defined, such that the set of rules with lowest cost can be applied in order to generate the cheapest (set of) instructions for the given statement.

To implement the shift operation, we will need to add several rules to the code generator to match the compiler known function calls. As the semantics of a shift operations are not representable in the IR, we will manually need to generate our own instruction-mapping for the function calls. We will start this section by explaining the available instructions that can be used for shifting on our targeted platform, and we will end the chapter with the optimal shift mapping for each possible shift.

5.4.1 Permutation instructions

For our implementation we targeted the x86 instruction set architecture, with SSE and SSE2 extensions. SSE and SSE2 provide 8 vector registers with a length of 16 bytes, and provides instructions to operate on 8 short valued integers, 4 single precision floating point and double-word integer values, or 2 double precision floating point and quad-word values. Some permutation instructions are data type specific, requiring a different strategy for the different data types. Stream-shift operations can be implemented in a variety of ways with these different permutation instructions. This section is devoted to explain the available permutation instructions of our target architecture and their possible implementation of stream-shift operations.

All vectors: Shift and logical OR

One of the most generic ways to extract a misaligned vector from two vector registers, is by shifting each vector register to the right offset in the vector using a logical shift instruction, masking the values that are not used, and OR-ing both vectors together into a single vector. The *old* vector needs to be shifted left by *shift amount* bytes, while the *new* vector needs to be shifted right by *VL - shift amount* bytes. As an example, consider shifting a stream three elements to the left.

1. Input: $C = [C_0, C_1, C_2, C_3], \quad N = [N_0, N_1, N_2, N_3]$
2. Shift C left by 3 elements: $[C_3, --, --, --]$
3. Shift N right by 1 element: $[--, N_0, N_1, N_2]$
4. Result of OR-ing C and N: $[C_3, N_0, N_1, N_2]$

To shift an entire vector register one or more bytes to the left or right, the SSE2 extensions provide the *pslldq* and *psrldq* instructions respectively. These instructions shift a *double quadword* register value (i.e. a whole SSE register) one or more bytes left or right and zero's the empty high/low-order bytes.

Integer vectors: Unpack and shuffle

The SSE instruction set has separate shuffle instructions for integer and floating point vectors, both with different semantics. The shuffle instructions for integer vectors, *pshufw* and *pshufd* (for words and doublewords), can reorder the elements within a vector given any ordering, as long as the ordering-operand is an immediate (i.e. compile time) constant. Figure 5.7 shows the semantics of the operation.

SSE2's unpacking instructions can be used to interleave two vectors, as shown in Figure 5.8. There are unpacking instructions to interleave the lower half of two vectors (*punpckldq*, *punpcklwd*), and instructions to interleave the higher half of two vectors (*punpckhdq*, *punpckhwd*).

By combining the integer shuffle instructions with unpacking instructions, two vectors can be merged together with the desired ordering. First an unpacking operation can combine values from two vectors together, which are reordered in the right order by a shuffle operation. However, these instructions always take either the low part of both vectors, or the high part. For shifting a stream, it is required to take the high part of one vector, and the low part of the other vector. This could be done by issuing a shuffle instruction for one of the vector registers first. For shift operations

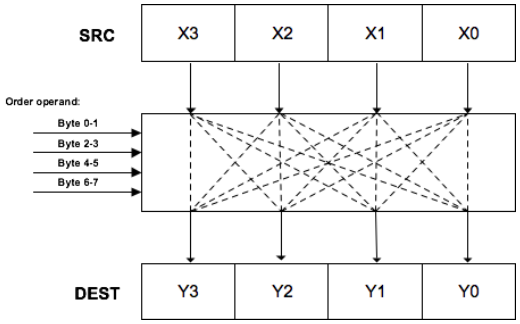


Figure 5.7: SSE2 Integer shuffle instruction where bytes 0-7 determine the ordering for the elements in SRC. The result is stored in DEST.

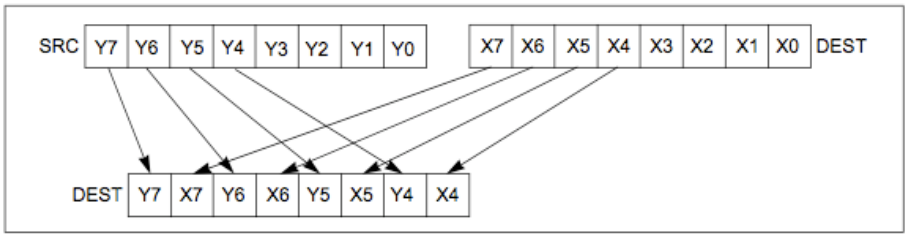


Figure 5.8: SSE2 Unpacking Instruction interleave elements of two vectors.

where the number of values required from one vector, is larger than the number of values required from the other vector, multiple passes will need to be issued.

To illustrate this with an example, consider again the case of shifting a vector stream 3 elements to the left.

1. Input: $C = [C_0, C_1, C_2, C_3], \quad N = [N_0, N_1, N_2, N_3]$
2. Unpack high C, N: T1 = $[C_2, N_2, C_3, N_3]$
3. Shuffle T1: T2 = $[C_3, N_2, C_2, N_3]$
4. Unpack low: T2,N: T3 = $[C_3, N_0, N_2, N_1]$
5. Shuffle T3: Result = $[C_3, N_0, N_1, N_2]$

As is intuitively clear from the above example, integer shuffle and unpacking operations are not very well suited for shifting register streams. Therefore, we will not consider them in the remaining part of this section.

Floating point vectors: Unpack and shuffle

Shuffle operations for floating point vectors have different semantics than those for integer vectors. Instead of reordering the data within a single vector, the floating point shuffle instruction allows the

possibility to select any *BF/2* values from one vector, and any *BF/2* values from another vector, and combine them (in fixed order) into the result vector. Figure 5.9 demonstrates the semantics of such an instruction. As this instruction can both combine and reorder values from two vectors, it has nice characteristics for shifting register streams.

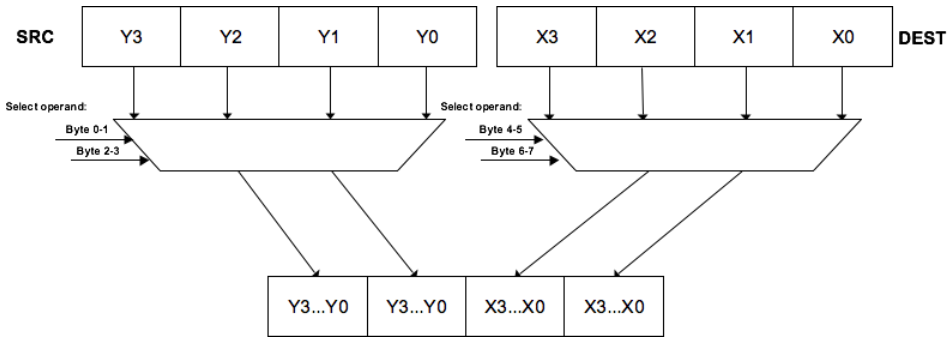


Figure 5.9: SSE2 Floating point shuffle instruction. Bytes 0-3 select two elements of SRC, while Bytes 4-7 select two elements of DEST. The resulting elements are combined in fixed order into the destination register.

Even though the floating point shuffle operations are meant to work on floating point vectors, in practice they can also operate on integer vectors. Effectively, only a reordering of the data is done, and no arithmetic is performed on the data. Therefore, only the size of the data elements is important, as opposed to the content of the registers. Before making this claim however, we checked that no floating point exceptions could occur due to specific values of the data. Even though the instruction set reference pointed out that no floating point exceptions are triggered by the instruction, executing several test-cases assured us that assigning an integer value denoting a 'signaling NaN' for example did not trigger any exceptions. The unpacking instructions are also available for floating point vectors, and operate similar to those for integer vectors. However, as the shuffle instruction already performs the task of 'combining' of vectors and the fact that unpacking operations are not well suited for shifting register streams, we will disregard the unpacking instructions in the remainder of this chapter.

Let's once again review the example of shifting a register stream to the left by 3 elements.

1. Input: $C = [C0, C1, C2, C3], \quad N = [N0, N1, N2, N3]$
2. Shuffle C, N: T1 $= [C1, C0, N1, N2]$ (Select C = {1, 0}, Select N = {1, 2})
3. Shuffle T1, N: Result $= [C0, N1, N2, N3]$ (Select T1 = {1, 2}, Select N = {2, 3})

As can be seen, using the floating point shuffle instructions requires only two operations, but can only be used for sizes equal to those of float and doubles.

Instruction latency

To be able to find a minimal mapping for shift operations to actual instructions, the latency of the permutation instructions need to be considered as well. The latency of the SSE instructions for our platform are obtained from [4]. Note that the given latencies are static estimates for the

latency and are based on the several assumptions. These assumptions include that the instruction has already been fetched and decoded, the memory operands are in the L1 data cache, and that there is no contention for execution resources or load-store unit resources.

Operation	**Instruction**	**Cycles**
Logical shift	pslldq,psrldq	2
Unpack int	punpckhdq	2
Unpack float	punpckhqdq	3
Unpack double	unpckhps	2
Unpack long long	unpckhpd	2
Shuffle int	pshufd	4
Shuffle float	shufps	4
Shuffle double	shufpd	4
Bitwise OR	por	2
Move register to register	movdqa/movaps/...	2

Figure 5.10: Latency table for SSE/SSE2 instructions that can be used for shift operations

The following section will give the optimal shift mappings that we used in our implementation.

5.4.2 Shift mappings

This section will present optimal mappings for the stream-shift operation. Note that we only have to create shift mappings based on the shift amount as opposed to the shift direction. This is because the shift operator itself is conceptually similar for both left and right shifts. Shifting left by K bytes is similar to shifting right by $VL - K$ bytes, with the distinction that a left shifts uses the *current* and *next* vectors, whereas a right shift uses the *previous* and *current* vector as input for the operation. However, selecting the input to the operator is not part of the operator definition.

As not all permutation instructions have similar latencies, we will need to consider the cost of individual instructions when calculating the cost of a shift operation. We will define the cost of a shift operator as the sum of latencies of its instructions. The dynamic cost of a series of instructions heavily depends on the pipeline characteristics. Therefore, using the sum of latencies is not an accurate representation for the actual cost. However, as the pipeline characteristics of our target platform are not publicly available, we will have to resort to static latencies in calculating the cost.

Optimal mapping for shifting left K bytes, where K is not a multiple of 4

For shift operations where the shift-operand is not a multiple of 4 (this is the case for shifting register streams containing elements of the *short integer* and *byte* data type), SSE's floating point shuffle instructions cannot be used, as they cannot operate on data types smaller than floats. As unpacking and integer shuffle instructions are not suited for the purpose of shifting streams (as this requires multiple passes), using a combination of logical shift instructions and an OR operation gives the best results, as this always requires a maximum of 4 instructions.

One instruction is needed to create a copy of the *new* vector. This is required because the logical shift instructions changes the contents of the *new* vector, while the contents of the vector

are needed for the next vector iteration. Then two instructions are needed to shift the *old* and *new* vector to the corresponding offsets. And finally, an OR operation should be used to combine the result. The estimated cost is 8 cycles.

Optimal mapping for shifting left 12 bytes

Shifting left by 12 bytes, can be done with a minimum of two instructions and a cost of 8 cycles. The combination of using logical shifts and a logical OR operation, costs 4 instructions and 8 cycles. The use of floating point shuffle instructions however, require only two shuffle instructions (also with a cost of 8 cycles). To compare the costs of two shift mappings with similar static costs (in latency) to see the influence of hardware pipeline characteristics, we measured the execution time of both mappings. This involved shifting using logical-shifts and an OR-instruction, and two shuffle instructions. The results did not show any significant difference in performance between the two mappings. As the *new* vector is not used as the destination register in any of the two instructions, it's contents are not changed and the vector doesn't need to be copied. The shift mapping with shuffle instructions is shown below.

1. Input: $C = [C0, C1, C2, C3], \quad N = [N0, N1, N2, N3]$
2. Shuffle C, N: T1 = $[C1, C0, N1, N2]$
3. Shuffle T1, N: Result = $[C0, N1, N2, N3]$

Optimal mapping for shifting left 8 bytes

Shifting 8 bytes to the left, requires only a single shuffle instruction. By using the *new* vector as source operand, and the *old* vector as the destination operand, the contents of the *new* vector are not altered thus not requiring a copy. By selecting the high-order bytes from the *old* vector, and the low-order bytes of the *new* vector, the data is combined into the vector using a single shuffle operation. The cost of this shuffle operation is only 4 cycles.

Optimal mapping for shifting left 4 bytes

When shifting left by 4 bytes, the use of the shuffle instruction is not optimal. Two shuffle instructions are required similar to shifting 12 bytes, but in this case the *new* vector is used as a destination, thus requiring an additional instruction to copy the *new* vector. This brings the total cost to 10 cycles as opposed to 8 cycles when using logical shifts and an additional OR operation.

5.5 Implementation issues

We will conclude this chapter with several issues that resulted from the presented implementation.

5.5.1 Multiple data types within statements

When statements with distinct data sizes exist within a loop body, the SIMD analysis engine selects the stripmining factor (i.e. the size of a stripmine loop) according to the smallest data size in the loop body. Our implementation splits up a loop when a stream-shift operation is encountered inside a statement. This loop splitting assumes the blocking factor to be equal to the stripmining size,

as it is assumed that a single vector iteration is performed after the stripmine loop. But when there are multiple data sizes in a loop body, this assumption is not valid. The stripmining factor is calculated as the blocking factor the smallest data size. Statements operating on larger data sizes, perform more than one vector iterations in the stripmine loop. This would mean that not one, but multiple shift operations are required.

As we were not able to implement a solution for this problem, we prevented this issue by adding constraint (C5) to the loop body. However, a possible solution could be to distribute the original loop into a series of loops with homogenous statements. This way, each loop has its own stripmining factor, causing the blocking factor for every statement in the loop to be equal to the stripmining factor. By considering less statements, it might improve the cache-hit ratio, as the cache is more dedicated to the lower number of expressions. Another possible solution is to leave the concept of loop splitting, and insert the shift operation after vector instructions have been generated. This way, issues with distinct data sizes will become solvable, but due to the nature of the SIMD engine it will prove to be quite the engineering challenge.

5.5.2 Common subexpressions

During implementation we came across an issue when considering statements where the left-hand side has no memory-offset. Figure 5.11 demonstrates the situation where an expression is moved to a temporary variable to avoid re-computation in multiple statements. When the left-hand side of the assignment is not a memory address it will also lack a memory-offset and the question what offset to chose for the right-hand side arises. One could choose the left-hand side to have a memory-offset of 0. However, as the chosen offset impacts the shift placement of other statements as well, selecting offset 0 may or may not be the best choice. Our proposed algorithm for calculating an optimal shift-configuration does not take common subexpressions into account ([11, 12] elaborates more on this subject). Even when a (convenient) offset has been chosen, the resulting variable will have a fixed offset in subsequent expressions, as it is not a memory stream that can be shifted to another offset because this would require a 'previous' or 'next' vector as well. A solution would be to generate three vectors (previous, current and next) for each statement where the left-hand side has no offset and substitute every use of that variable with an appropriate shift to a preferable offset using these vectors. In this way, the result of the statement will be transformed into a 'stream' as well, instead of a single value. However we were unable to implement this feature within the given timeframe.

```
for(i=0; i<N; i++)
{
    /* What offset to give tmp ? */
    tmp = a[i+1] + b[i+2] * c[i+3];

    x[i] = a[i] + tmp;
    y[i] = a[i+1] + tmp;
}
```

Figure 5.11: Left-hand of assignment statement side has no offset

5.5.3 Loop versioning

Some of the benchmarks contained runtime alignments due to multi-dimensional arrays with an inner-dimension that was not a multiple of the vector length. By versioning the loop using the iteration-variable of the outer-loop, the alignment of the inner-dimension can be determined as demonstrated in Figure 5.12. This is done using the modulo-operator, as the memory-offsets of the inner-dimension exhibit modulo-behavior. The CoSy framework offers value range analysis that calculates the possible values (i.e. ranges) a variable can have in any basic block. This knowledge can be used to determine information for many purposes like for instance determining reachability of code. This knowledge is also used in calculation of alignments and memory-offsets. Even though the value-range framework was quite elaborate, the modulo was still left for us to be implemented.

```
// inner dimension causes runtime alignments
for(i=0; i<N; i++)
    for(j=0; j<K; j++)
        in[i][j] = out[i][j];

// version the inner-dimension for compiletime alignments
for(i=0; i<N; i++){
    if(i % B == C){
        for(j=0; j<K; j++)
            in[i][j] = out[i][j];
    }else if(i % B == C+1){
        for(j=0; j<K; j++)
            in[i][j] = out[i][j];
    }else if(...){
        ...
    }...
}
```

Figure 5.12: Versioning

Value range analysis uses set-theory to represent ranges and perform calculations. For the statement as shown in Figure 5.12, the set of values resulting from the true-branch of the modulo operation, is a set of integer values with stride B, where C is one of the values in the set. By intersecting this set with the value-range of i, we achieve the possible set of values for i. For the else-branch of the statement, the value-range is the complemented set of the one calculated for i in the true-branch (see Equations 5.4, 5.2. Note that values B and C need to be positive constants in order to make assumptions about the alignments.

$$range_{mod,true}(\mathbb{Z}, B, C) = \{C - k * B, \ldots, C - B, C, C + B, \ldots, C + k * B\} \; where \; k \to \infty \tag{5.2}$$

$$range_{mod,true}(var, B, C) = range(var) \cap range_{mod,true}(\mathbb{Z}, B, C) \tag{5.3}$$

$$range_{mod,false}(var, B, C) = range^{C}_{true} \tag{5.4}$$

Chapter 6

Results

Benchmarking on our targeted platform proved to be quite a challenge. Instead of solely measuring the effect of explicit realignment alone, one also measures many side effects of the memory hierarchy, operating system and processor pipeline behavior. Looking 'inside' the processor is difficult, as cycle-accurate analysis tools for our architecture are not available. A tool by AMD called CodeAnalyst can gather statistics about an application like the number of cache misses, branch (mis-)predictions and executed instructions. However, the resolution of the supplied tools did not allow for assembly level analysis. Section 6.2 explains how we attempted to get reliable results.

It should be mentioned that explicit realignment is not required to vectorize loops with misalignments for our targeted architecture, as the SSE instruction-set provides move-instructions for unaligned data. Section 6.2 elaborates on the characteristics of unaligned instructions even though we do not aim at comparing unaligned move instructions with our explicit realignment solution. Our primary goal is to measure the speedup of vectorized applications that cannot be vectorized without explicit realignment and to investigate whether SIMD still benefits when permutation instructions are required to realign the data streams. Furthermore, we want to compare the performance differences of the proposed shift-placement algorithms. To place the results of the explicit realignment solution in context with unaligned instructions however, we included results of unaligned instructions as well.

This chapter will be structured as follows. Section 6.1 will elaborate on our targeted platform. Section 6.2 will explain some of the problems we encountered during benchmarking and will elaborate on effects that influenced our measurements. The results of the actual benchmarks are presented in Section 6.3.

6.1 Benchmarking platform

The system we used for benchmarking, consisted of an AMD Athlon X2 Dual Core processor, 4GB of main memory, running Red Had Linux. The processor has 512 KB of L2 shared data/instruction cache and 64KB of dedicated L1 data and instructino cach. The Athlon X2 is based on AMD's K8 architecture and supports MMX, SSE and SSE2 vector extensions. A noteworthy detail of this architecture is that the machine can only load 64 bits from the cache at a time. This means that 128 bit vector loads are actually composed of two 64 bit loads. As a result, vector loads themselves will become more dominant in vector expressions than on other architectures. Table 6.1 sums up

the specifications.

Processor	AMD Athlon X2 Dual Core Processor 5600+
Clock speed	2915 Mhz
Level 2 Cache	512KB
Level 1 Cache (data)	64KB
Level 1 Cache (instructions)	64KB
System Memory	4GB
Vector Extensions	SSE, SSE2, MMX
OS	Linux, kernel version 2.6.9-67.0.4

Table 6.1: Platform specifications

6.2 Benchmarking observations

To obtain reliable results, we executed several runs of every benchmark and took the shortest time as the result. This reduces the impact of context switches from the operating system and also ensures a warmed up cache for all except the first run. The linux 'time' function proved most accurate for the measurements. Even though the resolution of the timer function is not very high, it measures the program time disregarding any time taken by context switches to other operating system tasks. As the benchmarks were executed a large number of times, the resolution of the timer was not that important.

During the benchmarking process however, we noticed that the achieved results were somewhat uncorrelated. In some cases the results displayed a dramatic decrease in performance when using SIMD instructions, while in other cases the speedup increased with a factor that was higher than the blocking factor. These uncorrelated results are experienced throughout the entire benchmarking process, even for trivial programs containing a single loop without any misalignments. As we wanted to make sure we measured the effects of vectorization instead of irrelevant side-effects of our targeted architecture, we decided to investigate the issue.

Caching

When benchmarking on a platform like the AMD Athlon the caching mechanism plays a dominant role in the execution. As for this platform the caching mechanism cannot be switched off, it must be circumvented. This can be done by keeping the data small enough to fit in the L1 cache, and warming up the cache before use. However, even with these precautions a simple loop which adds two arrays without any misalignments, still causes cache-related problems. Figure 6.1 displays the measurements for both SIMD as well as scalar instructions for variable array sizes. When the array size is a multiple of 1024, a large number of cache misses occur and the performance drops significantly. Our hypothesis is that the mapping of memory address to cache line inhibits modulo behavior, where the last 10 bits of the address specify the cache line (this also corresponds to the 1024 number of cache lines of the K8 architecture). When accessing the arrays, a cache miss follows on each access, as the data from separate arrays is mapped to the same cache-line, causing the cache-line to be refilled each access. For SIMD execution the effects have less impact, as this only

happens for each vector access, instead of each element access. As a result, the measured speedups will approach the blocking factor. At first glance it will seem as though the achieved speedup approximates the estimated (and desired) speedup, but the results are not due to vector arithmetic but due to caching issues. To resolve this, the arrays must have different starting address which can be imposed by padding the power-of-two arrays with several elements.

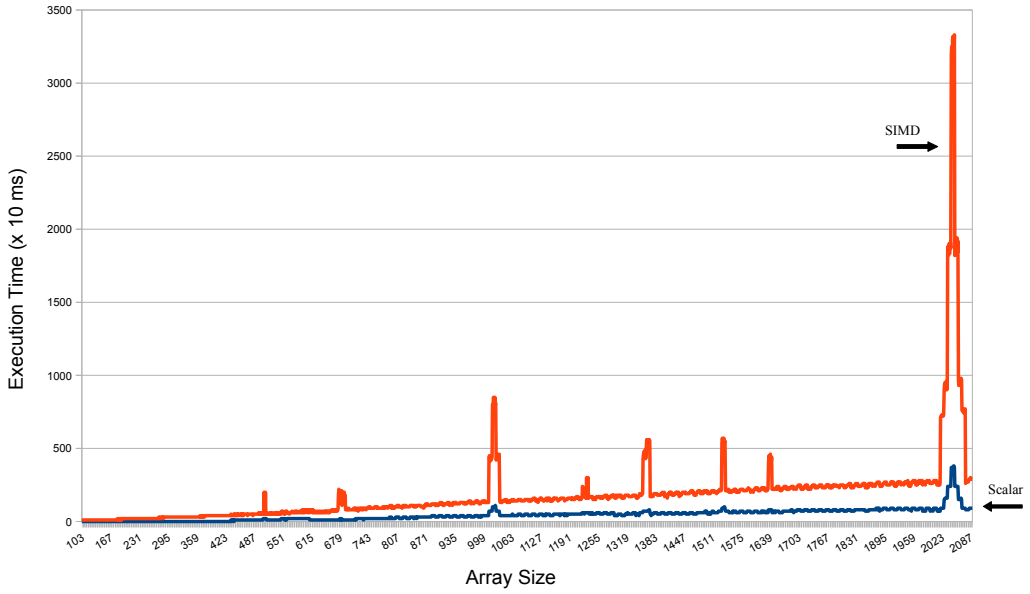


Figure 6.1: Address mapping causes cache-misses for array sizes that are a multiple of 1024

When the data fits only partially in the L1 cache, which is the case for many of the benchmark applications, the influence of the L2 cache and main memory may need to be taken into account as well. When this is the case, the program will spend more time in the memory hierarchy than it will spend performing arithmetic calculations. This will cause the memory to be the dominant factor in the calculation, therefore reducing the benefits obtained from vector arithmetic. [1]

Other compiler optimizations

Apart from runtime side-effects, we noticed that common subexpression elimination (CSE) complicates the results. When adjacent values are loaded from an array, the shift-configuration influences the behavior of CSE. To illustrate the issue, Figure 6.2 shows a code segment where neighboring values from an array are added together. In the steady-state loop, two vectors will be loaded on each vector-iteration. Because the memory-addresses are truncated to load at aligned boundaries, the same 'next'-vector will be loaded twice when the shift-operations are placed on the leafs (both *b[i+1]*, *b[i+2]* will be truncated to *b[i+0]*, loading the vector at *b[i+0]* as the 'current' vector and *b[i+4]* as the 'next' vector). This allows CSE to eliminate one of the load-operations. However,

[1] Our hypotheses on caching were backed-up by CodeAnalyst that can count the number of L1 and L2 cache misses for a given application.

when considering a shift-configuration where one shift is performed on one of the leafs and the other shift-operation is performed on the result of the addition, two different vectors will be loaded each iteration. Array-reference *b[i+2]* will cause the 'next'-vector at address *b[i+8]* to be loaded, while *b[i+1]* will cause the vector at *b[i+4]* to be loaded as the 'next'-vector, due to the hierarchy of shifts as described in Chapter 4. As both load-operations load their vectors at different addresses, CSE cannot eliminate one of the loads. This means that there is an additional penalty for shift-operations on the critical path when CSE is involved.

```
for(i=0; i<K; i++){
   x[i] = b[i+1] + b[i+2];
}
```

(a) Original loop

```
prologue(); //calculates b1curr, b2curr
for(i=0; i<K; i+=BL) // steady-state
{
   b1next = load_vector_at(b[i+BL]);
   b2next = load_vector_at(b[i+BL]); //CSE eliminated

   b1shift = shift(b1curr, b1next, -1);
   b2shift = shift(b2curr, b2next, -2);

   store_vector_at(x[i], b1shift + b2shift);

   b1curr = b1next;
   b2curr = b2next;
}
epilogue();
```

(b) Shift operations on leafs

```
prologue(); //calculates b1curr, b2curr
for(i=0; i<K; i+=BL) // steady-state
{
   b1next = load_vector_at(b[i+BL]);
   b2next = load_vector_at(b[i+2*BL]);

   b2shift = shift(b2curr, b2next, -1);
   b1next  = b1next + b2shift;

   store_vector_at(x[i], shift(b1curr, b1next -1));

   b1curr = b1next;
   b2curr = b2next;
}
epilogue();
```

(c) One shift operation on leaf, other on result of addition

Figure 6.2: Same number of stream-shifts but different configuration causes CSE to eliminate 1 load for (b), but no loads for (c)

Apart from CSE, we also tried a number of other optimizations including an optimization that removed the unnecessary 'i += 8; i -= 8' statement-pairs that resulted from our optimization (see Section 5.3.1). In some cases this caused the performance to decrease for reasons that we were unable to explain. Therefore, in the remainder of this chapter we take some precaution when trying to explain the results of our measurements.

Performance of SSE's unaligned instructions

In order to place the results of explicit realignment in context with SSE's unaligned instructions, we need to know properties of their behavior. We created a program that measures the differences in performance between unaligned and aligned instructions. By comparing the performance against the number of load operations, we can see if there is correlation between unaligned overhead and the number of vector loads. We initially expected the processor to perform a similar operation as with explicit realignment, by loading two vectors and extracting the unaligned vector, meanwhile buffering some of these vectors for subsequent iterations. However, the linear behavior shown in Figure 6.2 does not show any effect of buffering.

It does appear that the program with unaligned loads performs on average about 25% worse than it's aligned counterpart. As unaligned loads make up 50% of the instructions (i.e. the other 50% are aligned store instructions), we can conclude that unaligned loads are about 50% slower

than aligned loads.

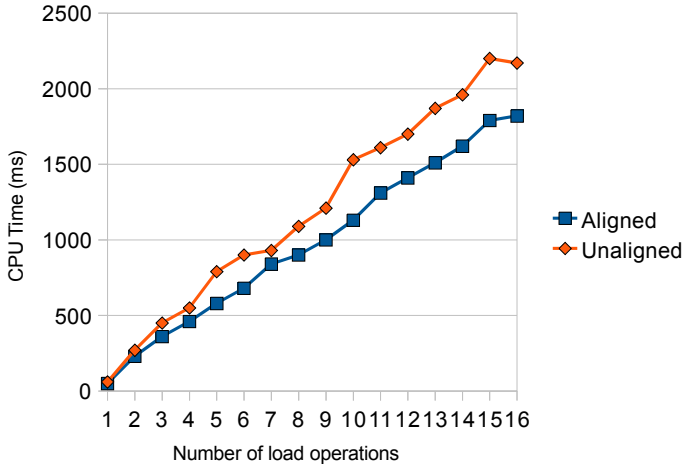


Figure 6.3: Results showing the overhead of unaligned- over aligned load instructions.

6.3 Benchmarks

This section elaborates on the benchmark results we acquired. Apart from showing the speedup achieved by explicit realignment we also specify which elements make up the execution time for some of the benchmarks. We subdivided the execution time into three elements; the number of aligned instructions and vector operations, the (explicit) shift overhead, and additional overhead caused by the realignment algorithm. The latter may be caused by loop peeling, vector prologue loop, loop versioning, register spilling (due to the additional registers required for explicit realignment) and saving of 'next'-vectors for subsequent vector iterations (for software-pipelining).

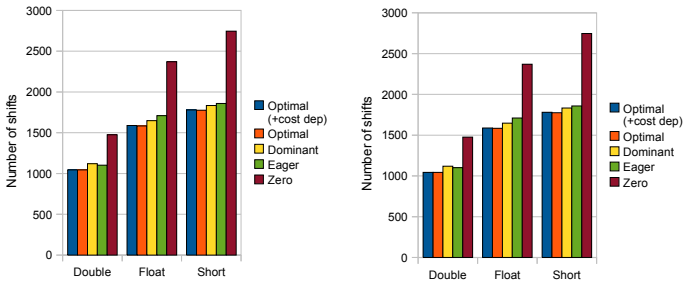
6.3.1 Generated loops

To examine the effects of the presented stream-shift placement methods, we created a suite of loop constructs from a loop-generator (see Appendix A.4 for an example). We gave this loop-generator a number of parameters to differentiate the data type, the number of array-references within a statement, the number of statements, and the choice for left-recursive or non-left-recursive expression trees. The latter choice is implemented by choosing the operation to be either addition or multiplication for the left-recursive trees, or a combination of the two for non-left-recursive trees. We decided to load the values from two separate arrays and storing the results in a third array. By operating on only three arrays, we enhance the spatial locality in memory, and circumvent the use of the cache hierarchy by making sure all three arrays fit in L1 data cache. We chose not to use the CSE optimization for these loops, as we want to compare the differences between the proposed shift-placement techniques without any side-effects.

The offsets of the array-references are chosen randomly using a normal distribution, where the mean of this distribution is chosen from a uniform distribution. For the variance we chose a sigma of

two, as this still allows shift-policies to benefit from loads at similar offsets. When choosing a sigma higher than two, the offsets are too far apart to gain advantage from lazy or dominant-shifting, while choosing a sigma of one on the other hand did not result in a significant number of dissimilar shift-configurations. Note that for larger data types, a lower number of possible offsets are available as different array-offsets are mapped to a smaller set of memory-offsets. For smaller data types this has the inverse effect, and memory-offsets will more often be further apart from each-other with a similar set of array-offsets. As a result, it becomes difficult to benefit from the locality in offsets by combining them with several stream-shift operations, causing a larger number of stream-shift operations to be required to realign the expressions. This causes the speedup to degrade and the benefit from clever shifting to drop.

Figure 6.5 shows the results from vectorizing the generated loops. As performance measure we chose to take the speedup from (explicitly) realigned vectors with respect to the unaligned vector speedup. By doing this, all possible side-effects from vectorization itself are hidden from the results, allowing us to compare the shift-placements amongst each-other instead of comparing vectorized and scalar code. Because the amount of generated loops are massive, the results show only the average speedups. When considering loops with multiple statements, the speedups from individual expressions are automatically averaged with the other statements in the loop body. For many expressions the calculated shift-configurations do not differ all too much. Therefore, the results of the proposed shift-placement techniques are rather similar, making it difficult to draw conclusions.



(a) Non left recursive expressions (b) Left recursive expressions

Figure 6.4: Number of stream-shift operations per type

We noticed that differences in performance are more apparent for left-recursive trees than for non-left-recursive trees. When examining the issue, we noticed that left-recursive trees are impacted more by the lazy-shift property. When an expression contains several array-references with similar offsets, the lazy-property will automatically group them when they are in the same subtree. For left-recursive trees, there is only a single subtree, while for non-left-recursive trees there is a larger 'variety' of subtrees. This causes the lazy-shift property to be more dominant in the shift-placement process for non-left-recursive trees.

Figure 6.4 does not show a large discrepancy in the amount of stream-shift operations between the proposed shift-placement methods. It shows that on average, the proposed dynamic programming algorithm results in the lowest number of stream-shift operations, followed by dominant, eager and finally the zero-shift placement methods. However, the number of shift-operations cannot be directly linked to the performance. Figure 6.5 shows that most cases, dominant shift performs poorly

when compared to eager shift, even though the dominant shift-policy generates less stream-shifts.

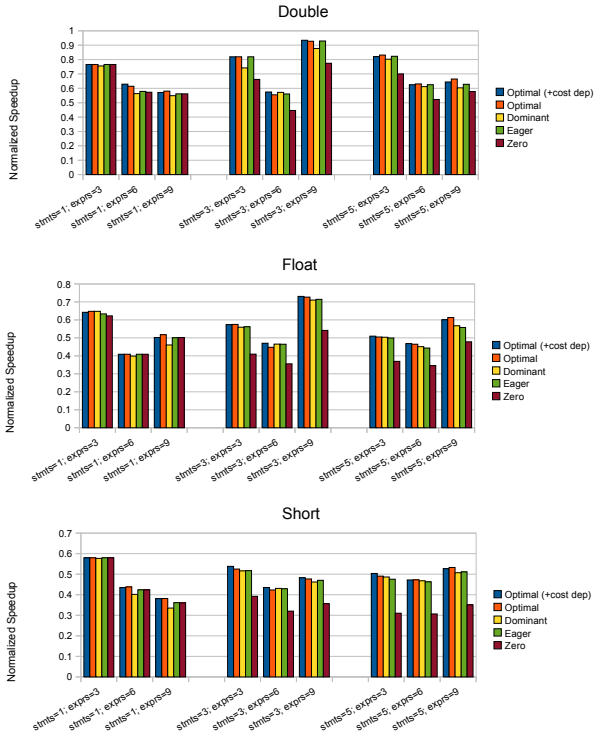
The results do indicate clearly that optimal shifting gives best results overall. Especially for large left-recursive expressions, we can see that the heuristics have more difficulty in choosing an optimal configuration and calculating an optimal configuration increases the average speedup about 10% with respect to the other shift-placement heuristics. For some loops, optimal shift-placement can even perform about twice as good, while for other loops the speedup may be a little worse, up to about 25% depending on the expression. Figure 6.5c illustrates the speedups of several individual loops. It is difficult to understand why our proposed 'optimal' solution does not always appear to be optimal. Maybe our cost-model is not sufficient enough to take support for latency-hiding into account. Or maybe the scheduler could be to blame for not scheduling the transformed code optimally. The compiler back-end is configured for a generic x86 processor where specific properties like pipelining characteristics, instruction latencies and other properties that define the instruction cost could be somewhat different from the AMD implementation we used. This may cause the code to be scheduled less optimal than expected.

Adding cost to prevent shift-operations on the critical path does not indicate a clear effect on the results. For some expressions the added cost benefit, while for other expressions it does not. We were however unable to find any correlation in the results for the expanded cost model.

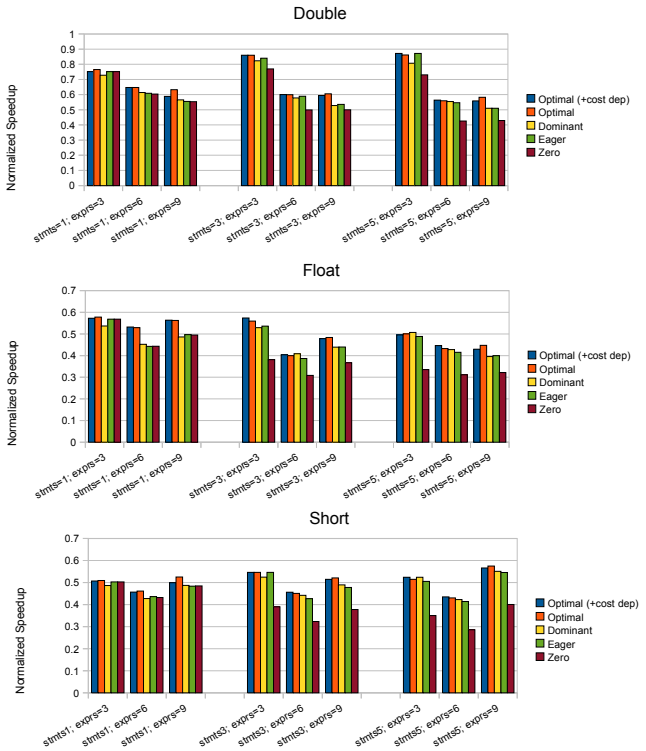
6.3.2 SPEC95 Tomcatv

The SPEC95 Tomcatv benchmark is a mesh generation program, that operates on a double precision floating point matrix of 513 by 513 elements. As the inner-dimension of the array is not a multiple of the blocking factor 2, the alignments are runtime. In order to be able to vectorize the benchmark, we first had to convert the source code from Fortran to C, as our compiler is configured with a C front-end. To deal with runtime alignments, we modified the algorithm slightly resulting in two versions. One version uses loop-versioning to select the vector loop optimized for the given misalignment. The other version uses multi-dimensional array padding, resulting in a matrix of 513x514 elements. The Tomcatv benchmark has only one suitable loop-candidate for vectorization. Other loops are hindered by dependences or involve reductions. The original algorithm stores common subexpressions into local variables, so they can be reused in other parts of the calculation. As common subexpressions are currently not yet supported for vector statements containing misalignments (see Section 5.5.2), we had to overcome this issue by storing the intermediate results in a temporary array. To reuse these values, they need to be reloaded into registers every time they are needed. Appendix A.2 displays both the modified and unmodified (original) loop constructs.

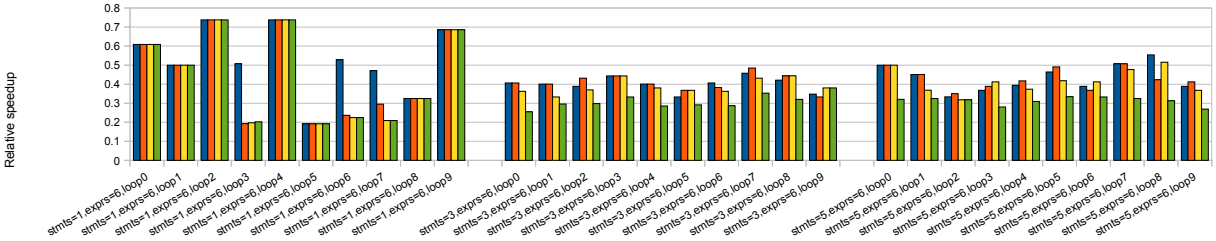
We compiled the benchmarks both with and without CSE. The results are depicted in Figure 6.6. One may notice that CSE makes a large differences for either scalar and vector instructions, which cannot be entirely contributed to our findings from Section 6.2. One may notice that there is a large distinction in performance between scalar and vector instructions when CSE is issued. The distinction is too large to be contributed to our findings from Section 6.2. The reason for this, is that for scalar code the back-end generates arithmetic instructions to calculate the addresses for the multi-dimensional array accesses, while for the vector code more elaborate addressing modes available for the architecture are used. CSE eliminates the intermediate results from the address calculations, significantly reducing the number of operations for scalar-execution. For vector execution, CSE cannot optimize the address calculation as it is delegated to hardware. As a result, CSE causes a significant speedup for scalar instructions, but not for vector instructions. This accounts for the 40% of performance difference.



(a) Average performance for 'normal' expression trees

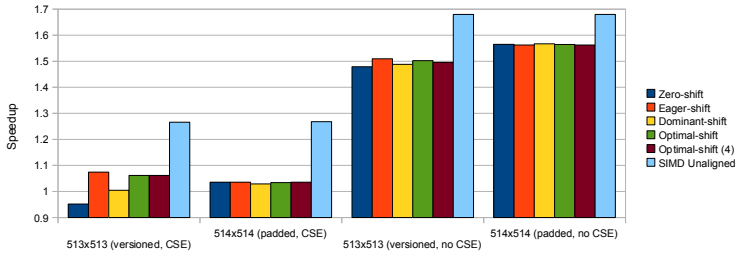


(b) Average performance for Left-recursive expression trees

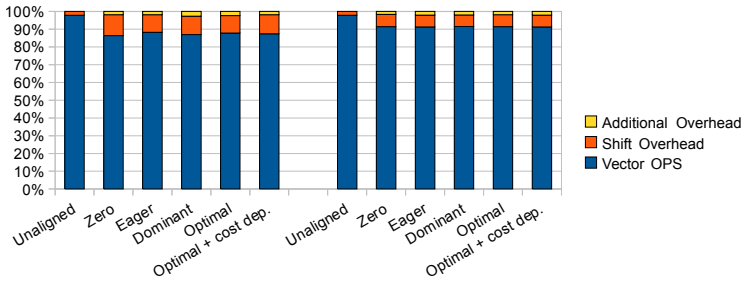


(c) A sample of results for loops of type float showing the variance in performance for some loops

Figure 6.5: Performance of generated loops. The results are normalized and display the speedup of explicitly realigned expressions relative to the speedup achieved by using unaligned instructions (i.e. realigned/unaligned, meaning that a result of 1.0 would indicate similar performance as unaligned instructions.



(a) SIMD speedup



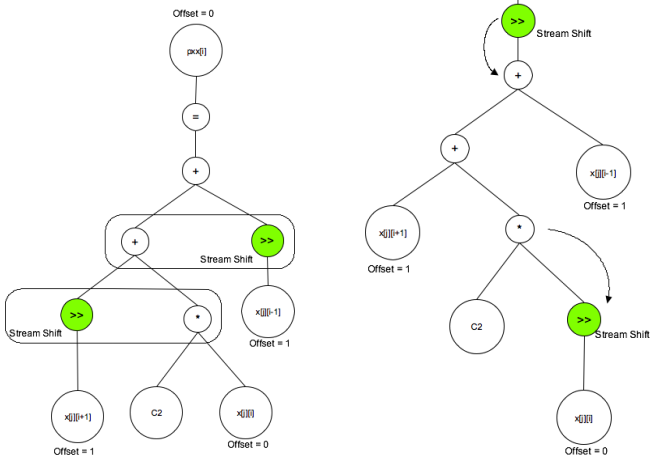
(b) Specification of results for versioned (left) and padded (right) versions (no CSE)

Figure 6.6: Benchmark results of Tomcatv

Compared to the eager-shift policy, we notice that the dominant shift policy performs poorly for the loop-versioned algorithm without CSE. After examination, we concluded that both shift policies create the same number of stream-shift operations with similar cost, but with different configurations. As mentioned in Section 4.3.5, the dominant shift policy places the stream-shift operations on the 'critical path' of several expression such that the latency of the operation cannot be scheduled together with another vector operation, while the eager shift policy places both stream-shift operations on the leaves of the expression. Figure 6.7 demonstrates the situation. This accounts for the performance difference of about 7% with CSE, and 3% without CSE between the two policies. For the padded algorithm, this difference does not exist, as the initial offsets of the memory references and therefore the shift configurations are different. Also note that the 5% benefit of padding becomes apparent when CSE is not used. When padding the arrays, no runtime checks for loop-versioning need to be performed and the offsets require less stream-shift operations. However, the impact of versioning is hardly noticeable as shown in Figure 6.6b, while the same figure displays that padding the arrays does reduce the number of shift-operations.

The intermediate arrays used to temporarily store the common subexpressions unnecessarily pollute the cache and thus increase the number of cache-misses for the actual data. This is shown as Additional overhead in Figure 6.6b. Due to the lack of support for common subexpressions, we currently cannot compare the results of using either registers and memory as placeholders for the intermediate results. On the other hand we should note that using registers as opposed to arrays causes register spilling as SSE only has 8 vector registers. However, this will hardly impact the cache, as the top values of the stack are accessed frequently enough to remain in the cache.

Overall we can conclude that the benchmark still benefits from explicit realignment even though



(a) Eager-shift policy, latencies can be combined with arithmetic operations

(b) Dominant-shift policy, shift operations on critical path

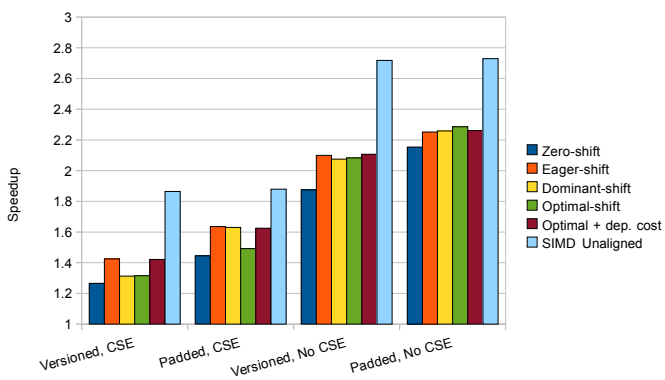
Figure 6.7: Shift configurations from eager and dominant shift

a 'trick' is needed to reuse common subexpressions.

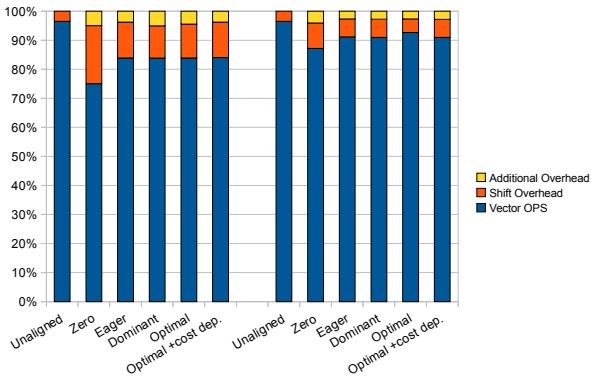
6.3.3 SPEC95 Swim

The SPEC95 Swim benchmark solves shallow water equations used in the prediction of weather conditions. The benchmark has 10 vectorizable loops, where 3 of them require realignment. Swim operates on single precision floating point matrices of 513x513 elements, resulting in a blocking factor of 4. Similarly to Tomcatv, we modified the algorithm to use either loop-versioning or multi-dimensional array padding to deal with the runtime alignments. The translated C code for the Swim benchmark can be found in Appendix A.1.

Figure 6.8 shows that the benchmark really profits from vector execution. Using our realignment technique, the achieved speedups range from 1.3 to 2.3 with and without CSE respectively. When CSE is not applied, the differences between shift policies are minimal varying 1 to 2 percent between each of the shift policies (except zero-shift). For most loops, the shift configurations are similar as the expressions are small in size. Only several statements have more complicated expression trees, resulting in different configurations. However there are too few of these statements to seriously impact the performance of the benchmark overall.



(a) SIMD Speedup



(b) Specification of results for versioned (left) and padded (right) versions (no CSE)

Figure 6.8: Benchmark results of Swim

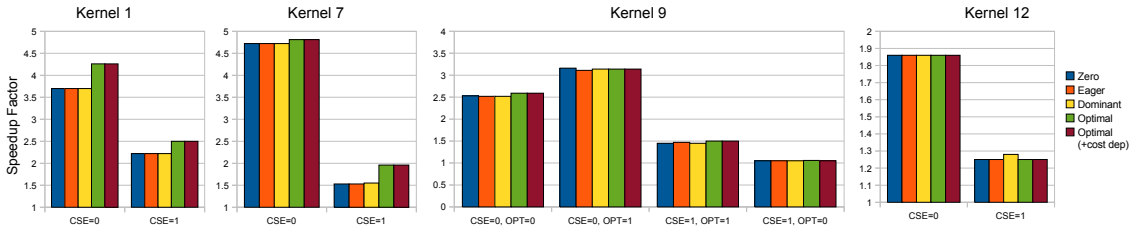
From Figure 6.8 it is evident that CSE complicates the results. For the Swim benchmark, we can see that our dynamic programming algorithm to find a shift-configuration achieves a similar performance to the zero-shift policy when CSE is applied. The benchmark contains a number of statements in which operations are performed on consecutive elements from the same array. Placing stream shift-operations on inner-nodes effects the ability of CSE to eliminate a number of load operations, which corresponds to exactly the same issue as explained in Section 6.2. This results in a performance-loss of about 10% to 12% for optimal-shifting.

When calculating the shift configuration the effect of CSE is not taken into account. Still, even with CSE, vectorization yields optimistic results. Padding the arrays gives an additional improvement of about 30% because vector loads at similar columns and distinct rows have similar offsets which reduces the number of misalignments within some of the statements. Furthermore,

Figure 6.8b shows that padding the arrays also reduces *additional overhead* due to the lack of loop-versioning. For Swim, loop-versioning plays a larger role than for Tomcatv, as the smaller element-size causes more *versions* to be generated.

6.3.4 Livermore kernels

The Livermore benchmark consists of 24 kernels that represent computations found in numerical applications. We were able to successfully vectorize 4 of these kernels, which all have (compile-time known) misalignments. The code for these kernels is given in Appendix A.3. The kernels exhibit a high vector-operation to memory-load ratio, as there are a lot of constant-multiplications in the expressions. Kernel 9 can be optimized by grouping together load operations with similar memory-offsets, resulting in an additional speedup of 50%. The results as shown in Figure 6.9 indicate that the different shift-placement methods do not differ that much. Only Kernel 1 seems to really profit from the dynamic programming algorithm. When looking more closely at the configurations, the 'optimal' configuration only differs from the heuristics in the fact that the shift-operations are placed on the loads, instead of placing it on the multiplication of that load with a scalar constant. We were however unable to explain the distinction in the results, as both configurations are very similar. Finally though, we can conclude that vectorizing with explicit realignment did prove to be profitable, as we were able to achieve speedup factors of 1.5x -4x depending on the kernel.



(a) SIMD speedup

Figure 6.9: Benchmark results of Livermore kernels

6.3.5 Motion estimation

Video encoders use motion estimation algorithms to determine the motion vectors describing a transformation from one 2 dimensional image to another. This is usually calculated between subsequent frames in a video. A motion vector is determined for each N x N sized block in the image. Several motion estimation algorithms are available, but for the purpose of automatic vectorization, we have chosen full-search block matching as a benchmark. The reference block is matched against a block in the subsequent frame for any offset in the horizontal and vertical direction by calculating the minimal sum of absolute differences. This is done for any possible offset in a given area around the block. The size of this area (kernel size) impacts the approximated movement of the object. The block size impacts the granularity of the objects for which the motion is approximated. Therefore, the choice of these parameters is important. The criteria we use to match the blocks is the sum of absolute differences. As this involves both a reduction and a type conversion, we split the loop into two loops. One loop calculates the differences, the other loop performs the reduction (sum).

The sum also requires a type-conversion from byte to integer. Note that for this benchmark we made sure the reference block is always aligned, as opposed to block which it is matched against, as this block is shifted several elements in both the horizontal and vertical direction. Figure 6.3.5 illustrates the procedure, whereas Appendix A.5 displays the code for the benchmark kernel. We tried different number of block sizes, as an increased size of the inner-loop reduces the overhead imposed by loop-peeling and loop-versioning. We combined the different block-sizes with two kernel sizes to scale the motion distances along with the block sizes.

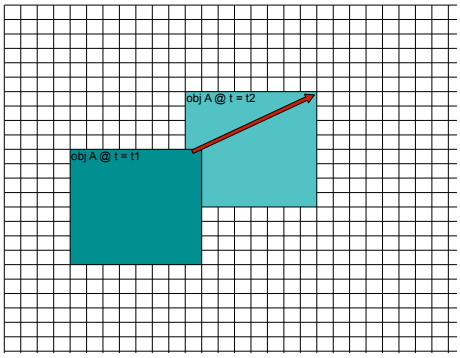
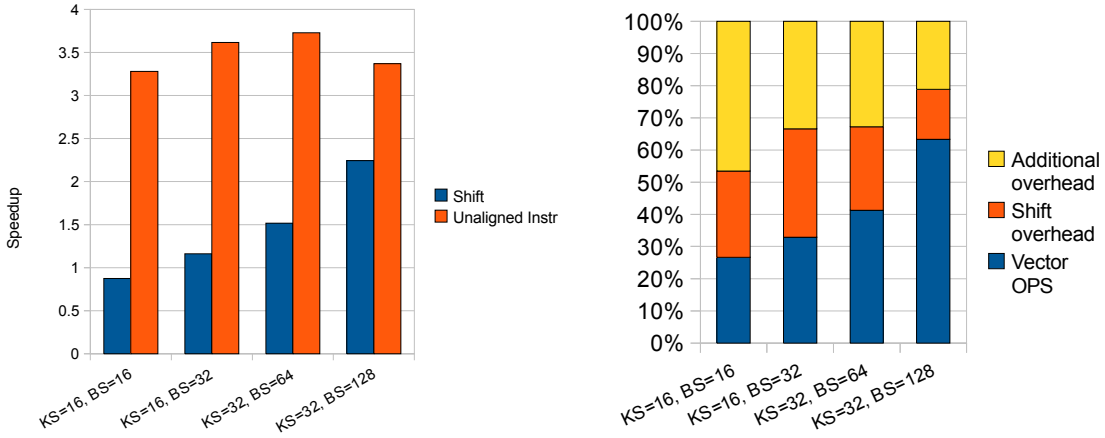


Figure 6.10: Full search block matching. Matching is performed by calculating the SAD between two blocks at different time intervals.



(a) Results of blockmatching algorithm

(b) Specification of results

Figure 6.11: Motion estimation results

The results of performing block-matching on a 1080x1920 pixel image are shown in Figure 6.11a. For this benchmark we have not made a distinction between the different shift-policies. The statement contains a single misalignment causing all shift-policies to generate the same configuration. For a block size of 16, no speedup can be obtained. At least 32 iterations are required before vector

instructions can be used, as the first 16 iterations will be peeled off in the prologue. This is due to the fact that 16 is also the blocking factor for the given data type, and at least one iteration will have to be loaded in advance, as explained in Section 3.3.3. It does however display the overhead imposed by loop versioning. By increasing the block size of the algorithm, more vector iterations are possible. We tried a number of block sizes ranging from 16 to 128 pixels, corresponding to 0 to 7 vector iterations respectively. The larger the block size, the more efficient and therefore beneficial SIMD execution becomes due to the reducing impact of loop-versioning overhead. This is clearly seen in Figure 6.11b where the ratio of 'additional overhead' is reduced and the ratio of vector operations increases. Finally, a speedup factor of 2.25x can be obtained with explicit realignment, as opposed to a factor 3.7 from unaligned move instructions.

6.3.6 Sobel filter

In image processing the Sobel filter is used to detect edges. The filter is a Finite Impulse Response filter which convolutes a block of filter coefficients with an image. Equations 6.1 and 6.2 show the filter coefficients for filtering edges in both the horizontal and vertical directions. Two images showing the effects of the Sobel operation are shown in Figure 6.12. Even though most images are represented as three single-byte values for the red, green and blue component, we implemented the benchmark to work on arrays of 2-byte values. We did this to avoid type conversion (to support larger data sizes) and because SSE does not support byte multiplication. The benchmark could be rewritten to work with single-byte values, but this does not add to the goal of benchmarking our optimization.

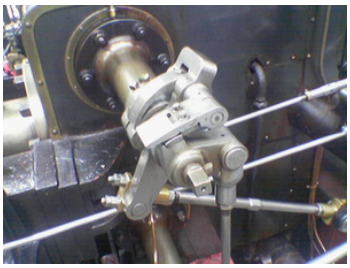
We applied the filter on two images, one image of 800x600 pixels with compile-time alignments and the other 801x600 with runtime alignments. We were able to optimize the expressions by grouping together similar offsets, enhancing the properties for realignment for the image with compile-time alignments. For the 801x600 image (i.e. with runtime misalignments) the number of distinct misalignments is larger than for the 800x600 image. As a result, this does not allow the possibility of grouping load-operations with similar offsets. With compile-time alignments, we are able to achieve a speedup factor of about 2x. As can be seen from the results in Figure 6.12, the zero and eager shift policies perform equal. This is because the LHS of the statement already has offset 0. Note that an additional speedup can be achieved by reusing values (or vectors) from subsequent kernel-iterations. Unrolling the kernel three times will expose loads at similar addresses, which can be removed by a CSE optimization. Also note that loop-versioning has a large impact on the results, accounting for about 15%-20% of the total performance (see Figure 6.12b).

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * image \tag{6.1}$$

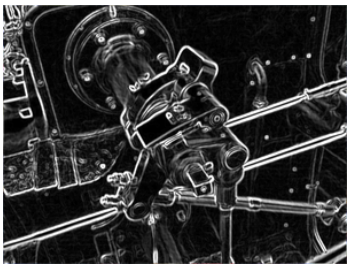
$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * image \tag{6.2}$$

6.3.7 Mediabench GSM

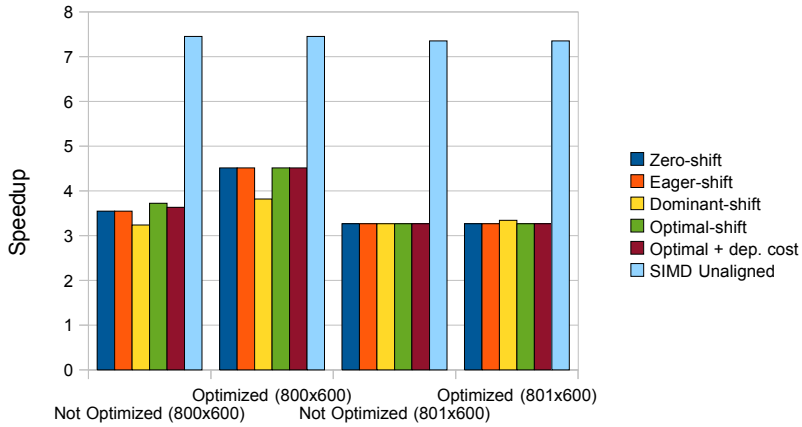
The encoder in the Mediabench GSM benchmark has one loop that can potentially be vectorized. The loop and it's transformed version for SIMD are shown in Figure 6.13. The operations are



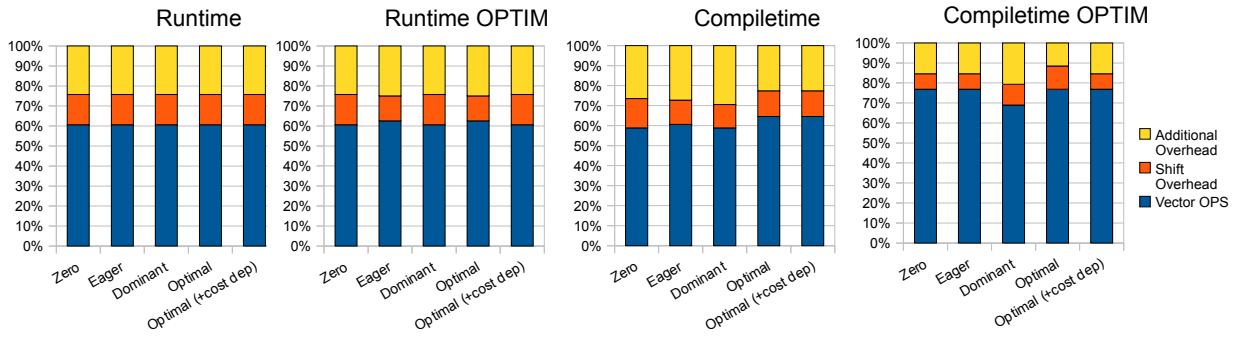
(a) Original image



(b) Filtered



(a) Benchmark results from Sobel



(b) Specifications of results

Figure 6.12: Results of Sobel FIR filter

performed on 16-bit data types. However, the code shows that the loop is only 8 iterations in length, and a dependence between intermediate results every 8 iterations does not allow a transformation that results in more loop-iterations and therefore more vector iterations. Performing only 8 iterations on 16-bit data types, is equal to a single vector operation on a 128-bit vector. However, these 8 iterations will be 'peeled' by the algorithm because loading vectors outside the bounds of the arrays is not allowed. At least 16 iterations will be required before vector arithmetic can be performed. Therefore the results are only impacted by the overhead from peeling and suffer a performance decrease of about 5%-10%. As no actual vector-instructions are executed, we do not explicitly show the results here.

```
// Short term analysis filter for GSM */
void filter(LARp,s,sav,d,u,k_n)
{
  // note dependence between:
  // d <-> u <-> sav, of length 8

  while(k_n--)
  {
    sav[0] = *s;
    d[0] = *s;

#ifdef SIMD
    simd_kernel(LARp, d, sav, u);
#else
    scalar_kernel(LARp, d, sav, u);

    for(i=0; i<8; i++)
    {
      u[i] = sav[i];
    }

    *s++ = d[8];
  }
}
```

(a)

```
void scalar_kernel(LARp,d,sav,u){
  for(i=0; i<8; i++) {
    tmp1 = (LARp[i]*d[i] + ROUND)>>15;
    sav[i+1] = clip16(u[i] + tmp1);

    tmp2 = (LARp[i]*u[i] + ROUND)>>15;
    d[i+1] = clip16(d[i] + tmp2);
  }
}

void simd_kernel(LARp, d, sav, u){
  /* Loop distributaion applied
   * to remove dependence in loop
   * '(...+ROUND)>>15' can be
   * performed by mulhw-instruction */

  // not vectorizable, true dependence
  for(i=0; i<8; i++)
    d[i+1] = d[i] + LARp[i]*u[i];

  // vectorizable with misalignment
  for(i=0; i<8; i++)
    sav[i+1] = u[i] + LARp[i]*d[i];
}
```

(b)

Figure 6.13: GSM Benchmark, Short term filter contains vectorizable kernel

Chapter 7

Conclusions

Explicit realignment allows loops with misaligned memory accesses to be vectorized when loop peeling, data duplication and multi-dimensional array padding are insufficient to resolve the alignment-issues. Following from Chapter 6, we can conclude that explicit realignment remains beneficial even though some overhead is produced from inserting shift operations to realign the data streams. We should note that obtaining reliable results was extremely difficult given our benchmarking platform as the caching mechanism affected the results. Even though we tried to minimize these influences, we made some observations we were unable to explain. Unfortunately, no tools were available that display the inner-workings of our target processor in sufficient detail that could help us find all explanations.

Even though explicit realignment allows loops with alignment-issues to be vectorized, there are several things that need to be taken into account before rigorously applying the transformation. First, the number of loop iterations largely determine the usefulness of the transformation, as two prologue loops (one scalar, one vector) and a (scalar) epilogue are generated. The shift-configuration determines the number of loops that are to be peeled off, as explained in Chapter 4. For any vector operations to be performed, the number of loop iterations must exceed the number of iterations peeled off in the prologue and epilogue. Second, the shift-operations needed to realign an expression impose overhead. We have not been able to find any expressions where this overhead dominates the speedup from SIMD instructions when all but the zero-shift policy was used for determining the shift-configuration. However, we should note that this depends on both the array-indices as well as the the data size. For large data types, array-offsets are mapped to a smaller set of memory-offsets. This allows the shift-placement algorithms to more easily create combinations of loads with similar offsets. Smaller data types may require more shift-operations as the set of possible memory-offsets is larger. On the other hand, more data can be processed with a single vector operation which again reduces the impact of a shift-operation. Even though we have not found any expressions where the overhead from shift operations rendered vectorization useless, it is important to understand that this largely depends on the target architecture and the available instructions for shifting register streams. While researching the capabilities of other compilers as well, we observed that automatically vectorizing applications often proves unprofitable due to the large number of expensive transformations involved to support short vector operations. Therefore, we believe it is important to accurately determine the cost of vectorization before performing all the transformations.

We presented four shift-placement methods in this thesis. Chapter 6 pointed out that the zero-shift policy produces most overhead, sometimes resulting in worse performance than when scalar instructions are used. We were unable to implement zero-shifting for runtime alignments, as these can not be mapped to vector instructions due to limitations of our target architecture. However, when they are supported by the architecture, vectorization may not be profitable. Overall, eager- and dominant-shifting performed more or less similar to each other. Depending on the expression, one may be more advantageous than the other, but the results did not indicate a clear winner. We were unable to determine exactly which expressions performed better with eager or dominant shift. Our other shift-placement method that is supposed to provide an optimal solution gave the best results on average. There were however some expressions where the optimal solution performed worse than the configurations as calculated by the eager or dominant shift placement policies. We assume that this is due to the disability of the algorithm to take latency-hiding into account. Stream-shift operations can be scheduled to be performed in parallel with arithmetic or memory operations. When there are data dependences between two subsequent shift-operations, this delay cannot be masked. Adding additional costs to model this dependence did not prove to be sufficient, as the overall performance was no better than without these costs. We should note however that some expressions do benefit from the expanded cost-model, but it may also have an opposite effect on other expressions.

Finally, Chapter 6 leads us to conclude that the performance of the distinct shifting-configurations are impacted by common subexpression elimination. This causes for example 'optimal' shift-configurations to be far from optimal in some scenarios. CSE is an important optimization that has a large impact on the performance of shift-configurations. We can therefore not ignore the effects of CSE on the placement of stream shift-operations. We therefore believe more research is needed to coordinate the functionality of both transformations.

Chapter 8

Future Work

We would like to conclude our findings with some remarks on possible future work on the field of automatic vectorization.

Apart from alignment issues, other issues like inductions, reductions, length conversion, non-unit stride access, conditional statements and dependences are often encountered in applications. Quite some research has been performed on these topics, as has been explained in Chapter 2. Incorporating optimizations to (partly) remove these issues could have a large impact on the number of applications that can be vectorized. However, all these optimizations may cause too much overhead to render the use of short vector instructions to be useful. Therefore we propose to incorporate a good approximation of the actual cost before applying transformations. Choosing a good configuration for the compiler is known to be a difficult problem. [13] proposes a technique that extracts certain features from the program's IR, and looks up the best compiler configuration from a database, which has been trained by a large number of applications. By choosing SIMD-specific features, this technique might be applied for auto-vectorizing transformations as well.

We concluded that CSE has a significant impact on the chosen shift-configurations. We were however unable to understand the exact relationship between the two optimizations, and therefore unable to come up with a more suitable algorithm to place stream-shift operations. We therefore believe more research on the interaction between SIMD and 'standard' compiler optimizations needs to be done in order to understand their relation.

Finally, we should note that this research has focussed on improving execution times. Unaligned instructions may prove to be more efficient than the proposed realignment solution in terms of execution time, but this may be different in terms of power. Unaligned instructions require a permutation unit in hardware to extract aligned vectors. This additional unit requires power even when it is not used. For embedded devices power is an important criterium, which may be a good reason to use the realignment solution instead of a specific unaligned load unit. Currently, we cannot make any conclusions about this trade-off, but it may be an interesting topic to elaborate on in future research.

Bibliography

[1] *Intel 64 and IA-32 Architectures Software Developer's manual. Volume 1: Basic Architecture.*

[2] J. Abel. Applications tuning for streaming simd extensions. 1999.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures - A Dependence based approach.* Morgan Kaufmann, 2001.

[4] AMD. *Software optimization Guide for AMD64 Processors.*

[5] A. J. Bik. *The Software Vectorization Handbook.* Intel Press, 2004.

[6] D. Blythe. The direct3d 10 system. 2006.

[7] H. Chang. Efficient vectorization of simd programs with non-aligned and irregular data access hardware. *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, 2008.

[8] A. Darte and Y. Robert. On the alignment problem. 2007.

[9] A. E. Eichenberger. Vectorization for simd architectures with alignment constraints. *Conference on Programming Language Design and Implementation*, 2004.

[10] A. E. Eichenberger. Efficient simd code generation for runtime alignment and length conversion. *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[11] L. Fireman. The complexity of simd alignment. 2007.

[12] L. Fireman. New algorithms for simd alignment. *Lecture Notes in Computer Science*, 2007.

[13] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, June 2008.

[14] M. Kirsten. Loop nest optimization for simd architectures.

[15] B. D. Koblenz. Constraint based vectorization. *Proceedings of the 3rd international conference on Supercomputing*, 1989.

[16] A. Kudriatsev. Generation of permutations for simd processors. *LCTES*, 2005.

[17] S. Larsen. Exploiting superword level parallelism with multimedia instruction sets. *Conference on Programming Language Design and Implementation*, 2000.

[18] C. Loeffler. Practical fast 1-d dct algorithms with 11 multiplications. 1989.

[19] D. Nuzman. Auto-vectorization of interleaved data for simd. *PLDI*, 2006.

[20] D. Nuzman. Outer-loop vectorization - revisited for short simd architectures. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[21] M. Phar, E. Kilgariff, and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[22] A. Shahbahrami. Performance impact of misaligned accesses in simd extensions. *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing*, 2006.

[23] A. Shahbahrami. *Avoiding Conversion and Rearrangement Overhead in SIMD Architectures*. TU Delft, 2008.

[24] J. Shin. Superword-level parallelism in the presence of control flow. *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[25] C. Tenllado. Improving superword level parallelism support in modern compilers. *CODES+ISSS'05*, 2005.

[26] P. Wu. An integrated simdization framework using virtual vectors. *Proceedings of the 19th annual international conference on Supercomputing*, 2005.

Appendix A

Benchmarking source code

A.1 Spec95.Swim

```
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>
#include <math.h>

#define N1  513
#define N2  513

#define m   (N1-2)
#define n   (N2-2)
#define min(a,b)  (a < b ? a : b)
#define abs(a)    (a < 0 ? (-a) : a)
float u[N2][N1];
float v[N2][N1];
float p[N2][N1];

float unew[N2][N1];
float vnew[N2][N1];
float pnew[N2][N1];

float uold[N2][N1];
float vold[N2][N1];
float pold[N2][N1];

float cu[N2][N1];
float cv[N2][N1];
float z[N2][N1];
float h[N2][N1];
float psi[N2][N1];

float    dt, tdt, dx, dy, a, alpha;
float el, pi, tpi, di, dj, pcf;
int   itmax, mprint, mp1, np1;

void initial(void)
{
   int i, j;
   /* IO Read */
   dt = 20.0f;
   dx = 0.25E5f;
   dy = 0.25E5f;
   a = 1.6E6f;
   alpha = 0.001f;
```

```
    itmax = 1200;
    mprint = 1200;
    tdt = dt;
    mp1 = m+1;
    np1 = n+1;
    el = (float) n * dx;
    pi = 4.0f * atanf(1.0f);
    tpi = pi+pi;
    di = tpi/(float)m;
    dj = tpi/(float)n;
    pcf = (pi*pi*a*a) / (el*el);

/* 50 */
    for(j=1; j<=np1; j++){
        for(i=1; i<=mp1; i++){
            psi[j][i] = a * sinf(((float)i-0.5f)*di)
                      * sinf(((float)j-0.5f)*dj);
            p[j][i] = pcf*(cosf(2.0f*(float)(i-1)*di)
                    + cosf(2.0f*(float)(j-1)*dj)) + 50000;
        }
    }

/* 60 */
    for(j=1; j<=n; j++){
        if(j%4 == 0){
            for(i=1; i<=m; i++){
                u[j][i+1] = -(psi[j+1][i+1] - psi[j][i+1])/dy;
                v[j+1][i] = (psi[j+1][i+1]-psi[j+1][i])/dx;
            }
        }else if(j%4 == 1){
            for(i=1; i<=m; i++){
                u[j][i+1] = -(psi[j+1][i+1] - psi[j][i+1])/dy;
                v[j+1][i] = (psi[j+1][i+1]-psi[j+1][i])/dx;
            }
        }else if(j%4 == 2){
            for(i=1; i<=m; i++){
                u[j][i+1] = -(psi[j+1][i+1] - psi[j][i+1])/dy;
                v[j+1][i] = (psi[j+1][i+1]-psi[j+1][i])/dx;
            }
        }else if(j%4 == 3){
            for(i=1; i<=m; i++){
                u[j][i+1] = -(psi[j+1][i+1] - psi[j][i+1])/dy;
                v[j+1][i] = (psi[j+1][i+1]-psi[j+1][i])/dx;
            }
        }
    }

/* 70 */
    for(j=1; j<=n; j++){
        u[j][1] = u[j][m+1];
        v[j+1][m+1] = v[j+1][1];
    }

/* 75 */
    for(i=1; i<=m; i++){
        u[n+1][i+1] = u[1][i+1];
        v[1][i] = v[n+1][i];
    }

    u[n+1][1] = u[1][m+1];
    v[1][m+1] = v[n+1][1];

    for(j=1; j<=np1; j++){
        for(i=1; i<=mp1; i++){
            uold[j][i] = u[j][i];
            vold[j][i] = v[j][i];
            pold[j][i] = p[j][i];
        }
    }
```

```
}

void calc1(void)
{
   int i, j;
   float fsdx = 4.0f/dx;
   float fsdy = 4.0f/dy;

/* 100 */
   for(j=1; j<=n; j++){
      if(j%4 == 0){
      for(i=1; i<=m; i++){
         cu[j][i+1]  = 0.5f*(p[j][i+1]+p[j][i])*u[j][i+1];
         cv[j+1][i]  = 0.5f*(p[j+1][i]+p[j][i])*v[j+1][i];

         z[j+1][i+1]    = (  fsdx*(v[j+1][i+1]-v[j+1][i])
                   - fsdy*(u[j+1][i+1]-u[j][i+1]) )
                   /(p[j][i]+p[j][i+1]+p[j+1][i+1]+p[j+1][i]);
         h[j][i]  = p[j][i] + 0.25f*(
                   u[j][i+1]*u[j][i+1] + u[j][i]*u[j][i]
                   + v[j+1][i]*v[j+1][i] + v[j][i]*v[j][i] );
      }
      }else if(j%4 == 1){
      for(i=1; i<=m; i++){
         cu[j][i+1]  = 0.5f*(p[j][i+1]+p[j][i])*u[j][i+1];
         cv[j+1][i]  = 0.5f*(p[j+1][i]+p[j][i])*v[j+1][i];

         z[j+1][i+1]    = (  fsdx*(v[j+1][i+1]-v[j+1][i])
                   - fsdy*(u[j+1][i+1]-u[j][i+1]) )
                   /(p[j][i]+p[j][i+1]+p[j+1][i+1]+p[j+1][i]);
         h[j][i]  = p[j][i] + 0.25f*(
                   u[j][i+1]*u[j][i+1] + u[j][i]*u[j][i]
                   + v[j+1][i]*v[j+1][i] + v[j][i]*v[j][i] );
      }
      }else if(j%4 == 2){
      for(i=1; i<=m; i++){
         cu[j][i+1]  = 0.5f*(p[j][i+1]+p[j][i])*u[j][i+1];
         cv[j+1][i]  = 0.5f*(p[j+1][i]+p[j][i])*v[j+1][i];

         z[j+1][i+1]    = (  fsdx*(v[j+1][i+1]-v[j+1][i])
                   - fsdy*(u[j+1][i+1]-u[j][i+1]) )
                   /(p[j][i]+p[j][i+1]+p[j+1][i+1]+p[j+1][i]);
         h[j][i]  = p[j][i] + 0.25f*(
                   u[j][i+1]*u[j][i+1] + u[j][i]*u[j][i]
                   + v[j+1][i]*v[j+1][i] + v[j][i]*v[j][i] );
      }
      }else if(j%4 == 3){
      for(i=1; i<=m; i++){
         cu[j][i+1]  = 0.5f*(p[j][i+1]+p[j][i])*u[j][i+1];
         cv[j+1][i]  = 0.5f*(p[j+1][i]+p[j][i])*v[j+1][i];

         z[j+1][i+1]    = (  fsdx*(v[j+1][i+1]-v[j+1][i])
                   - fsdy*(u[j+1][i+1]-u[j][i+1]) )
                   /(p[j][i]+p[j][i+1]+p[j+1][i+1]+p[j+1][i]);
         h[j][i]  = p[j][i] + 0.25f*(
                   u[j][i+1]*u[j][i+1] + u[j][i]*u[j][i]
                   + v[j+1][i]*v[j+1][i] + v[j][i]*v[j][i] );
      }
      }
   }
/* 110 */
   for(j=1; j<=n; j++){
      u[j][1]  = cu[j][m+1];
      cv[j+1][m+1]   = cv[j+1][1];
      z[j+1][1]   = z[j+1][m+1];
      h[j][m+1]   = h[j][1];
   }

/* 115 */
```

```
    for(i=1; i<=m; i++){
       cu[n+1][i+1]    = cu[1][i+1];
       cv[1][i]      = cv[n+1][i];
       z[1][i+1]    = z[n+1][i+1];
       h[1][i]   = h[n+1][i];
    }

    cu[n+1][1]   = cu[1][m+1];
    cv[1][m+1]   = cv[n+1][1];
    z[1][1]   = z[n+1][m+1];
    h[n+1][m+1]     = h[1][1];

}

void calc2(void)
{
    int i, j;
    float tdts8 = tdt/8.0f;
    float tdtsdx = tdt/dx;
    float tdtsdy = tdt/dy;

/* 200 */
    for(j=1; j<=n; j++){
       if(j%4 == 0){
       for(i=1; i<=m; i++){
          unew[j][i+1] = uold[j][i+1] + tdts8*(z[j+1][i+1]+z[j][i+1])
             * (cv[j+1][i+1]+cv[j+1][i]+cv[j][i]+cv[j][i+1])
             - tdtsdx*(h[j][i+1] - h[j][i]);
          vnew[j+1][i] = vold[j+1][i] - tdts8*(z[j+1][i+1]+z[j+1][i])
             * (cu[j+1][i+1]+cu[j+1][i] + cu[j][i]+cu[j][i+1])
             - tdtsdy*(h[j+1][i]-h[j][i]);
          pnew[j][i] = pold[j][i] - tdtsdx*(cu[j][i+1]-cu[j][i])
             - tdtsdy*(cv[j+1][i]-cv[j][i]);
       }
       }else if(j%4 == 1){
       for(i=1; i<=m; i++){
          unew[j][i+1] = uold[j][i+1] + tdts8*(z[j+1][i+1]+z[j][i+1])
             * (cv[j+1][i+1]+cv[j+1][i]+cv[j][i]+cv[j][i+1])
             - tdtsdx*(h[j][i+1] - h[j][i]);
          vnew[j+1][i] = vold[j+1][i] - tdts8*(z[j+1][i+1]+z[j+1][i])
             * (cu[j+1][i+1]+cu[j+1][i] + cu[j][i]+cu[j][i+1])
             - tdtsdy*(h[j+1][i]-h[j][i]);
          pnew[j][i] = pold[j][i] - tdtsdx*(cu[j][i+1]-cu[j][i])
             - tdtsdy*(cv[j+1][i]-cv[j][i]);
       }
       }else if(j%4 == 2){
       for(i=1; i<=m; i++){
          unew[j][i+1] = uold[j][i+1] + tdts8*(z[j+1][i+1]+z[j][i+1])
             * (cv[j+1][i+1]+cv[j+1][i]+cv[j][i]+cv[j][i+1])
             - tdtsdx*(h[j][i+1] - h[j][i]);
          vnew[j+1][i] = vold[j+1][i] - tdts8*(z[j+1][i+1]+z[j+1][i])
             * (cu[j+1][i+1]+cu[j+1][i] + cu[j][i]+cu[j][i+1])
             - tdtsdy*(h[j+1][i]-h[j][i]);
          pnew[j][i] = pold[j][i] - tdtsdx*(cu[j][i+1]-cu[j][i])
             - tdtsdy*(cv[j+1][i]-cv[j][i]);
       }
       }else if(j%4 == 3){
       for(i=1; i<=m; i++){
          unew[j][i+1] = uold[j][i+1] + tdts8*(z[j+1][i+1]+z[j][i+1])
             * (cv[j+1][i+1]+cv[j+1][i]+cv[j][i]+cv[j][i+1])
             - tdtsdx*(h[j][i+1] - h[j][i]);
          vnew[j+1][i] = vold[j+1][i] - tdts8*(z[j+1][i+1]+z[j+1][i])
             * (cu[j+1][i+1]+cu[j+1][i] + cu[j][i]+cu[j][i+1])
             - tdtsdy*(h[j+1][i]-h[j][i]);
          pnew[j][i] = pold[j][i] - tdtsdx*(cu[j][i+1]-cu[j][i])
             - tdtsdy*(cv[j+1][i]-cv[j][i]);
       }
       }
    }
```

```
/* 210 */
   for(j=1; j<=n; j++){
      unew[j][1]   = unew[j][m+1];
      vnew[j+1][m+1]     = vnew[j+1][1];
      pnew[j][m+1]    = pnew[j][1];
   }


/* 215 */
   for(i=1; i<=m; i++){
      unew[n+1][i+1] = unew[1][i+1];
      vnew[1][i]   = vnew[n+1][i];
      pnew[n+1][i]    = pnew[1][i];
   }

   unew[n+1][1]    = unew[1][m+1];
   vnew[1][m+1]    = vnew[n+1][1];
   pnew[n+1][m+1]     = pnew[1][1];
}

void calc3(void)
{
   int i, j;
/* 300 */
   for(j=1; j<=n; j++){
      for(i=1; i<=m; i++){
         uold[j][i] = u[j][i]
            + alpha*(unew[j][i]-2.0f*u[j][i]+uold[j][i]);
         vold[j][i] = v[j][i]
            + alpha*(vnew[j][i]-2.0f*v[j][i]+vold[j][i]);
         pold[j][i] = p[j][i]
            + alpha*(pnew[j][i]-2.0f*p[j][i]+pold[j][i]);

         u[j][i] = unew[j][i];
         v[j][i] = vnew[j][i];
         p[j][i] = pnew[j][i];
      }
   }

/* 320 */
   for(j=1; j<=n; j++){
      uold[j][m+1] = uold[j][1];
      vold[j][m+1] = vold[j][1];
      pold[j][m+1] = pold[j][1];
      u[m+1][j] = u[j][1];
      v[m+1][j] = v[j][1];
      p[m+1][j] = p[j][1];
   }
/* 325 */
   for(i=1; i<=m; i++){
      uold[n+1][i] = uold[1][i];
      vold[n+1][i] = vold[1][i];
      pold[n+1][i] = pold[1][i];
      u[n+1][i] = u[1][i];
      v[n+1][i] = v[1][i];
      p[n+1][i] = p[1][i];
   }

   uold[n+1][m+1] = uold[1][1];
   vold[n+1][m+1] = vold[1][1];
   pold[n+1][m+1] = pold[1][1];
   u[n+1][m+1] = u[1][1];
   v[n+1][m+1] = v[1][1];
   p[n+1][m+1] = p[1][1];

}

void calc3z(void)
```

```
{
    int i, j;
    tdt = tdt+tdt;
    for(j=1; j<=np1; j++){
        for(i=1; i<=mp1; i++){
            uold[j][i] = u[i][j];
            vold[j][i] = v[j][i];
            pold[j][i] = p[j][i];
            u[j][i] = unew[j][i];
            v[j][i] = vnew[j][i];
            p[j][i] = pnew[j][i];
        }
    }
}

void do_swim(void)
{
    int i, j;
    int ncycle, icheck, jcheck;
    float mnmin, time;
    float pcheck, ucheck, vcheck, ptime;

    initial();
/*
    printf(  "Number of points in the x direction %d\n"\
        "Number of poitns in the y direction %d\n"\
        "Grid spacing in the x direction %f\n"\
        "Grid spacing in the y direction %f\n"\
        "Time step %f\n"\
        "Time filter parameter %f\n"\
        "Number of iterations %d\n", n, m, dx, dy, dt, alpha, itmax);
*/
    mnmin = min(m, n);
    time = 0;
    ncycle = 0;

L90:  ncycle = ncycle + 1;

    calc1();
    calc2();

    time = time + dt;
    if(ncycle % mprint != 0)
        goto L370;

    ptime = time/3600.0f;
/*
    printf("Cycle number %d\n Model time in hours %f\n", ncycle, ptime);
*/
    pcheck = 0.0f;
    ucheck = 0.0f;
    vcheck = 0.0f;

/* 3500 */
    for(icheck = 1; icheck <= mnmin; icheck++){
        for(jcheck = 1; jcheck <= mnmin; jcheck++){
            pcheck = pcheck + abs(pnew[jcheck][icheck]);
            ucheck = ucheck + abs(unew[jcheck][icheck]);
            vcheck = vcheck + abs(vnew[jcheck][icheck]);
        }
    }
/*
    printf("Pcheck = %f\nUcheck = %f\nVcheck = %f\n", pcheck, ucheck, vcheck);
*/
L370:
    if(ncycle >= itmax)
        return;

    if(ncycle <= 1)
```

```
      calc3z();
   else
      calc3();

   goto L90;
}

int main(void)
{

   int i,j;
   struct tms start;
   struct tms end;
   long min, diff;
   min = 9999999;

   for(j=0; j<5; j++)
   {
      times(&start);
      do_swim();
      times(&end);
      diff = ((end.tms_utime - start.tms_utime) * 1000.0) / sysconf(_SC_CLK_TCK);
      min = min > diff ? diff : min;
   }

   printf("%ld\n", min);

   return 0;
}
```

A.2 Spec95.Tomcatv

```
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>

#define max(a, b) (a > b ? a : b)
#define abs(a)    (a < 0 ?(-a) : a)
#define  NMAX  513
#define ITMAX  1000
int   n, itact, iter;

double   c25 = 0.25;
double   c125 = 0.125;
double   c2 = 2.0;
double   neg = -1.0;
double   rel = 2.0/0.98;
double   relfa    = 0.98;
double   eps   = 0.5E-8;

double   dd[NMAX][NMAX];
double   aa[NMAX][NMAX];
double    x[NMAX][NMAX];
double    y[NMAX][NMAX];
double   rx[NMAX][NMAX];
double   ry[NMAX][NMAX];

double   d[NMAX][NMAX];
double   rxm[ITMAX], rym[ITMAX];
double   r, abx, aby;
double   pxx[NMAX], qxx[NMAX], pyy[NMAX], qyy[NMAX], pxy[NMAX], qxy[NMAX];
double   a[NMAX], b[NMAX], c[NMAX];

int init(void)
{
   int   i, j;
   FILE* file;
```

```
    /* do input/output */
    file = fopen("TOMCATV.MODEL", "r");
    if(!file){
       printf("file 'TOMCATV.MODEL' does not exist; stop\n");
       return 1;
    }
    for(j=1; j<=NMAX-1; j++)
    {
       for(i=1; i<=NMAX-1; i++)
       {
          double tmp1, tmp2;
          fscanf(file, "%e%e", &tmp1, &tmp2);
          x[j][i] = (double) tmp1;
          y[j][i] = (double) tmp2;
       }
    }
    fclose(file);
}

void finish(void)
{
    int    i, j;
    /* do IO */
    printf("      2-D ITERATION BEHAVIOR\n");
    printf("    IT     X-RES          Y-RES\n");

    for(i=1; i<=iter-1; i++)
    {
       printf("   %d     %011f   %011f\n", i, rxm[i], rym[i]);
    }
    printf("\n\n");
}

int do_tomcatv(void)
{
    int    i, j;
    /* initialize constant values */
    n  = NMAX;
    itact     = ITMAX;


    /* perform algorithm */
    for(iter=1; iter<=itact; iter++)
    {
       rxm[iter] = 0.0;
       rym[iter] = 0.0;

       /* residuals of iter iteration */
       for(j=1; j<=n-1; j++)
       {
          if(j%4==0){
             for(i=1; i<=n-1; i++)
             {
                pxx[i] = x[j][i+1]-x[j][i-1];
                qxy[i] = y[j][i+1]-y[j][i-1];
                pxy[i] = x[j+1][i]-x[j-1][i];
                pyy[i] = y[j+1][i]-y[j-1][i];
                a[i]   = c25   * (pxy[i]*pxy[i]+pyy[i]*pyy[i]);
                b[i]   = c25   * (pxx[i]*pxx[i]+qxy[i]*qxy[i]);
                c[i]   = c125  * (pxx[i]*pxy[i]+qxy[i]*pyy[i]);
                aa[j][i] = neg * b[i];
                dd[j][i] = b[i]+b[i]+a[i]*rel;
                pxx[i] = x[j][i+1]-c2*x[j][i]+x[j][i-1];
                qxx[i] = y[j][i+1]-c2*y[j][i]+y[j][i-1];
                pyy[i] = x[j+1][i]-c2*x[j][i]+x[j-1][i];
                qyy[i] = y[j+1][i]-c2*y[j][i]+y[j-1][i];
                pxy[i] = x[j+1][i+1]-x[j-1][i+1]-x[j+1][i-1]+x[j-1][i-1];
                qxy[i] = y[j+1][i+1]-y[j-1][i+1]-y[j+1][i-1]+y[j-1][i-1];
```

```
                rx[j][i]    = a[i]*pxx[i]+b[i]*pyy[i]-c[i]*pxy[i];
                ry[j][i]    = a[i]*qxx[i]+b[i]*qyy[i]-c[i]*qxy[i];
             }
          }else if(j%4==1){
             for(i=1; i<=n-1; i++)
             {
                pxx[i] = x[j][i+1]-x[j][i-1];
                qxy[i] = y[j][i+1]-y[j][i-1];
                pxy[i] = x[j+1][i]-x[j-1][i];
                pyy[i] = y[j+1][i]-y[j-1][i];
                a[i]   = c25  * (pxy[i]*pxy[i]+pyy[i]*pyy[i]);
                b[i]   = c25  * (pxx[i]*pxx[i]+qxy[i]*qxy[i]);
                c[i]   = c125 * (pxx[i]*pxy[i]+qxy[i]*pyy[i]);
                aa[j][i] = neg * b[i];
                dd[j][i] = b[i]+b[i]+a[i]*rel;
                pxx[i] = x[j][i+1]-c2*x[j][i]+x[j][i-1];
                qxx[i] = y[j][i+1]-c2*y[j][i]+y[j][i-1];
                pyy[i] = x[j+1][i]-c2*x[j][i]+x[j-1][i];
                qyy[i] = y[j+1][i]-c2*y[j][i]+y[j-1][i];
                pxy[i] = x[j+1][i+1]-x[j-1][i+1]-x[j+1][i-1]+x[j-1][i-1];
                qxy[i] = y[j+1][i+1]-y[j-1][i+1]-y[j+1][i-1]+y[j-1][i-1];
                rx[j][i]    = a[i]*pxx[i]+b[i]*pyy[i]-c[i]*pxy[i];
                ry[j][i]    = a[i]*qxx[i]+b[i]*qyy[i]-c[i]*qxy[i];
             }
          }
       }

       /* Determine maximum values rxm, rym of residuals */
/*V 80 */    for(j=1; j<=n-1; j++)
       {
          for(i=1; i<n-1; i++)
          {
             rxm[iter] = max(rxm[iter], abs(rx[j][i]));
             rym[iter] = max(rym[iter], abs(ry[j][i]));
          }
       }

/*V 90 */    for(i=1; i<=n-1; i++)
       {
          d[2][i] = 1.0/dd[2][i];
       }


/*V 100 */   for(j=2; j<= n-1; j++)
       {
          for(i=1; i<=n-1; i++)
          {
             r = aa[j][i]*d[j-1][i];
             d[j][i] = 1.0/ (dd[j][i] - aa[j-1][i] * r);
             rx[j][i] = rx[j][i] - rx[j-1][i]*r;
             ry[j][i] = ry[j][i] - ry[j-1][i]*r;
          }
       }

/*V 110 */   for(i=1; i<=n-1; i++)
       {
          rx[n-1][i] = rx[n-1][i]*d[n-1][i];
          ry[n-1][i] = ry[n-1][i]*d[n-1][i];
       }

/*V 120 */   for(j=n-2; j>=1; j--)
       {
          for(i=1; i<= n-1; i++)
          {
              rx[j][i] = (rx[j][i]-aa[j][i]*rx[j+1][i])*d[j][i];
              ry[j][i] = (ry[j][i]-aa[j][i]*ry[j+1][i])*d[j][i];
          }
       }
```

```
/*V 130 */  for(j=1; j<=n-1; j++)
      {
         for(i=1; i<=n-1; i++)
         {
            x[j][i] = x[j][i] + rx[j][i];
            y[j][i] = y[j][i] + ry[j][i];
         }
      }

      abx = abs(rxm[iter]);
      aby = abs(rym[iter]);

/*     if(abx <= eps && aby <= eps)
         break;
*/
   }

   return 0;
}


int main(void)
{
   int i,j;
   struct tms start;
   struct tms end;
   long min, diff;
   min = 9999999;

   /* Initialize */
   if(init())
      return 1;

   for(j=0; j<5; j++)
   {
      times(&start);
      do_tomcatv();
      times(&end);
      diff = ((end.tms_utime - start.tms_utime) * 1000.0) / sysconf(_SC_CLK_TCK);
      min = min > diff ? diff : min;
   }
   //finish();

   printf("%ld\n", min);

   return 0;
}
```

A.3 Livermore

```
#include <stdio.h>
#include <assert.h>
#include <sys/times.h>
#include <unistd.h>

float     dm22,dm23,dm24,dm25,dm26,dm27,dm28,q,r,t,c0;
float     u[1001], x[1001], y[1001], z[1001];
float     px[25][1001];


/* Hydro fragment */
void kernel1()
{
   int l, k;
   int n = 1001;
   int lp = 80000;
```

```
    for (l = 0; l < lp; l++)
        for (k = 0; k < n; k++)
            x[k]= q + y[k]*(r*z[k+10] + t*z[k+11]);
}

/* Equation of state fragment */
void kernel7()
{
    int l, k;
    int n = 1001;
    int lp = 80000;

    for (l = 0; l < lp; l++){
        for (k = 0; k < n; k++) {
            x[k]=           u[k  ] + r*( z[k  ] + r*y[k  ])
                + t*( u[k+3] + r*( u[k+2] + r*u[k+1])
                + t*( u[k+6] + r*( u[k+5] + r*u[k+4])));
        }
    }
}

/* integrate Predictors */
void kernel9()
{
    int i, l;
    int n = 1001;
    int lp = 80000;

    for (l = 0; l < lp; l++){
        for (i = 0; i < n; i++) {
#ifndef OPTIM
            px[ 0][i] = dm28*px[12][i] + dm27*px[11][i] + dm26*px[10][i] +
                    dm25*px[9][i] + dm24*px[8][i] + dm23*px[7][i] +
                    dm22*px[6][i] + c0*(px[4][i] + px[5][i]) + px[ 2][i];
#else
            px[ 0][i]=
                  (dm28*px[12][i] +  dm24*px[ 8][i])
                + (dm27*px[11][i] +  dm23*px[ 7][i])
                + (dm26*px[10][i] +  dm22*px[ 6][i] + px[ 2][i])
                + (dm25*px[ 9][i] +  c0 * px[ 5][i])
                + c0* px[ 4][i];
#endif
        }
    }
}

/* First difference */
void kernel12()
{
    int l, k;
    int n = 1001;
    int lp = 500000;

    for (l = 0; l < lp; l++)
        for (k = 0; k < n; k++)
            x[k]= y[k+1] - y[k];
}

typedef void (* kern_fp )( void );

int main(int argc, char** argv)
{
    int   i, j, k;
    struct   tms start;
    struct   tms end;
    long  min, diff;
    kern_fp  kernel;

    /* Get kernel */
```

```
    if(argc != 2){
       printf("./livermore [1, 7, 9, 12]\n"\
       "\tExample: ./livermore 9\n");
       return 1;
    }
    assert(sscanf(argv[1], "%d", &k));
    switch(k){
       case 1: kernel = kernel1; break;
       case 7: kernel = kernel7; break;
       case 9: kernel = kernel9; break;
       case 12: kernel = kernel12; break;
    }

    /* Benchmark */
    min = 9999999;
    for(j=0; j<10; j++)
    {
       times(&start);
          kernel();
       times(&end);
       diff = ((end.tms_utime - start.tms_utime) * 1000.0) / sysconf(_SC_CLK_TCK);
       min = min > diff ? diff : min;
    }

    printf("%ld\n", min);

    return 0;
}
```

A.4 Generated loop (example)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/times.h>

float z[1024];
float dummy2[31];
float b[1024];
float dummy1[31];
float a[1024];
float dummy0[31];

void init(void)
{
    int iter;
    for(iter=0; iter<1024; iter++)
    {
       a[iter] = (float) (iter%20);
    }
    for(iter=0; iter<1024; iter++)
    {
       b[iter] = (float) (iter%20);
    }
}

void check(void)
{
    int    iter;
    int    check;
    check = 1;
    for(iter=0; iter<1020; iter++)
    {
       check = check && (z[iter+8] == a[iter+8] - b[iter+9] * a[iter+9] * b[iter+8] * a[iter+8] - b[iter+7]);
       check = check && (z[iter+7] == a[iter+10] + b[iter+6] * a[iter+7] - b[iter+7] * a[iter+7] * b[iter+7]);
       check = check && (z[iter+6] == a[iter+6] * b[iter+6] * a[iter+3] * b[iter+9] + a[iter+8] * b[iter+6]);
       check = check && (z[iter+5] == a[iter+2] * b[iter+3] * a[iter+6] * b[iter+3] * a[iter+6] + b[iter+6]);
       check = check && (z[iter+4] == a[iter+5] * b[iter+2] * a[iter+4] - b[iter+4] * a[iter+2] * b[iter+3]);
```

```
    }
    printf("Check "); check ? printf("Passed\n") : printf("Failed\n");

}
static void kernel(void)
{
    int iter;
    for(iter=0; iter<1020; iter++)
    {
        z[iter+8] = a[iter+8] - b[iter+9] * a[iter+9] * b[iter+8] * a[iter+8] - b[iter+7];
        z[iter+7] = a[iter+10] + b[iter+6] * a[iter+7] - b[iter+7] * a[iter+7] * b[iter+7];
        z[iter+6] = a[iter+6] * b[iter+6] * a[iter+3] * b[iter+9] + a[iter+8] * b[iter+6];
        z[iter+5] = a[iter+2] * b[iter+3] * a[iter+6] * b[iter+3] * a[iter+6] + b[iter+6];
        z[iter+4] = a[iter+5] * b[iter+2] * a[iter+4] - b[iter+4] * a[iter+2] * b[iter+3];

    }
}

int main(void)
{
    int iter,j;
    struct tms start;
    struct tms end;
    long min, diff;
    min = 9999999;

    init();
    for(iter=0; iter<5; iter++)
    {
        times(&start);
        for(j=0; j < 12000; j++)
        {
            kernel();
        }
        times(&end);
        diff = ((end.tms_utime - start.tms_utime) * 1000.0) / sysconf(_SC_CLK_TCK);
        min = min > diff ? diff : min;
    }
    check();
    printf("%ld\n", min);
    return 0;
}
```

A.5 Motion estimation

```
#include <sys/times.h>
#include <unistd.h>

#define make_vector(...)    0
#define abs(a) ((a) < 0 ? -(a) : (a))
#define KSL -KS/2
#define KSU KS/2

short ref[HEIGHT][WIDTH];
short other[HEIGHT+KS][WIDTH+KS];
short match[BS][BS];


static inline void diff(int blocky, int blockx, int dy, int dx)
{
    int y, x;
    for(y=0; y<BS; y++)
        for(x=0; x<BS; x++)
            match[y][x] = ref[blocky+y][blockx+x] - other[blocky+dy+y][blockx+dx+x];

}
```

```
static inline int sad(void)
{
   int sum, y, x;

   sum = 0;
   for(y=0; y<BS; y++)
      for(x=0; x<BS; x++)
         sum += abs(match[y][x]);

   return sum;
}

void blockmatch(void)
{
   int dy, dx;
   int blocky, blockx;
   int mindx, mindy, min, tmp;

   //block
   for(blocky = KS; blocky < HEIGHT-BS; blocky += BS)
   {
      for(blockx = KS; blockx < WIDTH-BS; blockx += BS)
      {
         tmp = 99999;
         for(dy = KSL; dy < KSU; dy++)
         {
            for(dx = KSL; dx < KSU; dx++)
            {
               if(dx%BS < -4){
                  if(dx%BS < -6){
                     if(dx%BS == -7)
                        diff(blocky, blockx, dy, dx);
                     else if(dx%BS == -8)
                        diff(blocky, blockx, dy, dx);
                  }else{
                     if(dx%BS == -6)
                        diff(blocky, blockx, dy, dx);
                     else if(dx%BS == -5)
                        diff(blocky, blockx, dy, dx);
                  }
               }else if(dx%BS < 0){
                  if(dx%BS < -2){
                     if(dx%BS == -3)
                        diff(blocky, blockx, dy, dx);
                     else if(dx%BS == -4)
                        diff(blocky, blockx, dy, dx);
                  }else{
                     if(dx%BS == -2)
                        diff(blocky, blockx, dy, dx);
                     else if(dx%BS == -1)
                        diff(blocky, blockx, dy, dx);
                  }
               }else if(dx%BS > 4){
                  if(dx%BS > 6){
                     if(dx%BS == 7)
                        diff(blocky, blockx, dy, dx);
                     else if(dx%BS == 8)
                        diff(blocky, blockx, dy, dx);
                  }else{
                     if(dx%BS == 6)
                        diff(blocky, blockx, dy, dx);
                     else if(dx%BS == 5)
                        diff(blocky, blockx, dy, dx);
                  }
               }else if(dx%BS > 0){
                  if(dx%BS > 2){
                     if(dx%BS == 3)
                        diff(blocky, blockx, dy, dx);
                     else if(dx%BS == 4)
```

```
                        diff(blocky, blockx, dy, dx);
                    }else{
                        if(dx%BS == 2)
                            diff(blocky, blockx, dy, dx);
                        else if(dx%BS == 1)
                            diff(blocky, blockx, dy, dx);
                    }
                }else if(dx%BS == 0){
                    diff(blocky, blockx, dy, dx);
                }

                tmp = sad();
                if(tmp < min){
                    mindy = dy;
                    mindx = dx;
                    min = tmp;
                }
            }

        }
    }
}

void main(void)
{
    int i,j;
    struct tms start;
    struct tms end;
    long min, diff;
    min = 9999999;

    for(j=0; j<10; j++)
    {
        times(&start);
            blockmatch();
        times(&end);
        diff = ((end.tms_utime - start.tms_utime) * 1000.0) / sysconf(_SC_CLK_TCK);
        min = min > diff ? diff : min;
    }

    printf("%ld\n", min);
}
```

Appendix B

A Compiler Comparison