Compilation Techniques for Short-Vector Instructions

by

Samuel Larsen

Bachelor of Science, Computer Engineering
University of Utah, 1998

Master of Science, Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2000

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

April 2006

Author . . . .
Department of Electrical Engineering and Computer Science
April 14, 2006

Certified by . . . .
Saman P. Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Compilation Techniques for Short-Vector Instructions

by

Samuel Larsen

Submitted to the Department of Electrical Engineering and Computer Science
on April 14, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Multimedia extensions are nearly ubiquitous in today's general-purpose processors. These extensions consist primarily of a set of short-vector instructions that apply the same opcode to a vector of operands. This design introduces a data-parallel component to processors that exploit instruction-level parallelism, and presents an opportunity for increased performance. In fact, ignoring a processor's vector opcodes can leave a significant portion of the available resources unused. In order for software developers to find short-vector instructions generally useful, the compiler must target these extensions with complete transparency and consistent performance.

This thesis develops compiler techniques to target short-vector instructions automatically and efficiently. One important aspect of compilation is the effective management of memory alignment. As with scalar loads and stores, vector references are typically more efficient when accessing aligned regions. In many cases, the compiler can glean no alignment information and must emit conservative code sequences. In response, I introduce a range of compiler techniques for detecting and enforcing aligned references. In my benchmark suite, the most practical method ensures alignment for roughly 75% of dynamic memory references.

This thesis also introduces selective vectorization, a technique for balancing computation across a processor's scalar *and* vector resources. Current approaches for targeting short-vector instructions directly adopt vectorizing technology first developed for supercomputers. Traditional vectorization, however, can lead to a performance degradation since it fails to account for a processor's scalar execution resources. I formulate selective vectorization in the context of software pipelining. My approach creates software pipelines with shorter initiation intervals, and therefore, higher performance. In contrast to conventional methods, selective vectorization operates on a low-level intermediate representation. This technique allows the algorithm to accurately measure the performance trade-offs of code selection alternatives.

A key aspect of selective vectorization is its ability to manage communication of operands between vector and scalar instructions. Even when operand transfer is expensive, the technique is sufficiently sophisticated to achieve significant performance gains. I evaluate selective vectorization on a set of SPEC FP benchmarks. On a realistic VLIW processor model, the approach achieves whole-program speedups of up to 1.35× over existing approaches. For individual loops, it provides speedups of up to 1.75×.

Thesis Supervisor: Saman P. Amarasinghe
Title: Associate Professor

Acknowledgments

This thesis would not exist without the help, advice, and encouragement of many people. I thank my advisor, Saman, for giving me the freedom and resources to pursue research that interested me. I am especially grateful for his guidance and accessibility through nearly eight years of graduate school.

I am also indebted to those who donated their time to read various drafts of the thesis, and whose comments greatly improved the quality of this document: Alexandre Eichenberger and Aart Bik, and particularly the members of my thesis committee, Krste Asanović and Charles Leiserson.

Many thanks to the members of CAG and Commit, who made the lab an exciting place to work, debate, and hang out. So many people enriched my experience. I especially want to thank Mary McDavitt, who was the first to arrive at my defense, and who convinced me that I could actually pass. I am also grateful to Matt Frank, who was as much an advisor as a colleague.

I devote a separate paragraph to Rodric Rabbah. Without his help, I'd still be toiling with implementation details. Rodric may be the most productive computer scientist I've ever met. I was fortunate to have redirected some of his energy to my projects. I also appreciate his advice and his perpetual positive attitude.

For me, graduate school was tolerable because I had a group of close friends. Mine were the best: Mark Stephenson, Mike Zhang, Mike Gordon, Steve Gerding, Bill McNicol.

Finally, I thank my family. To my brother, Mike. Your ability to create anything is an inspiration. To my sister, Sara. The two years you spent in Boston were my best. To my parents, who were ultimately the people who prepared me for this endeavor. I owe you all my success. And lastly, to Steph. What can I say? You read every word of every tech report, conference paper, and thesis I wrote. How could I have persevered without you?

This research was funded in part by NSF grants EIA9810173, EIA-0071841, and CCR-0073510, and DARPA grants DBT63-96-C-0036, F29601-01-2-0166, PCA-F29601-03-2-0065, F3060200-2-0562, and HPCA/PERCS-W0133890.

Contents

1 Introduction **9**

1.1 Address Alignment 11
1.2 Code Selection and Resource Allocation 12
1.3 Selective Vectorization 16
1.4 Contributions 19
1.5 Thesis Outline 21

2 Multimedia Extensions **23**

2.1 Overview 23
2.2 Historical Perspective 25
2.3 Compilation for Multimedia Extensions 30

3 Background **35**

3.1 Vectorization 36
3.2 Superword-Level Parallelism 41
3.3 Software Pipelining 44

4 Selective Vectorization **49**

4.1 Intermediate Representation 50
4.2 Algorithm Description 52
4.3 Partitioning Example 56
4.4 Cost Calculation for Complex Interfaces 59
4.5 Modulo Scalar Expansion 64
4.6 Code Generation 66
4.7 Related Work 68

5 Address Alignment **71**

5.1 Background 72
5.2 Alignment Analysis 74
5.3 Interprocedural Alignment Analysis 78
5.4 Array Padding 81
5.5 Multi-Versioning 83
5.6 Dynamic Loop Peeling 85
5.7 Alignment Profiling 87
5.8 Per-statement Peeling 91
5.9 Code Generation 93
5.10 Historical Perspective 96
5.11 Alignment and Selective Vectorization 97

6 Evaluation **99**
6.1 Methodology . . . 100
6.2 Selective Vectorization vs. Traditional Approaches . . . 103
6.3 Opportunities for Selective Vectorization . . . 108
6.4 Hardware Support for Operand Transfer . . . 110
6.5 Performance Impact of Alignment Information . . . 114
6.6 Vector vs. Scalar Operation Throughput . . . 116

7 Conclusion **121**
7.1 Limitations . . . 121
7.2 Future Work . . . 122
7.3 Summary . . . 124

Chapter 1

Introduction

Multimedia extensions represent one of the biggest advances in processor architecture in the past decade. Today, they are prevalent in embedded designs and nearly ubiquitous in general-purpose designs. Examples of multimedia extensions include VIS [90], MAX-2 [58], MMX [69], 3DNow! [68], SSE [72, 89], SSE2 [34], and VMX/AltiVec [20]. Multimedia extensions are one method by which processor architects have employed a wealth of available transistors. If used efficiently, they present an opportunity for large performance gains. Before multimedia extensions become generally useful to the computing community, however, they must be completely invisible to the high-level programmer.

This thesis develops compiler techniques that automatically target the primary component in multimedia extensions: short-vector instructions. Short-vector instructions apply the same opcode to vectors of operands, usually in parallel. This model of execution closely matches the structure of multimedia applications which often contain compute-intensive kernels that operate on streams of independent data. Vector instructions first appeared in vector supercomputers such as the Cray-1 [79], and more recently in vector microprocessors such as T0 [7] and VIRAM [49, 50]. In contrast to these designs, multimedia extensions provide vector instructions that operate on relatively short vectors of packed data. A short-vector design more readily integrates in existing general-purpose pipelines since the processor can operate on all elements simultaneously. By comparison, the long vectors implemented by vector supercomputers typically require iterative execution.

Compared to superscalar or VLIW execution, short-vector instructions offer three primary advantages. First, they provide a means for parallel execution which they realize with moderate architectural complexity and cost. For example, vector instructions require fewer register-file ports to accommodate an equivalent number of functional units. Similarly, vector units do not require an intra-element bypassing network. Second, vector instructions lessen the burden on the instruction-fetch unit and instruction cache since they specify multiple independent operations using a compact instruction encoding. Finally, vector memory instructions are particularly powerful since they exploit a wide memory interface to transfer multiple operands simultaneously.

Early multimedia extensions provided a modest set of short-vector instructions. Initial designs supported only a handful of integer arithmetic instructions that operated on short-width data of 8 or 16 bits. Over time, multimedia extensions have grown increasingly complex. Contemporary designs offer extensive vector instruction sets, including support for floating-point computation. As such, short-vector instructions now provide a potential performance improvement for a broader class of applications.

If short-vector instructions are to be widely used, they must be accessible through high-level languages. An early solution required the programmer to use inline assembly or processor-specific macro instructions. This approach provides a marginal advantage over pure assembly-level programming; it is tedious, error-prone, and not portable among different platforms. The use of inline assembly also necessitates an in-depth knowledge of the target processor's vector instruction set. Still worse, the programmer must manually identify opportunities for employing vector opcodes. As a result, early use of multimedia extensions was limited to hand-coded libraries and the inner loops of a few performance-critical kernels.

A better solution for employing short-vector instructions is to use compiler technology to target them automatically. This approach completely hides the processor's instruction set from the programmer and makes short-vector operations universally available. Automatic compilation faces two major challenges: identifying vector parallelism in sequential descriptions, and using this information to employ short-vector instructions efficiently. For the former, we can leverage decades of innovation devoted to compilation for vector supercomputers. In terms of performance, however, multimedia extensions offer new obstacles that the compiler must address. Two of the most important issues are address alignment of vector memory references and vector code selection. This thesis addresses both of these topics.

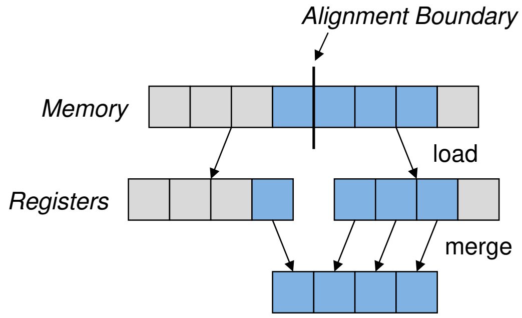


Figure 1-1: A misaligned vector load handled explicitly in software. The highlighted region represents the accessed data.

1.1 Address Alignment

One issue relating to performance is the alignment of short-vector memory instructions. As with loads and stores of basic data types, vector references are more efficient when accessing aligned regions. A memory operation is *aligned* if its address is a multiple of the data width. Most designs restrict memory references to aligned segments. For example, the VMX extension ignores the low-order bits of a vector memory address [37]. In order to access misaligned data, the compiler must explicitly merge values in two aligned regions. Figure 1-1 illustrates this process for a vector load. Misaligned vector stores are even more troublesome since the compiler must splice vector operands with existing values in memory. Intelligent code generation can eliminate much of this overhead in vector loops by reusing data produced in the previous iteration. This optimization removes the additional memory operations typically required by a misaligned access. The overhead of explicit merging is unavoidable, however, unless the compiler can guarantee alignment.

Some architectures, such as Intel's IA-32, support misaligned references directly in hardware [42]. IA-32 implementations experience a performance degradation when accessing misaligned regions, particularly if the data cross a cache-line boundary. The most recent IA-32 ISA provides both aligned and unaligned vector opcodes. The aligned versions provide higher performance [41], but are valid only when the compiler can guarantee alignment. Intel does not reveal the performance details of its misaligned opcodes, but one can assume they use a series of micro-operations to perform a function similar to that illustrated in Figure 1-1. For a simple loop that copies one array to another, my experiments revealed that a version utilizing aligned opcodes executes roughly 2× faster on a Pentium 4 than a version using unaligned opcodes.

```
void add(float* a, float* b, float* c) {
  ...
  for (i=0; i<N; i++) {
    a[i] = b[i] + c[i];
  }
}

int main() {
  float a[N+1], b[N+1], c[N+1];
  ...
  add(a, b, c);
  add(a+1, b+1, c+1);
  add(a+1, b+1, c+2);
}
```

Figure 1-2: Pointer arguments complicate alignment detection.

Whether misaligned references are supported in hardware or software, the compiler can improve performance by generating aligned vector memory references. Unfortunately, ensuring the safety of aligned opcodes is a difficult task. In most cases, the compiler has no information about the runtime alignment of a memory operation and must conservatively assume it is misaligned.

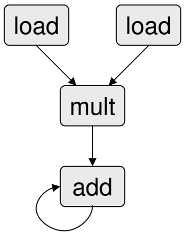
One of the biggest impediments to ensuring alignment is the use of pointer arguments, as shown in Figure 1-2. In order to determine whether vectorization of the loop in `add` will result in aligned references, the compiler must have information about the values of the procedure's pointer parameters. Since languages like C allow the programmer to pass pointers to arbitrary locations in memory, the alignments of the references in the loop depend on the calling context. Complicating the issue is the fact that different call sites may pass different offsets, as in the calls from `main` in Figure 1-2. This thesis introduces compiler techniques that both detect and enforce aligned accesses. The most effective approach combines loop-peeling with runtime profiling. For a set of SPEC FP benchmarks, the technique is able to ensure alignment for roughly 75% of dynamic references.

1.2 Code Selection and Resource Allocation

When targeting general-purpose architectures, the most important performance consideration is the overall code generation scheme. When compiling data-parallel loops, a popular approach is to directly adopt technology developed for vector supercomputers. Traditional vectorization, however, is not well-suited for compilation to multimedia extensions, and can

```
for (i=0; i<N; i++) {
  s = s + a[i] * b[i];
}
```

(a)



(b)

Cycle	Slot 1	Slot 2	Slot 3
1	load	load	
2	mult		
3	load	load	add
4	mult		
…	…	…	…

(c)

Figure 1-3: (a) Dot product kernel. (b) Its data-dependence graph. (c) A valid modulo schedule.

actually lead to a performance degradation. The primary cause for poor performance is the compiler's failure to account for a processor's scalar processing capabilities. General-purpose processors exploit instruction-level parallelism, or ILP. When loops contain a mix of vectorizable and non-vectorizable operations, the conventional approach *distributes* a loop into vector and scalar portions, destroying ILP and stifling the processor's ability to provide high performance. A traditional vectorizing compiler also vectorizes as many operations as possible. This approach is problematic for general-purpose processors which, in contrast to vector supercomputers, do not devote the vast majority of resources to vector execution. On today's general-purpose designs, full vectorization may leave a significant fraction of the processor's execution units underutilized.

As an example of the problems created by traditional vectorization, consider the dot product kernel in Figure 1-3 (a). Part (b) shows the kernel's data-dependence graph. For simplicity, the figure omits address calculations and loop control instructions. When compiling this loop for a general-purpose processor, one approach is to employ an ILP technique such as software pipelining to exploit the processor's scalar resources. Suppose we intend to execute the dot product on an architecture that exports three issue slots as the only compiler-visible resources, and single-cycle latency for all operations. Part (c) shows a low-level schedule for the operations in part (b). The schedule constitutes a software pipeline, and in particular, a *modulo schedule*. Chapter 3 provides a review of software pipelining. For this example, it is sufficient to focus on the schedule length of the *kernel*, highlighted in cycles 3 and 4. After the pipeline fills in the first two cycles, the kernel executes repeatedly, and is therefore the primary factor in overall loop performance. In modulo scheduling, the schedule length of the kernel is termed the *initiation interval*, or II. The schedule in part (c) has an initiation interval of 2 cycles.

```
for (i=0; i<N; i++) {
  t[i] = a[i] * b[i];
}
for (i=0; i<N; i++) {
  s = s + t[i];
}
```

(a)

Cycle	Slot 1	Slot 2	Slot 3
1	vload		
2	vload		
3	vmult		
4	vstore		
...	...	...	...

(b)

Cycle	Slot 1	Slot 2	Slot 3
1	load		
2	load	add	
...	...	...	...

(c)

Figure 1-4: (a) Vectorization of a dot product using loop distribution. (b) Modulo schedule for the vector loop. (c) Modulo schedule for the scalar loop.

If the target architecture supports short-vector instructions, the dot product in Figure 1-3 (a) is also a candidate for vectorization. The cyclic dependence, however, prevents vectorization off the `add` operation[1]. When faced with a partially-vectorizable loop, a traditional scheme distributes the loop into vectorizable and non-vectorizable portions, as shown in Figure 1-4 (a). *Scalar expansion* introduces a temporary array to communicate intermediate values between the loops. While this approach may be appropriate for vector supercomputers, it is not the best solution for a general-purpose design. In addition to the increased loop overhead, distribution eliminates parallelism among the operations in different loops.

Assume that the example architecture supports execution of one vector instruction each cycle, and that vector instructions operate on two elements. After vectorization of the code segment in Figure 1-4 (a), it is reasonable to apply software pipelining to both loops in order to gain as much performance as possible. Since the processor can execute only one vector instruction per cycle, software pipelining cannot discover additional parallelism in the vector loop. The four vector operations require four cycles to execute, as shown in Figure 1-4 (b). This schedule has an average initiation interval of 2 cycles since one iteration of the vector loop completes 2 iterations of the original loop. Part (c) shows a schedule for the scalar loop. Here, software pipelining achieves an initiation interval of 1 cycle. Overall, the vectorized dot product requires an average of $2 + 1 = 3$ cycles per iteration, which is inferior to software pipelining alone. Some of the performance degradation is due to the extra memory operations introduced by scalar expansion. Even if we ignore this overhead, however, vectorization does not outperform baseline modulo scheduling in this example.

[1]In some cases, the compiler can vectorize reductions using multiple partial summations. Since the scheme reorders the addition operations, it is not always legal (*e.g.*, when using floating-point data.)

```
for (i=0; i<N; i+=2) {
  T = a[i:i+1] * b[i:i+1];
  s = s + T(0);
  s = s + T(1);
}
```

(a)

Cycle	Slot 1	Slot 2	Slot 3
1	vload		
2	vload		
3	vmult		
4	vload	add	
5	vload	add	
6	vmult		
...	...	...	...

(b)

Figure 1-5: (a) Vectorized dot product without distribution. (b) Its modulo schedule.

As an alternative to loop distribution, a short vector length allows us to *unroll* the scalar operations within the vector loop. This approach, illustrated in Figure 1-5, allows vector and scalar instructions to execute concurrently. Part (a) shows the vectorized loop with the `add` operation unrolled by a factor of 2 to match the work output of the vector instructions. The corresponding modulo schedule in part (b) achieves an average initiation interval of 1.5 cycles since the kernel completes two iterations every three cycles.

The schedule in Figure 1-5 (b) provides a performance improvement over baseline modulo scheduling, but it does not make the most efficient use of available machine resources. In particular, the schedule overcommits vector resources and leaves several scalar issue slots unused. We can achieve better resource utilization by *selectively* vectorizing a subset of the candidate operations. For example, the dot product executes more efficiently on the example architecture if we do not vectorize the multiply operation. Figure 1-6 illustrates this approach. The modulo schedule in part (b) utilizes all issue slots in the kernel and executes a maximum of one vector operation each cycle.

```
for (i=0; i<N; i+=2) {
  T = a[i:i+1];
  V = b[i:i+1];
  s = s + T(0) * V(0);
  s = s + T(1) * V(1);
}
```

(a)

Cycle	Slot 1	Slot 2	Slot 3
1	vload		
2	vload		
3	vload	mult	
4	vload	mult	add
5	vload	mult	add
6	vload	mult	add
...	...	...	...

(b)

Figure 1-6: (a) Selectively vectorized dot product. (b) Its modulo schedule.

Technique	Average II
Software pipelining	2
Traditional vectorization	3
Vectorization without distribution	1.5
Selective vectorization	1

Figure 1-7: Initiation intervals for competing compilation techniques.

Figure 1-7 summarizes the initiation intervals achieved with each technique. By accounting for scalar and vector processing capabilities, selective vectorization produces an average initiation interval of 1 cycle and outperforms the other methods.

1.3 Selective Vectorization

The previous section illustrates the potential of balancing computation across scalar and vector resources. The example overlooks a crucial detail, however. Namely, the software pipeline in Figure 1-6 (b) does not account for explicit operand communication between vector and scalar instructions. The schedule assumes that operands retrieved via vector loads are immediately available to scalar multiplications. Modern multimedia extensions implement separate scalar and vector register files; transferring operands between them requires explicit instructions. Some designs, such as VMX, offer no specialized support and require the compiler to transfer operands through memory using a series of load and store instructions. If the compiler does not account for this overhead, communication costs could negate the benefits of selective vectorization. As I will demonstrate, judicious code selection can still produce large performance gains.

As an example of selective vectorization for a realistic architecture, I now target a hypothetical VLIW processor, which I term the L-machine. The model is representative of contemporary designs and has a resource set similar to IBM's PowerPC 970 [36] and Intel's

Processor Parameter	Value
Issue width	6
Integer units	4
Floating-point units	2
Memory units (scalar & vector)	2
Vector floating-point units	1
Vector length (64-bit elements)	2

Figure 1-8: Details of the L-machine, a hypothetical VLIW architecture supporting short-vector extensions.

```
      DO 60 J = 2,N-1
        DO 50 I = 2,N-1
          XX = X(I+1,J)-X(I-1,J)
          YX = Y(I+1,J)-Y(I-1,J)
          XY = X(I,J+1)-X(I,J-1)
          YY = Y(I,J+1)-Y(I,J-1)
          A  = 0.25D0  * (XY*XY+YY*YY)
          B  = 0.25D0  * (XX*XX+YX*YX)
          C  = 0.125D0 * (XX*XY+YX*YY)
          AA(I,J) = -B
          DD(I,J) = B+B+A*REL
          PXX = X(I+1,J)-2.D0*X(I,J)+X(I-1,J)
          QXX = Y(I+1,J)-2.D0*Y(I,J)+Y(I-1,J)
          PYY = X(I,J+1)-2.D0*X(I,J)+X(I,J-1)
          QYY = Y(I,J+1)-2.D0*Y(I,J)+Y(I,J-1)
          PXY = X(I+1,J+1)-X(I+1,J-1)-X(I-1,J+1)+X(I-1,J-1)
          QXY = Y(I+1,J+1)-Y(I+1,J-1)-Y(I-1,J+1)+Y(I-1,J-1)
          RX(I,J) = A*PXX+B*PYY-C*PXY
          RY(I,J) = A*QXX+B*QYY-C*QXY
50      CONTINUE
60    CONTINUE
```

Figure 1-9: Primary loop in `tomcatv`.

Itanium 2 [62]. Chapter 6 performs an extensive evaluation of selective vectorization using the L-machine. Figure 1-8 lists the high-level processor details that are pertinent to the immediate discussion.

Consider the code fragment in Figure 1-9. The inner loop accounts for roughly 50% of the total execution time of `tomcatv`, an application from the SPEC CFP95 benchmark suite [85]. Figure 1-10 shows the loop's low-level data dependence graph after optimization. For clarity, the figure omits address calculations and loop control operations. Suppose we wish to compile this code segment for execution on the L-machine. The inner loop contains no cyclic dependences, which makes it a prime candidate for software pipelining. In general, an absence of dependence cycles allows a software pipeline to hide operation latency, and the target processor's available resources dictate performance. For this loop, the L-machine's floating-point units constitute the performance bottleneck. Assuming single-cycle throughput for all operations, each loop iteration requires 23 cycles to execute the 46 floating-point operations.

As an alternative to a strictly ILP-based approach, we might choose to vectorize the loop. An absence of dependence cycles also means the loop is fully vectorizable. A traditional vectorizing compiler would replace all operations in Figure 1-10 with the corresponding vector instructions. We could then software pipeline the vector loop to hide vector operation

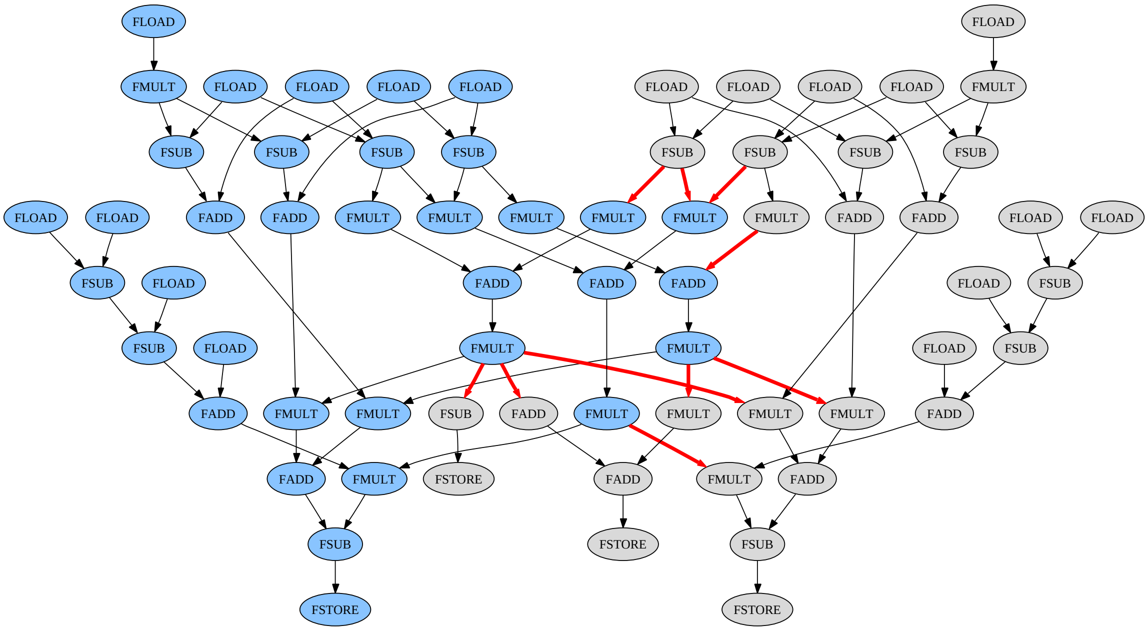


Figure 1-10: Data-dependence graph for the primary loop in `tomcatv`. Highlighted nodes are those chosen for vectorization using the selective vectorization algorithm described in this thesis. Highlighted edges indicate dataflow between vector and scalar operations.

latency. The L-machine's single vector unit requires 46 cycles to execute the 46 floating-point operations. For a vector length of 2, one iteration of the vector loop executes two iterations of the original loop. As a result, traditional vectorization combined with software pipelining achieves an average initiation interval of 23 cycles, matching the performance of software pipelining alone.

Figure 1-10 also shows the results of selective vectorization using the algorithm described in this thesis. Shaded nodes represent those chosen for vectorization; the remainder execute on scalar resources. Figure 1-11 summarizes the workload of the relevant processor resources resulting from the competing compilation techniques. The operation allocation of Figure 1-10 requires 26 cycles to execute each vector iteration, reduced from 46 cycles. This improvement in schedule length represents a speedup of 1.77× over the conventional techniques.

Technique	MEM	FPU	VEC	II
Software pipelining	11	23	0	23
Traditional vectorization	13	0	46	23
Selective vectorization	26	20	26	13

Figure 1-11: Resource usage and resulting initiation interval of competing compilation techniques.

In Figure 1-10, highlighted edges indicate dataflow between vector and scalar operations. Since the L-machine does not provide a specialized network for transferring operands, this dataflow requires that we transmit operands through memory[2]. Despite the high cost, selective vectorization achieves a significant performance improvement. One reason for this success is that vectorization of memory operations reduces the workload of the memory units, freeing them for operand communication. In addition, software pipelining hides the latency of transmitting operands through memory.

To produce an efficient schedule, selective vectorization faces two major difficulties: accurately predicting the performance impact of different vectorization strategies, and using this information to compute an efficient schedule and resource allocation. The technique presented in this thesis overcomes these obstacles using two approaches. First, I formulate selective vectorization in the context of software pipelining. Software pipelining provides a framework for exploiting a processor's vector and ILP capabilities simultaneously. Additionally, it allows the selective vectorization algorithm to ignore the latency of vectorizable operations and focus on overall resource requirements. Second, selective vectorization operates on a low-level intermediate representation. This approach, which contrasts with traditional vectorization, allows the algorithm to accurately predict performance on a specific target architecture.

1.4 Contributions

The objective of this thesis is to develop compiler technology that targets short-vector instructions automatically and efficiently. Automatic compilation allows for increased performance while hiding the details of the target architecture from the high-level programmer. Toward this end, I have made contributions in two major areas. The first addresses alignment of vector memory references. I am one of the first researchers to employ compiler techniques that reduce the overhead of misaligned references. Within this area,

- I am the first to propose and devise a static analysis to extract alignment information at compile-time. I formulate the analysis as a dataflow problem. When available, static alignment information allows the compiler to select more efficient, aligned vector memory opcodes.

[2]Note that an operation with more than one consumer requires a maximum of one communication sequence since transferred operands are available to all consumers.

- I am the first to use profiling to guide transformations that enforce aligned references at runtime. The technique is essential in eliminating unnecessary code expansion. I show that profiling is practical for the alignment problem since alignment characteristics vary little between data sets. On a set of SPEC FP benchmarks, my methods are able to enforce alignment for roughly 75% of dyanmic memory references.

- I present a comprehensive description of the breadth of alignment research.

My second major contribution addresses performance from the perspective of vector code selection. In this field,

- I am the first to suggest that automatic compilation should specifically target a processor's scalar and vector resources simultaneously in order to obtain top performance.

- I have formulated *selective vectorization*, a technique that leverages software pipelining to provide a balanced resource allocation. An important aspect of selective vectorization is its ability to manage operand transfer between scalar and vector operations. Even when this communication is expensive, the technique is sufficiently sophisticated to achieve large performance gains.

- I have developed and implemented a practical selective vectorization algorithm by adapting a two-cluster partitioning heuristic. The approach is applicable to any architecture that provides a combination of ILP and short-vector hardware.

- I have devised *modulo scalar expansion*, a code generation technique which leads to efficient software pipelines on architectures that provide no hardware support for communicating operands between vector and scalar operations.

- I am the first to advocate vectorization on a low-level intermediate representation. This method contrasts with traditional vectorization which operates on source-level representations. A back-end approach is crucial for predicting performance on a specific target architecture.

- I have evaluated selective vectorization using a realistic architectural model. For a set of SPEC FP benchmarks, the technique achieves whole-program speedups of up to $1.35\times$ over existing approaches. For individual loops, selective vectorization provides speedups of up to $1.75\times$.

1.5 Thesis Outline

The inspiration for this thesis is the untapped potential of short-vector instructions prevalent in general-purpose and embedded processors. Chapter 2 overviews modern multimedia instruction sets and traces their development. It also identifies the major challenges in targeting multimedia instructions automatically in the compiler.

Chapter 3 reviews two established compilation techniques for extracting fine-grain parallelism: traditional vectorization and modulo scheduling. These methods are important for the discussion of selective vectorization since the approach leverages prior research in both areas. Traditional vectorization provides a framework for identifying data parallelism, and modulo scheduling creates the low-level schedules of selectively vectorized loops.

Chapter 4 describes the specific algorithm I have developed to perform selective vectorization. It also discusses the complications of vectorizing a low-level IR, and describes the methods by which my implementation overcomes these issues. A key component of selective vectorization is its efficient handling of operand transfer between vector and scalar operations. This ability is especially important for architectures that do not provide direct support for operand communication. Chapter 4 develops a code generation technique that is crucial for minimizing the overhead of transferring operands through memory. The conclusion of Chapter 4 discusses prior research that is related to selective vectorization.

Chapter 5 describes the alignment restriction placed on vector memory operations, and develops a suite of static and dynamic techniques that reduce misalignment overhead. In addition to my own contributions, Chapter 5 integrates the work of others in order to provide a comprehensive overview of alignment research.

Chapter 6 evaluates selective vectorization on a realistic VLIW architecture with short-vector extensions. It also examines the performance impact of hardware support for misaligned references and operand communication. Finally, Chapter 7 identifies limitations in my approaches and outlines areas for future work.

Chapter 2

Multimedia Extensions

This chapter describes the multimedia extensions available in today's general-purpose processors. Section 2.1 provides an overview of the broad features that comprise modern extensions. Section 2.2 describes the development of multimedia instructions through their introduction in popular commodity processors. Section 2.3 identifies the major challenges multimedia extensions present to the compiler writer and reviews the predominant compilation techniques currently available.

2.1 Overview

Most of today's general-purpose processors implement a set of short-vector instructions. Initially, designers introduced these extensions to improve performance in multimedia applications such as image processing, audio and video encoding, 2D and 3D graphics, speech processing, and digital signal processing. These codes generally contain compute-intensive kernels that process long streams of data. To match this structure, multimedia architectures adopt a single instruction multiple data (SIMD) paradigm, which applies the same opcode to a vector of operands. Figure 2-1 shows an example of SIMD execution.

Vector instructions provide a means for parallel execution without the costs associated with a conventional ILP design. For example, vector instructions require no bypassing within a vector operand since all elements are independent. In addition, vector operations require fewer register file read and write ports to supply an equivalent number of functional units. The simplicity of this design is significant because the complexity and size of the register file and bypassing network grow quadratically with the number of functional units.

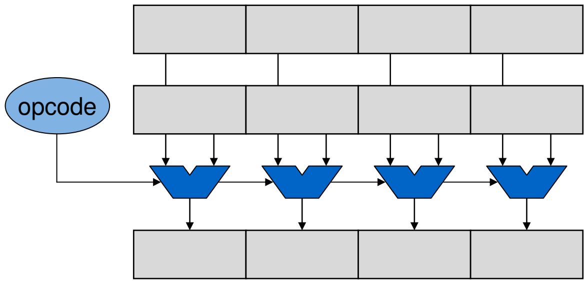


Figure 2-1: SIMD execution of packed data.

Vector instructions also offer a more compact instruction encoding which improves instruction fetch efficiency. Vector memory operations are particularly powerful since they transfer multiple operands with a single instruction.

Vector execution is typically associated with supercomputers [26, 33]. Contemporary multimedia extensions differ from traditional vector processors in three major areas. First, multimedia extensions implement relatively short vector lengths. Current designs *pack* operands in existing datapaths, meaning that the datapath width and element type determine the effective vector length. For example, a 64-bit word could contain two 32-bit elements, four 16-bit elements, or eight 8-bit elements. This short vector length allows for a design that computes all elements in parallel. In contrast, the long vectors provided by vector supercomputers typically require iterative execution. Second, multimedia extensions usually enforce aligned vector memory references. In other words, n-byte loads and stores must access data on n-byte boundaries. In most designs, direct access to misaligned regions is illegal and it is the compiler's responsibility to merge data in two aligned regions. Processors that support misaligned references directly operate more efficiently when accessing aligned segments [41]. Finally, today's multimedia extensions do not offer complete vector instruction sets. Noticeably absent are strided and scatter/gather memory operations for accessing non-contiguous data.

The multimedia instructions available in contemporary general-purpose processors fall into the following categories:

- Memory operations, which transfer vectors of data or individual vector elements. Designs that disallow misaligned accesses typically provide specialized instructions to facilitate merging of two aligned regions. Some extensions also provide memory prefetch and non-caching store instructions to facilitate access of regular data streams.

- Data rearrangement operations, which combine elements from two vectors or rearrange elements within a vector register. A common operation interleaves elements from two sources. Some multimedia extensions provide the ability to arbitrarily permute elements within a vector register.

- ALU operations, which execute vector versions of familiar scalar operations such as addition, multiplication, shift, and compare. Some extensions also provide operations to convert between integer and floating-point data types.

- Kernel-specific operations, which accelerate particular code sequences. For example, some multimedia extensions provide a *sum of absolute difference* operation, which is useful in MPEG encoding.

The composition of a multimedia instruction set varies among implementations. Early designs were modest in terms of the number and type of instructions they provided. Over time, multimedia extensions have evolved to offer fairly rich vector instruction sets. The latest designs support wide integer and floating-point data types, making them suitable for more general-purpose computation.

2.2 Historical Perspective

In this section, I trace the evolution of multimedia extensions through their introduction in commodity processors. To my knowledge, the first design to provide SIMD instructions operating on packed data was the Lincoln Labs TX-2 [3], completed in 1958. This processor implemented instructions to operate on a single 36-bit data word, two 18-bit values, or four 9-bit values. The ISA also provided an extensive set of permutation operations. The first modern microprocessor offering short-vector instructions was the Intel i860, introduced in 1989 [48]. The i860's instructions facilitated shading and hidden-surface removal for 3D graphics. Compared to more recent instruction sets, the i860's graphics operations were heavily specialized. This chapter focuses on extensions that are more applicable to general-purpose computation.

Figure 2-2 provides the high-level details of some representative multimedia extensions. Other designs worth noting include the MVI extension in the Alpha architecture [14] and the MDMX extension in MIPS [63]. The processors represented in Figure 2-2 are super-

Vendor	HP	Sun	Intel	AMD	Motorola + IBM	Intel
Extension	MAX-1/2	VIS	MMX	3DNow!	VMX	SSE/2/3
Year introduced	1994/'96	1995	1996	1998	1999	1999
Vector length (bits)	32/64	64	64	64	128	128
Element width (bits)	16	16/32	8/16/32	32	8/16/32	8/16/32/64
Vector register file	GP	FP	FP	FP	Separate	Separate
Misaligned accesses	Software	Software	Hardware	Software	Software	Hardware
ALU ops	Limited	Yes	Yes	n/a	Yes	Yes
Saturating arithmetic	Yes	No	Yes	n/a	Yes	Yes
Compare ops	No	Yes	Yes	Yes	Yes	Yes
Pack/unpack ops	No	Yes	Yes	n/a	Yes	Yes
Rearrangement ops	I+P	I	I	No	I+P	I
Floating-point ops	No	No	No	Yes	Yes	Yes

Figure 2-2: Relevant features of popular multimedia extensions. **I**=interleave, **P**=permute, **GP**=general-purpose register file, **FP**=floating-point register file.

scalar designs, but short-vector instructions are also present in many VLIW architectures. For example, Intel's IA-64 instruction set contains a set of short-vector operations [40]. Multimedia instructions also appear in many embedded processors. Examples include the TI TigerSHARC [28], the Microunity Mediaprocessor [32], Chromatic's Mpact 1 [44] and Mpact 2 [71], Philips Trimedia [92], and the MAP1000A Mediaprocessor [8].

MAX-1 and MAX-2

In 1994, HP introduced the MAX-1 extension in the PA-7100LC [47]. MAX-1 implements five vector instructions, all operating on 16-bit elements: add, subtract, shift-left-and-add, shift-right-and-add, and arithmetic mean. The shift-and-add operations allow for limited vector multiplication (*i.e.*, multiplication by a constant). In addition to standard modulo arithmetic, the instruction set provides saturating add and subtract opcodes, which image processing kernels often use for clipping.

In 1996, HP introduced the MAX-2 instructions in the PA-8000 [58]. At 64 bits, the PA-8000 datapath is twice the width of its predecessor and supports vectors of four 16-bit elements. In addition to the MAX-1 instructions, MAX-2 provides two data rearrangement operations. The *mix* instruction interleaves elements from two registers, and the *permute* instruction arbitrarily permutes the elements in one register.

In the MAX designs, vector instructions operate on data packed in integer registers. By reusing an existing register file, designers avoided the need to upgrade the operating system to save and restore additional processor state. By using the *integer* registers, several vector operations are available in existing opcodes, including loads, stores, cache prefetch, and bitwise logical operations. Also, since all memory operations are subject to alignment constraints, the PA-RISC instruction set provides a concatenate-and-shift instruction to facilitate access of misaligned data.

VIS

In 1995, Sun introduced the VIS instruction set in the UltraSparc I [90]. Other than the lack of saturating arithmetic, VIS implements a more extensive set of vector operations than MAX-2. For example, the extension provides a full 16-bit multiplication. VIS also includes a set of vector compare instructions which generate vector masks. These masks can be used to control the elements written to memory on a vector store, thereby providing a mechanism for vectorizing instructions in control flow.

Like MAX-2, VIS implements a total vector length of 64 bits. In addition to 16-bit vector instructions, VIS also provides several instructions that operate on 32-bit elements. 32-bit multiplication is absent, however. VIS also provides *pack* and *unpack* operations for converting between 8- and 16-bit data.

In order to avoid modification of the integer pipeline, Sun chose to implement VIS in the floating-point pipeline, with vector instructions accessing floating-point registers. Designers felt this strategy would free the integer pipeline for address arithmetic and branching [90]. The disadvantage is that VIS required a new set of opcodes to support vector logic operations. In addition, mixed floating-point and vector computation is potentially less efficient since these groups compete for resources.

MMX

Intel introduced the MMX extension to IA-32 in 1996 [69]. MMX is slightly more extensive than VIS. For example, MMX provides instructions for operating directly on vectors of 8-bit elements rather than requiring conversion to 16 bits. Additionally, MMX implements a set of saturating arithmetic operations.

Compared to RISC architectures, the IA-32 design exhibits some complexities that propagate to its multimedia instructions. For example, IA-32 implements a two-operand destructive instruction encoding. Also, IA-32 ALU operations can reference memory locations directly. In this case, the ISA requires that addresses are 64-bit aligned. Explicit load and store instructions can access misaligned data, but incur a performance penalty when doing so. Like VIS, MMX uses the floating-point register file to store vector operands. The wider floating-point registers allow for a 64-bit extension on a 32-bit architecture.

3DNow!

AMD introduced the 3DNow! extension in the K6-2 processor in 1998 [68]. 3DNow! is a floating-point extension primarily intended to improve performance for 3D graphics applications. The extension reuses the IA-32 floating-point registers to store vectors of two 32-bit single-precision values. Vector floating-point instructions consist of comparison, conversion, and standard arithmetic operations. To reduce complexity, the divide and square root operations use reciprocal approximation. The ISA implements additional instructions to perform explicit Newton-Raphson iteration for higher precision.

3DNow! floating-point operations do not fully comply with the IEEE 754 standard. For example, vector instructions do not cause exceptions, do not support NaNs or infinities, and implement only one rounding mode. Nevertheless, the introduction of vector floating-point operations provides new opportunities for program acceleration.

AltiVec/VMX

In the late '90s, Motorola, IBM, and Apple jointly developed a short-vector extension to the PowerPC [20]. The extension is termed AltiVec by Motorola, Velocity Engine by Apple, and VMX by IBM. AltiVec first appeared in the Motorola MPC 7400 in 1999. IBM provides VMX in the PowerPC 970 [36] and the PowerPC core of the CELL processor [23].

VMX represents a major advance in short-vector instruction sets. In the design of earlier extensions, architects cite size and complexity as primary considerations. For example, VIS represents a 3% increase in the die area of the UltraSparc I [90], and MAX-2 accounts for less than 0.1% of the PA-8000 [58]. Overall, the PA-8000 required only 3.8 million transistors [53], compared to 58 million in the PowerPC 970 [78]. Rapidly increasing transistor counts allowed designers to commit a larger portion of the chip to vector resources. Therefore, VMX is the first extension to introduce a separate vector register file. It also extends the vector length to a total of 128 bits.

VMX provides an extensive vector instruction set with vector versions of most integer and floating-point operations. With a four-operand instruction format, it also supports operations such as *multiply-add*. Integer division and 32-bit integer multiplication are absent, however, and VMX does not support 64-bit vector elements. As with 3DNow!, VMX implements floating-point division and square root with reciprocal approximation. The extension's floating-point operations are compatible with the Java subset [29] of IEEE 754 [39]. VMX also supports a non-Java mode which provides lower latency execution.

VMX supplies extremely flexible data rearrangement functionality with the *vperm* instruction, which can combine elements of two source registers in any permutation. The instruction also facilitates explicit software realignment for misaligned memory references. For vectorization of operations in control flow, VMX offers an extensive set of vector compare operations. In VMX, vector compare instructions produce control operands for the *select* instruction, which copies elements from one of two source operands.

SSE

Intel introduced the Streaming SIMD Extensions (SSE) to the Pentium III in 1999 [72]. Like 3DNow!, SSE is primarily a floating-point extension. SSE introduces a separate vector register file and implements a vector length to 128 bits. In 2001, Intel added SSE2 to the Pentium 4 [34]. This extension provides a more powerful alternative to MMX with vector integer instructions which operate on the 128-bit SSE registers.

SSE and SSE2 combine to form an extensive vector instruction set. In contrast to VMX, they implement vector instructions that operate on 64-bit elements, including support for double-precision floating-point data. SSE2 is fully compliant with the IEEE 754 standard [39]. IA-32 implementations mitigate this complexity by executing 64-bit calculations as two sequential operations. On the Pentium 4, vector and scalar floating-point instructions use the same execution resources.

SSE and SSE2 provide aligned and unaligned versions of vector memory operations. The aligned versions are more efficient, but are only valid when the compiler can guarantee alignment. My experiments reveal that a simple loop copying one array to another executes roughly 2× faster on a Pentium 4 when using aligned versus unaligned opcodes. Unaligned vector memory accesses are particularly expensive if data cross a cache line boundary. In this case, a series of short-width accesses are more efficient [10]. Designers removed this anomaly in 2004 with the introduction of SSE3 and a more efficient 128-bit vector load. SSE3 also provides *horizontal* operations to sum the elements within a vector register.

2.3 Compilation for Multimedia Extensions

In order to gain widespread use, multimedia extensions must be available to high-level programmers. Initially, access to multimedia instructions was limited to inline assembly, macro calls, or specialized library routines. These approaches are tedious and error-prone, require in-depth knowledge of the underlying architecture, and are not portable. The ideal solution leverages compiler technology to target multimedia extensions automatically. This section identifies the major challenges faced by the compiler and overviews current approaches.

Identifying Data Parallelism

In order to utilize vector instructions, data parallelism must exist in the target application. When present, the compiler must identify it. The accurate characterization of memory dependences is the most difficult component of this process. Fortunately, dependence analysis is a key step in any parallelization technique and is backed by an extensive literature. Chapter 3 gives an overview of data-dependence theory.

Vectorizing technology originally targeted Fortran, the language of choice for scientific applications. Today, most programmers prefer C or C++ for developing multimedia applications. Since these languages allow pointer aliasing, the compiler must employ sophisticated methods to acquire accurate dependence information. A popular solution extends the language with directives that allow the user to convey dependence information (*e.g.*, the `restrict` keyword in C99 [43]). The disadvantage of this approach is that it imposes an extra burden on the programmer. An alternative is to employ an interprocedural alias analysis. Unfortunately, whole-program analyses are not practical for large applications and cannot extract information from libraries whose source code is unavailable. Perhaps the most practical solution is the use of runtime dependence checks. The Intel compiler takes this approach when targeting the MMX and SSE extensions [10].

Code Selection

Most of today's multimedia extensions do not have complete vector instruction sets. The absence of vector opcodes complicates code selection and code generation since the compiler must emit scalar operations instead. Particularly troublesome is the absence of strided and scatter/gather memory operations. IA-32 provides opcodes to move 64-bit elements

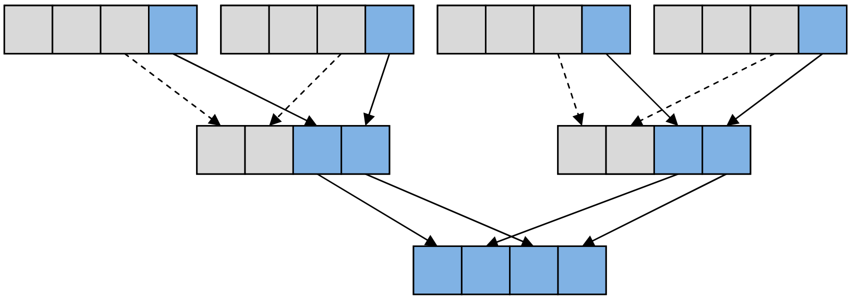


Figure 2-3: Strided load using interleaving operations.

between memory and vector registers. These instructions make imitation of scatter/gather operations fairly simple for 64-bit data. The architecture does not provide the same support for all data types, however. Figure 2-3 illustrates the software emulation of a strided load of a 4-element vector using four scalar loads and three interleaving operations. The scheme assumes the ability to load scalar values into the low-order element of a vector register. Unfortunately, not all designs provide this functionality. VMX provides no specialized support for loading scalar values into specific vector elements. In order to use non-contiguous data in vector calculations, the compiler must rearrange values in memory or use a complex sequence of permute operations.

Another factor complicating code selection is the availability of computation-specific instructions, such as saturating arithmetic. For example, consider the following code sequence:

```
unsigned char a[N], b[N], c[N];
unsigned short tmp;
...
for (i=0; i<N; i++) {
  tmp = b[i] + c[i];
  if (tmp > 0xFF)
    tmp = 0xFF;
  a[i] = tmp;
}
```

When vectorizing this loop, the compiler could employ a saturating vector add opcode, eliminating the explicit overflow test. Discovering these opportunities typically involves matching specific patterns and replacing the computation with an optimized code sequence. Unfortunately, this process is inherently unsystematic and difficult to make robust. Bik [10] describes a number of idiom recognition optimizations in the Intel compiler.

Profitability of Vectorization

An effective compiler should provide consistent and efficient performance on the target architecture. Compilation for multimedia extensions is complicated by the fact that vectorization can actually lead to a performance loss. This phenomenon has three primary causes. The first is a lack of an appropriate vector opcode. In this case, scalar emulation of a vector instruction may have an overhead that negates the benefit of vectorization. An important example is the absence of scatter/gather operations and the cost of assembling vector operands from non-contiguous memory.

Second, partially vectorizable loops often require communication between vector and scalar instructions. Current multimedia extensions implement separate scalar and vector register files. In these designs, transmission of operands between register files requires explicit instructions. Intel's IA-32 extensions provide support for moving data between scalar and vector registers. Unfortunately, accessing individual elements typically requires shift or permute operations. VMX provides no underlying support and the compiler must transfer data through memory using a series of memory instructions.

Finally, blind vectorization fails to account for a processor's scalar resources. On targets that provide a wealth of scalar units, some loops may execute more efficiently in scalar form. In order to make this determination, the compiler must have an accurate model of the target architecture's overall processing capabilities. Traditional approaches operate in the front-end on a source-level intermediate representation. With this method, the compiler must predict how its decisions will affect performance after the code is lowered and optimized. The selective vectorization algorithm presented in Chapter 4 operates in the compiler back-end, where it can more accurately measure the performance of vectorization alternatives.

Current Approaches

Most approaches for automatic compilation employ techniques developed for vector supercomputers [10, 15, 19, 24, 65, 66, 86, 98]. Commercial products targeting multimedia extensions include the Intel compiler [10], the IBM XL compiler [38], the VAST/AltiVec compiler [93], and VectorC [94]. At a high level, these approaches directly adopt the traditional vectorization strategy I review in Chapter 3. The idiosyncrasies of modern multimedia instruction sets require special attention, however. One important issue is the alignment of vector memory operations, which has received considerable attention in the literature. Chapter 5 discusses the alignment problem in detail. Bik [10] gives an excellent account of an industrial multimedia compiler in his description of the Intel compiler.

In some cases, straightline code can exhibit sufficient parallelism to warrant the use of short-vector opcodes. In [55], I proposed a method to automatically extract data parallelism from basic blocks. The scheme exposes loop-level parallelism by unrolling. Shin *et al.* [84, 83] extended these ideas to reduce memory references and to target instruction sequences with control flow. Kudriavtsev *et al.* [52] also targeted parallelism within a basic block with an emphasis on minimizing data reshuffling operations. Recently, Wu, Eichenberger, and their colleagues [98] developed a framework for exploiting data parallelism both within straight-line code and across loop iterations.

Chapter 3

Background

This chapter reviews two classic techniques for extracting fine-grain parallelism: vectorization and software pipelining. These topics are important for developing the selective vectorization algorithm in Chapter 4. Researchers first developed vectorizing technology [6, 96] for vector supercomputers such as the Cray-1 [79]. More recently, compiler engineers have adopted it for compilation to multimedia extensions [10, 15, 19, 24, 65, 66, 86, 98]. Vectorization identifies and extracts data parallelism, which implies the ability to safely execute the same operation on multiple data elements concurrently. Sections 3.1 and 3.2 discuss automatic vectorization.

Section 3.3 overviews software pipelining, a method for exploiting instruction-level parallelism (ILP), or parallelism among machine instructions. This thesis focuses on *modulo scheduling*, which is a popular approach to software pipelining. In fact, Rau's iterative modulo scheduling heuristic [74] appears to be the basis of most existing software pipelining implementations. Iterative modulo scheduling was developed for one of the first VLIW machines, the Cydra 5 [18, 76]. Today, it is widely used for obtaining high performance on ILP processors.

Despite some similarities, the compiler community generally views vectorization and modulo scheduling as alternatives for extracting fine-grain parallelism. In the past, the target architecture determined the appropriate technique (*e.g.*, vector versus VLIW or superscalar). This distinction, however, is not appropriate for today's processors since they dedicate similar resources to both instruction-level and data-parallel hardware. While vector supercomputers also provide ILP resources, the processing power of the vector unit greatly outweighs that of the scalar hardware. For these machines, large-scale vectorization is the primary goal. The selective vectorization algorithm described in Chapter 4 extends aspects of vectorization and modulo scheduling to derive highly efficient schedules.

3.1 Vectorization

Vectorization converts a sequential loop into a parallel version that utilizes a processor's vector instruction set. This transformation involves a substantial reordering of operations; semantically, a vector instruction computes multiple values before committing any of the results. Vectorization is only legal if it preserves dependences in the original loop. As a result, it relies heavily on the ability to accurately characterize dependences among operations. A straightforward reaching-definitions dataflow analysis [1, 64] uncovers dependences between scalar variables. The primary difficulty, therefore, is the accurate identification of dependences among memory operations. A simple approach that conservatively assumes dependence between any load and store almost always prevents vectorization. Fortunately, memory dependence analysis is backed by an extensive literature. The texts by Allen and Kennedy [6] and Wolfe [96] provide overviews of established techniques.

Data dependence theory categorizes dependences as follows:

- *Flow dependences* occur between two operations op_1 and op_2 when op_1 writes a value which is read by op_2.
- *Antidependences* occur between two operations op_1 and op_2 when op_1 reads a location which op_2 subsequently overwrites.
- *Output dependences* occur between two operations op_1 and op_2 when op_1 writes a value which op_2 subsequently overwrites.

Dependence theory further classifies dependences in a loop as *loop-carried* or *loop-independent*, depending on whether they occur between operations in different iterations or the same iteration, respectively. For loop-carried dependences, the dependence *distance* specifies the number of loop iterations separating the dependent operations. For dependences in loop nests, it is often useful to associate a distance with each loop, giving rise to a *distance vector*. For example, the following loop nest contains a loop-carried flow dependence from the store to the load with the distance vector $(1, 0, 3)$, indicating a dependence distance of 1 for the outer loop, a distance of 0 for the middle loop, and a distance of 3 for the inner loop:

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      a[i+1][j][k+3] = a[i][j][k];
```

Many loop transformations characterize dependences with *direction vectors*, which we derive from the distance vectors. Given a distance d_i associated with the loop at nesting level i, we define the direction D_i as

$$D_i = \begin{cases} \text{'}<\text{'} & \text{if } d_i > 0 \\ \text{'}=\text{'} & \text{if } d_i = 0 \\ \text{'}>\text{'} & \text{if } d_i < 0 \end{cases}$$

The loop above has a direction vector $(<, =, <)$. Since selective vectorization targets inner loops, it can safely ignore dependences that are not of the form $(=, =, ..., <)$ or $(=, =, ..., =)$. The former represent dependences carried by the inner loop, and the latter represent loop-independent dependences. The vector $(=, =, ..., >)$ is impossible since it implies executing the sink before the source.

Memory dependence analysis is most successful when analyzing a source-level representation where it can readily characterize loop bounds and array index functions. Traditional vectorization also operates at this level. In Chapter 4, I show that low-level vectorization is also possible as long as the compiler computes dependences early and preserves the information in the back-end. High-level memory dependence information is also extremely valuable in software pipelining [74].

After identifying dependences, the basic steps for vectorizing an inner loop are as follows [6, 96]:

1. Create a data-dependence graph for the loop body. Nodes in the graph represent statements in the inner loop and edges denote dependences between statements.

2. Identify cycles in the dependence graph using Tarjan's algorithm for strongly connected components [88]. Statements involved in a dependence cycle must execute sequentially. The rest are vectorizable.

3. Partition the graph into *piblocks* [6], where every piblock corresponds to a strongly connected component, and remove edges contained within piblocks. The resulting graph is acyclic.

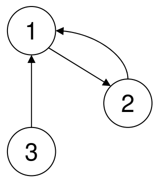
4. Perform a topological sort of the graph to determine a valid piblock ordering.

5. For each vectorizable node, emit a vectorized statement. Otherwise, construct a sequential loop to execute the operations in the piblock in original program order.

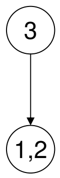
As an example, consider the following loop:

```
    for (i=0; i<N; i++) {
(1)   a[i]   = b[i] + c[i];
(2)   b[i+1] = a[i] + d[i];
(3)   c[i+1] = e[i] + f[i];
    }
```

The data-dependence graph is



Statements (1) and (2) form a strongly connected component and must execute sequentially. Statement (3) is vectorizable. Applying steps 2–4 yields



and step 5 produces

```
c[1:N+1] = e[0:N] + f[0:N];
for (i=0; i<N; i++) {
  a[i]   = b[i] + c[i];
  b[i+1] = a[i] + d[i];
}
```

Notice that the vector operations execute first, even though the originating scalar statement appears later in the source loop. This reordering results from the topological sort and is necessary to preserve the dependence from statement (3) to statement (1).

In order to execute a vectorized statement on actual hardware, the compiler must *strip-mine* vector operations to match the vector length. For example, stripmining the loop above for a vector length of 4 produces

```
for (i=0; i<N-3; i+=4) {
  c[i+1:i+4] = e[i:i+3] + f[i:i+3];
}
// Execute remaining iterations
for (; i<N; i++) {
  c[i+1] = e[i] + f[i];
}
for (i=0; i<N; i++) {
  a[i]   = b[i] + c[i];
  b[i+1] = a[i] + d[i];
}
```

The traditional vectorization algorithm isolates vector and scalar computation in separate loops. In some cases, this process creates a large number of distinct loops. To reduce the resulting overhead as much as possible, the compiler can perform *loop fusion* [6, 17, 45] to merge loops of the same type wherever dependences permit.

A particularly important issue in vectorization is the compiler's handling of scalar assignments. Consider the following loop:

```
     for (i=0; i<N; i++) {
(1)    s = a[i];
(2)    b[i] = s;
     }
```

The scalar `s` induces a flow dependence from statement (1) to statement (2), and a loop-carried antidependence from statement (2) to statement (1). The resulting cycle prevents vectorization even though the statements appear to be completely parallel. The flow dependence represents true dataflow between the statements. The antidependence, however, results from the reuse of `s` in every iteration. Vectorization can ignore the antidependence if it allocates a private copy of `s` to each iteration. Practically, the compiler should assign `s` to a vector register. This optimization is termed *scalar privatization*, and is legal whenever a scalar *use* is reached only by *definitions* in the same iteration [6].

In the loop above, scalar privatization enables full vectorization. When a scalar spans distributed loops, however, the optimization is not applicable, as the following example illustrates:

```
      for (i=0; i<N; i++) {
(1)     s = a[i] + b[i];
(2)     c[i+1] = c[i] + s;
      }
```

Even if we ignore the dependence cycle induced by `s`, the dependence from statement (2) to itself necessitates loop distribution. In this case, vectorization should employ *scalar expansion* [6] to communicate operands between the vector and scalar portions:

```
t[0:N] = a[0:N] + b[0:N];
for (i=0; i<N; i++) {
  c[i+1] = c[i] + t[i];
}
```

The disadvantage of scalar expansion is that it requires additional storage and address arithmetic. We can reduce the memory overhead by performing scalar expansion after stripmining:

```
for (i=0; i<N; i+=4) {
  t[0:3] = a[i:i+3] + b[i:i+3];
  for (j=0; j<3; j++) {
    c[i+1+j] = c[i+j] + t[j];
  }
}
```

Here, the temporary array requires only *vector length* elements. Furthermore, the short iteration count allows us to unroll the scalar loop completely, yielding the following:

```
for (i=0; i<N; i+=3) {
  t[0:3] = a[i:i+3] + b[i:i+3];
  c[i+1] = c[i+0] + t[0];
  c[i+2] = c[i+1] + t[1];
  c[i+3] = c[i+2] + t[2];
  c[i+4] = c[i+3] + t[3];
}
```

Since the size of `t` matches the vector length, it may be possible to replace the array with a vector register. Recall, however, that we first introduced the temporary array in order to communicate intermediate operands between scalar and vector operations. As discussed in Chapter 2, many multimedia architectures do not provide specialized hardware support for transferring data between vector and scalar registers. In this case, the communication overhead is unavoidable.

An important characteristic of the loop form above is that it preserves ILP since scalar and vector computation occupy the same loop body. The selective vectorization algorithm described in Chapter 4 also takes this approach. In contrast to traditional vectorization, however, selective vectorization generates the optimized loop directly in the back-end, rather than through a series of high-level transformations.

3.2 Superword-Level Parallelism

Vector supercomputers are most efficient when operating on long vectors. Practically, large-scale parallelism is available across loop iterations. The short vectors supported by contemporary multimedia architectures, however, make it profitable to search for data parallelism in other regions, namely, within basic blocks. To differentiate data parallelism inside a basic block from that exploited by vector supercomputers, I proposed *superword-level parallelism*, or SLP in [55]. This research introduced an algorithm which, for the first time, made it possible to identify and exploit vector parallelism in code segments other than loop nests.

SLP may result from programmer unrolling of data-parallel loops. Often, it is present in multimedia codes that perform the same operation on the red, green and blue components of a pixel, as in alpha blending:

```
for (i=0; i<N; i++) {
  blend[i].r = a * fore[i].r + (1-a) * back[i].r;
  blend[i].g = a * fore[i].g + (1-a) * back[i].g;
  blend[i].b = a * fore[i].b + (1-a) * back[i].b;
}
```

In cases where data parallelism is available across loop iterations, unrolling converts loop-level parallelism to superword-level parallelism. Consider the following:

```
for (i=0; i<N; i++) {
  a[i] = b[i] + c[i];
}
```

Unrolling this loop by a factor of 2 creates vector parallelism within the loop body:

```
for (i=0; i<N; i+=2) {
  a[i+0] = b[i+0] + c[i+0];
  a[i+1] = b[i+1] + c[i+1];
}
```

Loop unrolling allows a single technique to extract parallelism at either level. When data parallelism exists in straight-line code, Bik [10] advocates the reverse approach of forming a loop before employing traditional vectorization. Recently, Wu, Eichenberger and colleagues [23, 98] developed a framework which extracts data parallelism from both dimensions.

To exploit SLP, I proposed a simple greedy heuristic [55]. The algorithm locates groups of isomorphic expressions and replaces them with vector opcodes. Isomorphic operations are vectorizable when they are independent. Therefore, the first step of the algorithm identifies dependences. Data dependence analysis is potentially simpler for SLP than for traditional vectorization since the former can ignore loop-carried dependences. Practically, the implementation reported in [55] employed a traditional analysis.

Independence of a group of isomorphic statements does not allow vectorization in all cases. Consider the following loop:

```
for (i=0; i<N; i++) {
  a[i+1] = b[i];
  b[i+1] = a[i];
}
```

Assume that we target an architecture with a vector length of 2. Unrolling by a factor of 2 produces:

```
    for (i=0; i<N; i+=2) {
(1)   a[i+1] = b[i];
(2)   b[i+1] = a[i];
(3)   a[i+2] = b[i+1];
(4)   b[i+2] = a[i+1];
    }
```

Since there is no chain of dependences from statement (1) to statement (3), a single vector statement can execute both. The same is true for statements (2) and (4). Combining both pairs results in the following:

```
for (i=0; i<N; i+=2) {
  a[i+1:i+2] = b[i:i+1];
  b[i+1:i+2] = a[i:i+1];
}
```

Unfortunately, the dependence cycle in the original loop means that full vectorization is invalid. The algorithm presented in [55] arbitrarily unvectorizes operations to remove cycles, leading to partial vectorization not seen in traditional approaches. For example, the SLP heuristic might produce the following:

```
for (i=0; i<N; i+=2) {
  b[i+1] = a[i];
  a[i+1:i+2] = b[i:i+1];
  b[i+2] = a[i+1];
}
```

To determine when vectorization is profitable, superword-level parallelization employs an unsophisticated cost metric. Simply, the algorithm calculates the number of instructions needed to execute a group of operations in scalar versus vector form. The heuristic considers each group in isolation, and prefers vector opcodes whenever they decrease the instruction count. As the algorithm combines operations, it reduces the cost of communicating vector operands to successors, creating additional opportunities for vectorization. In order to initiate the process, the algorithm assumes that it is always profitable to vectorize unit-stride memory operations. This may not be the case, however, as the following loop illustrates:

```
for (i=0; i<N; i++) {
  a[i+1] = a[i] + b[i];
}
```

The load from `b` is vectorizable, but the loop-carried dependence forces the remaining operations to execute sequentially. If we vectorize the load, the data must be transferred directly to a scalar addition. In architectures that require communication of these operands through memory, the vector load is useless overhead. Although it is possible to identify these situations, the cost model presented in [55] has a more fundamental inefficiency in that it does not accurately measure the performance impact of its decisions. A more effective approach is the topic of Chapter 4.

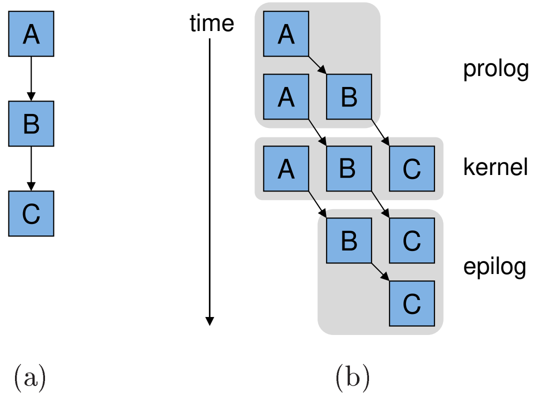


Figure 3-1: (a) Blocks of operations with data dependences between them. (b) Concurrent execution using software pipelining.

3.3 Software Pipelining

Software pipelining is a static scheduling technique for overlapping iterations of a loop. The technique takes advantage of ILP hardware to execute operations from multiple iterations concurrently. The term *software pipelining* derives from the observation that a transformed loop consists of a number of *stages* with computation flowing between stages akin to a hardware pipeline.

Figure 3-1 shows an illustration of software pipelining. The blocks in part (a) represent the computation in an inner loop with each block containing one or more machine instructions. Edges between blocks represent data dependences. If we schedule the loop in a traditional manner (*e.g.*, using list scheduling), the blocks must execute sequentially in order to satisfy dependences. Part (b) illustrates parallel execution with a software pipeline, with time progressing from top to bottom. The schedule preserves dependences between blocks since each stage operates on a different iteration. The *kernel* executes repeatedly and forms the steady-state of the software pipeline. The *prolog* and *epilog* fill and drain the pipeline, respectively.

Most existing software pipelining algorithms actually generate *modulo schedules*. A modulo schedule is an ordering for the operations in an inner loop such that repetition of the kernel at a regular interval results in no dependence violations or resource conflicts. The *initiation interval*, or II, is the schedule length of the kernel. In general, a lower II results in higher performance[1].

[1] A lower initiation interval tends to increase the length of the prolog and epilog. For loops with low iteration counts, the epilog and prolog can dominate execution time and actually degrade performance.

A loop's minimum initiation interval is limited by available machine resources and recurrences in the dependence graph. Resources limit the II in an obvious way. For example, executing a loop with 6 *add* operations on a machine with 2 ALUs requires at least 3 cycles. Recurrences limit the II because an operation cannot issue until its dependences from previous iterations are satisfied. In a loop with a dependence cycle from an operation to itself d cycles later, the total latency of the cycle must be less than or equal to $II \times d$ in order to ensure correct operation [74].

Rau's iterative modulo scheduling heuristic [73, 74] forms the basis of many software pipelining implementations, and is used in the evaluation in Chapter 6. Iterative modulo scheduling attempts to find a valid schedule at a given II. The high-level steps of the algorithm are as follows:

1. Calculate the theoretical minimum initiation interval.
2. Select an operation for scheduling.
3. Locate a valid time slot and resource binding.
4. Eject conflicting operations from the partial schedule.
5. Iterate until all operations are scheduled.
6. If scheduling fails, increase the II and start over.

To avoid scheduling at infeasible initiation intervals, the first step calculates the minimum II. Rau describes methods for computing the minimum II due to resources (ResMII) and recurrences (RecMII). The theoretical minimum II is the maximum of these two values. With this as a starting point, the algorithm iteratively selects unscheduled operations from a worklist according to a priority function. Many cost functions appear in the literature. Usually, they give precedence to operations on the critical path or operations in a dependence cycle.

Iterative modulo scheduling attempts to schedule each operation at the earliest possible time slot. A time slot is valid if it is late enough to satisfy dependences with scheduled predecessors, and contains sufficient resources to execute the operation. A *modulo reservation table*, or MRT [54] tracks resource usage. The MRT is a two dimensional table with time slots 0 to $II{-}1$ on one dimension and a list of machine resources on the other. Figure 3-2

	ALU 1	ALU 2	FPU	MEM
0				
1		add		
2				
3				

Figure 3-2: Example modulo reservation table.

shows an example. Resources in the table represent everything the compiler must consider for correct and efficient execution. For some machines, this could consist simply of issue slots. At a minimum, however, it is usually necessary to track functional-unit usage. The MRT in Figure 3-2 tracks the usage of four resources. For some architectures, the compiler may be responsible for managing low-level details such as register ports and operand networks.

Starting at the earliest time required by dependences with scheduled predecessors, the algorithm searches consecutive time slots for an allocation that does not conflict with existing reservations. On some machines, operations can execute on one of several units. In this case, the algorithm considers each alternative before moving to the next time slot. When the heuristic finally schedules an operation at time t, it records the resource usage in the MRT at $t \bmod II$. For example, an `add` operation assigned to the second ALU at cycle 5 occupies the highlighted slot in Figure 3-2. This placement captures the fact that operations in a modulo schedule initiate at regular intervals of II cycles. During scheduling, the algorithm must consider a maximum of $II-1$ consecutive time slots since the available resources at time n are the same as those at time $n+II$.

After scheduling an operation, the algorithm checks the issue time of any scheduled successors. It removes any operations that are scheduled too early to preserve dependences and reinserts them into the worklist. If an open slot is not available, the algorithm backtracks by ejecting one or more scheduled operations. Rau's formulation inserts the operation under consideration in the earliest time slot that preserves dependences with predecessors. The heuristic then removes any operations with resource conflicts and returns them to the worklist. This process continues for a predetermined number of steps. If the process fails to locate a valid schedule, the algorithm increases the II and repeats. After computing a valid schedule, the compiler generates the prolog and epilog [75], and performs register allocation [77].

Hardware Support for Modulo Scheduling

Modulo scheduling is applicable to any architecture providing ILP hardware. When available, however, it benefits from rotating registers and predication [18, 76]. Rotating registers provide a mechanism to queue multiple writes to the same nominal register. If they are unavailable in the target architecture, the compiler can employ unrolling and scalar renaming [54]. Predication provides a means for modulo scheduling loops with control flow. Combined with rotating registers, it also allows the same code to execute the kernel, epilog and prolog, thereby reducing code expansion. The Cydra 5 [18, 76] first introduced hardware assistance for modulo scheduling. Intel's IA-64 [62, 82] is a modern design which employs both predication and rotating registers.

Notable Extensions

Modulo scheduling is an extensive field of study. A complete overview of research in this area is beyond the scope of this thesis. There are several significant contributions worth mentioning, however. Lam introduced *modulo variable expansion*, a method for generating valid pipelines without rotating registers [54]. She also pioneered a method for modulo scheduling loops with control flow. Lavery and Hwu show how superblocks and hyperblocks facilitate scheduling of loops with control flow [57]. Several researchers have proposed methods to reduce register pressure in modulo scheduled loops [21, 35, 59]. This optimization is particularly important since excessive register pressure usually causes spilling and results in higher initiation intervals. Modulo scheduling is an NP-complete problem that requires a heuristic solution. Some researchers have investigated optimal methods [22, 30, 80]. While not realistic for a production compiler, such proposals provide a benchmark for the effectiveness of practical algorithms.

Chapter 4

Selective Vectorization

This chapter introduces *selective vectorization*, a compilation technique for targeting short-vector instructions in general-purpose architectures. The technique achieves higher performance than conventional approaches by utilizing a processor's vector *and* scalar resources as efficiently as possible. Selective vectorization also naturally uncovers those situations where full vectorization or no vectorization is most profitable.

Selective vectorization relies on the ability to accurately measure performance for a specific target architecture. To achieve this goal, the algorithm operates on a low-level intermediate representation, or IR. This approach contrasts to traditional vectorization which operates on a source-level representation. Section 4.1 discusses the complications introduced by a low-level IR, and the methods by which the compiler overcomes them.

Section 4.2 describes the specific algorithm I have implemented to perform selective vectorization. I adapt a min-cut partitioning heuristic described by Fiduccia and Mattheyses [27]. Section 4.3 gives a detailed example of the approach. The algorithm is applicable to architectures that export a simple interface to the compiler. Section 4.4 extends the method to support designs with arbitrary instruction set semantics.

When selective vectorization opts to employ vector opcodes, the compiler must generate a vectorized code sequence. A crucial aspect of code generation is the management of communication between vector and scalar instructions. Section 4.5 develops an efficient method for transferring data on architectures that do not provide a specialized communication network. Section 4.6 describes the overall code generation algorithm. Finally, Section 4.7 discusses prior research that is related to selective vectorization.

4.1 Intermediate Representation

Before describing the selective vectorization algorithm, it is important to discuss its input. Selective vectorization operates on a low-level intermediate representation (IR) which matches the target machine's ISA. This approach contrasts with the traditional method which operates on a source-level IR. Vectorizing a low-level IR has three primary advantages:

- The bulk of the compiler's existing optimizations and transformations do not require modification. With any vectorization scheme, compilation phases that execute after vectorization must be cognizant of vector opcodes. In a back-end approach, these phases include only the final few steps of compilation (*e.g.*, register allocation).

- The compiler can naturally identify opportunities for parallelizing subexpressions of partially vectorizable statements. Front-end approaches typically vectorize whole statements, relying on transformations such as node splitting [6] to extract vectorizable subexpressions.

- Selective vectorization can more accurately compare code selection alternatives. The algorithm in Section 4.2 operates on machine-level instructions and employs a detailed model of the target architecture. A front-end approach must estimate performance without knowledge of the operations that will populate the loop after optimization.

Compared to a front-end approach, vectorizing a low-level representation introduces four minor challenges:

- The compiler must identify loops in the control-flow graph. In contrast, high-level representations typically represent countable loops explicitly. Practically, software pipelining already requires a loop identification analysis which we can leverage for selective vectorization.

- Low-level representations typically employ a three-operand format (*i.e.*, two source operands and one destination operand). To produce a three-operand representation, the compiler introduces temporary registers to hold the intermediate values of source-level expressions. In a naïve system, these temporaries create loop-carried dependences that prevent vectorization. As discussed in Chapter 3, however, there is no impediment to privatizing scalar temporaries to remove false dependences.

- Memory dependence analysis is more effective for high-level representations. My infrastructure performs the analysis early and annotates memory operations with dependence information so that it is available in the back-end.

- Address calculations are explicit and optimized. In a high-level representation, address arithmetic is often embedded in array references.

The final point offers the only real difficulty. An effective optimizing compiler performs induction-variable recognition and strength reduction [64] in order to reduce the overhead of address arithmetic. As an example, consider the following loop:

```
for (i=0; i<N; i++) {
  a[i] = b[i];
}
```

Induction variable optimization produces

```
pa = a;
pb = b;
for (i=0; i<N; i++) {
  *pa = *pb;
  pa++;
  pb++;
}
```

For strided memory operations, this transformation reduces address calculations to one addition for each load or store. Without it, array references require several arithmetic operations. Induction variable optimization is actually advantageous when targeting multimedia extensions since it provides a straightforward mechanism for identifying unit-stride memory operations. When vectorizing load and store instructions, the only complication is ensuring the compiler updates the associated address arithmetic appropriately. For the loop above, vectorization should produce

```
pa = a;
pb = b;
for (i=0; i<N-1; i+=2) {
  pa[0:1] = pb[0:1];
  pa += 2;
  pb += 2;
}
```

4.2 Algorithm Description

This section describes the specific algorithm I have implemented to perform selective vectorization. Before discussing the details, however, it is important to re-emphasize that the compiler performs software pipelining, in the form of modulo scheduling [74], directly following selective vectorization. The use of software pipelining has strong implications for the design of the selective vectorization algorithm. When an operation does not lie on a dependence cycle, a software pipeline can hide its latency. Vectorizable operations rarely lie on dependence cycles. An exception is the case where a dependence distance equals or exceeds the vector length. For example, a loop statement `a[i+4] = a[i]` has a dependence cycle, but is vectorizable for vector lengths of 4 or less. In practice, these situations are uncommon and a dependence cycle typically prevents vectorization altogether. The consequence is that software pipelining allows selective vectorization to ignore operation latency and focus solely on resource usage.

To perform selective vectorization, I adapt a mincut partitioning heuristic due to Fiduccia and Mattheyses [27], who credit their minimization technique to Kernighan and Lin [46]. Figures 4-1 and 4-2 list the pseudocode. The high-level objective of the algorithm is to move operations between a vector and scalar *partition*, searching for a division that minimizes a cost function. The cost function for selective vectorization, described shortly, measures the configuration's overall resource usage. My implementation initially places all operations in the scalar partition (lines 1–2). A completely vectorized starting configuration is equally viable, however.

The partitioning heuristic is iterative. In Figure 4-1, lines 6–17 constitute a complete *iteration*. Within each iteration, the algorithm repartitions each vectorizable operation exactly once. At each *step*, the heuristic selects one operation to move to the opposing partition. Among the alternatives, it selects the operation that produces the configuration with the lowest overall cost (lines 19–28). Note that in many cases, the algorithm may actually cause cost increases as it transitions from one configuration to another. It is this strategy that allows the heuristic to climb out of local minima.

After each step, the algorithm locks the selected operation and removes it from consideration until the next iteration (line 12). It then computes the cost of the new configuration, noting the minimum encountered so far (lines 13–16). After repositioning all vectorizable

```
// Assign each operation to a vector or scalar partition
PARTITION-OPS ()
01  foreach op ∈ OPS
02     currPartition[op] ← SCALAR
03  bestPartition ← currPartition
04  minCost ← CALCULATE-COST (currPartition)
05  lastCost ← ⟨∞, ∞⟩
06  while lastCost ≠ minCost
07     lastCost ← minCost
08     locked ← ∅
09     foreach vectorizable op
10        bestOp ← FIND-OP-TO-SWITCH (currPartition, locked)
11        currPartition[bestOp] ← ¬ currPartition[bestOp]
12        locked ← locked ∪ bestOp
13        cost ← CALCULATE-COST (currPartition)
14        if COMPARE-COST (cost, minCost)
15           minCost ← cost
16           bestPartition ← currPartition
17     currPartition ← bestPartition
18  return bestPartition

// Select an unlocked operation to move to the opposing
// partition. Choose the alternative that produces the
// lowest overall cost.
FIND-OP-TO-SWITCH (currPartition, locked)
19  minCost ← ⟨∞, ∞⟩
20  foreach op ∈ OPS
21     if op is vectorizable ∧ op ∉ locked
22        currPartition[bestOp] ← ¬ currPartition[bestOp]
23        cost ← CALCULATE-COST (currPartition)
24        currPartition[bestOp] ← ¬ currPartition[bestOp]
25        if COMPARE-COST (cost, minCost)
26           minCost ← cost
27           bestOp ← op
28  return bestOp
```

Figure 4-1: Partitioner pseudocode, part 1.

operations, the heuristic initiates a new iteration using the lowest cost configuration as the starting point (line 17). This process repeats until an iteration fails to improve on its initial configuration. In the end, the operations remaining in the vector partition constitute the portion the compiler will vectorize.

As mentioned, the algorithm compares configurations according to their resource usage. Since the goal of selective vectorization is to produce more efficient software pipelines, a sensible cost function is a configuration's ResMII, or minimum initiation interval implied by resource usage. Rau observed that modulo scheduling is successful in achieving the theoretical minimum II in the vast majority of cases. In his experiments, modulo scheduling discovered the minimum II for 96% of loops [73, 74]. Based on this result, we can

```
// Calculate the cost of a configuration. Return a tuple
// containing the maximum weight of any resource (ResMII),
// and the sum of squares of the resource weights.
CALCULATE-COST (currPartition)
high ← 0
sum ← 0
usage ← GET-USAGE (currPartition)
foreach r ∈ RESOURCES
   cycles ← ⌈usage[r] / NUM-UNITS (r)⌉
   high ← MAX (high, cycles)
   sum ← sum + cycles²
return ⟨high, sum⟩

// Calculate the weight (in processor cycles) of each set of
// resources. Account for the fact that scalar ops will be
// unrolled by the vector length. Also account for any
// explicit communication ops implied by the configuration.
GET-USAGE (currPartition)
foreach r ∈ RESOURCES
   usage[r] ← 0
foreach op ∈ OPS
   if currPartition[op] = SCALAR
      for i ← 1 to VECTOR-LENGTH
         usage ← ADD-USAGE (SCALAR-OPCODE (op), usage)
   else
      usage ← ADD-USAGE (VECTOR-OPCODE (op), usage)
   if COMMUNICATION-NEEDED (op, currPartition)
      foreach c ∈ COMMUNICATION-OPS (op, currPartition)
         usage ← ADD-USAGE (c, usage)
return usage

// Add the resource requirements of "opcode" to the
// running total.
ADD-USAGE (opcode, usage)
foreach r ∈ RESOURCES-REQUIRED (opcode)
   usage[r] ← usage[r] + OCCUPANCY (opcode, r)
return usage

// Compare the cost of two configurations. Return true if
// the first is "better" than the second
COMPARE-COST (cost1, cost2)
⟨high1, sum1⟩ ← cost1
⟨high2, sum2⟩ ← cost2
if (high1 < high2) ∨ (high1 = high2 ∧ sum1 < sum2)
   return TRUE
return FALSE
```

Figure 4-2: Partitioner pseudocode, part 2.

assume that a reduction in ResMII will almost certainly lead to improved performance for resource-constrained loops. For recurrence-constrained loops, selective vectorization has no opportunity to improve performance.

For many architectures, computing a configuration's ResMII is straightforward. Most general-purpose processors provide sets of homogeneous functional units that operate on

Source benchmark	% exec. time	$\sum w^2$	$\sum w^3$	ResMII	1) ResMII 2) $\sum w^2$	1) ResMII 2) $\sum w^3$
101.tomcatv	51%	42	37	27	**26**	**26**
101.tomcatv	12%	11	11	10	**9**	**9**
101.tomcatv	9%	9	9	**7**	**7**	**7**
103.su2cor	13%	28	28	**16**	**16**	**16**
103.su2cor	7%	**9**	**9**	**9**	**9**	**9**
103.su2cor	5%	28	28	**18**	**18**	**18**
171.swim	33%	26	26	**19**	**19**	**19**
171.swim	32%	25	25	**15**	**15**	**15**
171.swim	20%	16	16	**14**	**14**	**14**
172.mgrid	52%	31	**23**	**23**	**23**	**23**
172.mgrid	27%	31	**23**	**23**	**23**	**23**
172.mgrid	3%	**30**	**30**	**30**	**30**	**30**

Figure 4-3: Initiation intervals achieved with selective vectorization using various cost functions.

strict subsets of the opcode space. In this case, the ResMII is simply the maximum *weight* across all resources. Given a set of resources R that executes instructions of type I, the weight of R is

$$\left\lceil \frac{\text{Number of operations of type } I \times \text{Occupancy of } I}{\text{Number of units in } R} \right\rceil \tag{4.1}$$

For now, I assume that partitioning can compute the ResMII using this method. Section 4.4 describes a scheme for extending the cost function to accommodate more complex interfaces. In Figure 4-2, lines 37–48 compute usage in processor cycles for each set of resources. The calculation includes two subtleties. First, the cost function must account for the fact that selective vectorization unrolls scalar operations by the vector length (lines 40–42). Second, the algorithm must consider the overhead of communicating data between operations in different partitions (lines 45–47).

In some cases, a group of alternative configurations may have the same ResMII. To better differentiate among them, I make one refinement to the cost function: given two configurations with equal ResMII, the partitioning heuristic prefers the alternative with a lower value for the sum of squares of the resource weights. This approach tends to favor configurations with lower *overall* resource usage. In the pseudocode, lines 29–36 compute the ResMII and sum of squares for a particular configuration. Lines 52–56 compare the cost of two configurations.

To justify the partitioner cost function, Figure 4-3 compares its performance to several similar methods. The figure lists initiation intervals produced by selective vectorization when targeting the L-machine from Chapter 1. I show performance for several loops drawn from SPEC FP benchmarks. In the evaluation of Chapter 6, these loops represent the cases for which selective vectorization is particularly important in providing whole-program performance improvements. Column 3 shows results when the partitioning algorithm compares configurations by the sum of squares of resource weights. Column 4 shows results for a sum of cubes calculation. Both methods drastically underperform a ResMII comparison, shown in column 5. The results in column 6 use the cost function in the pseudocode of Figure 4-2, and the results in column 7 show results for a similar cost function which uses a sum of cubes calculation for the secondary comparison. Both methods perform identically, and provide an improvement for two loops in `tomcatv`.

4.3 Partitioning Example

To better describe the selective vectorization algorithm, I now present a simple example which details the key steps. In this section, I target the architectural model in Figure 4-4, with compiler-visible resources consisting of 3 memory units, 2 scalar units, and a single vector unit which operates on vectors of 2 elements. All operations have single-cycle throughput. I assume the target architecture requires explicit instructions to communicate operands between scalar and vector instructions. Rather than requiring transmission through memory, however, the target processor provides a direct network between the vector and scalar register files. Transmitting a vector operand requires two explicit transfer operations which execute on the scalar arithmetic unit.

I use the data dependence graph in Figure 4-5 as the target loop body. The graph contains no cycles, which means that all operations are candidates for vectorization. The figure shows the 12 steps of the first iteration of the partitioning heuristic. For each configuration, highlighted operations designate those currently in the vector partition. Adjacent to each configuration, I show the weight of each resource. The ResMII is the maximum among the

Processor Parameter	Value
Memory units	3
Scalar arithmetic units	2
Vector arithmetic units	1
Vector length	2

Figure 4-4: Architectural model for the partitioning example in Figures 4-5 and 4-6.

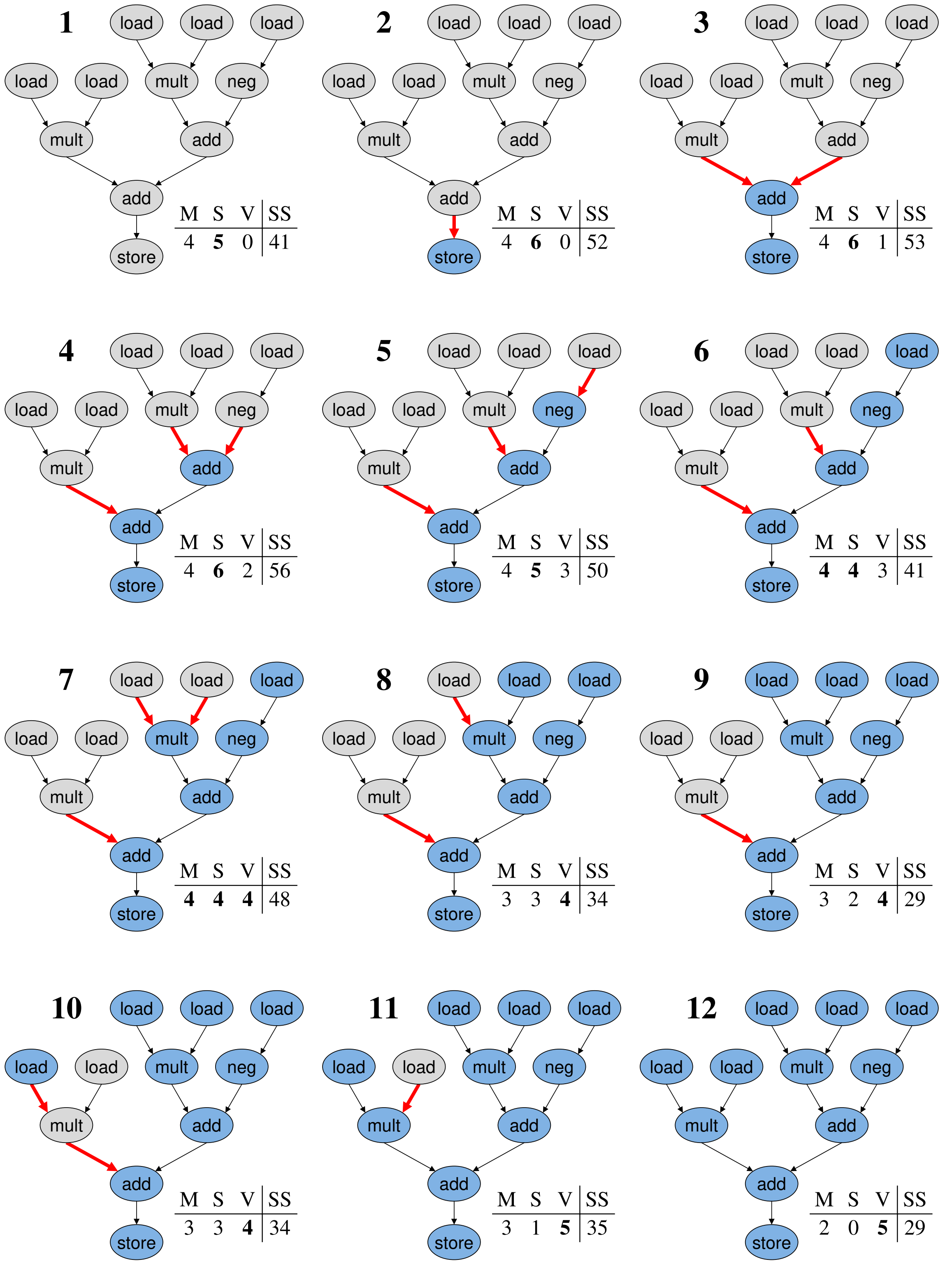


Figure 4-5: Example of two-cluster partitioning for selective vectorization. This figure shows the 12 steps in the first iteration of the algorithm. Highlighted nodes designate operations currently in the vector partition. Highlighted edges indicate dataflow between scalar and vector operations. For each configuration, **M**, **S**, and **V** show the weights for the memory, scalar, and vector units, respectively. **SS** is the sum of squares of resource weights.

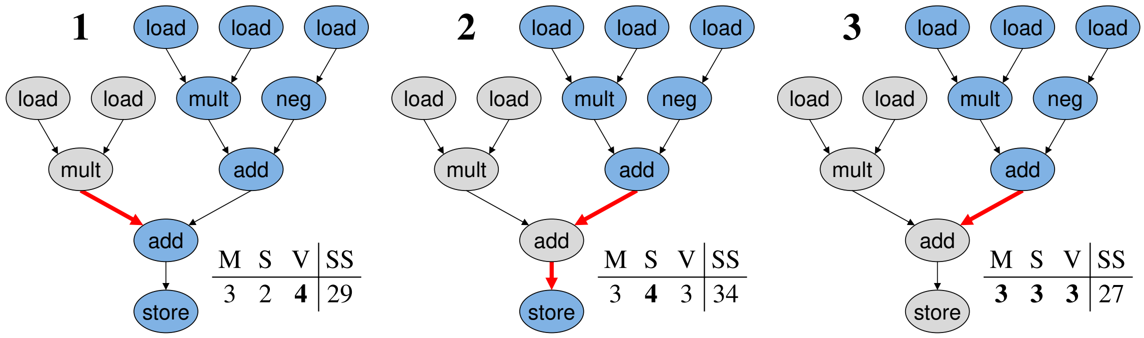


Figure 4-6: Second iteration for the example of Figure 4-5.

memory, scalar, and vector units. I also show the sum of squares of the weights. Initially, the algorithm places all operations in the scalar partition, as shown in the upper left. The associated usage table shows resource weights assuming all scalar operations are unrolled by a factor of 2. In other words, 3 memory units require 4 cycles to execute 2×6 memory operations, and 2 scalar arithmetic units require 5 cycles to execute 2×5 ALU operations. Even though the completely scalar partition in step 1 would not *require* unrolling, this assumption simplifies our comparison to the other configurations. For the first step, the sum of squares value is 41 $(4^2 + 5^2)$.

In step 2, the algorithm selects one operation to repartition. The heuristic attempts to locate the option that produces the lowest cost configuration. In this case, however, all memory operations are equivalent. For illustration, I arbitrarily select the `store` operation for repartitioning. As it turns out, this is a poor choice since the optimal configuration executes the `store` in scalar form. As we will see, the heuristic is able to amend this decision in a second iteration.

For each configuration, the associated resource weights include any explicit communication operations implied by the partition. For example, the weight of the scalar unit in step 2 includes two explicit transfer operations. These transfer operations increase the configuration's ResMII to 6 cycles. The algorithm tolerates this increase in the hope that it will later discover a better configuration. In Figure 4-5, step 5 rediscovers a ResMII of 5 cycles and step 6 locates a configuration requiring only 4 cycles.

The algorithm continues in this fashion until it repartitions every operation exactly once. At this point, it selects the lowest cost configuration as the starting point for another iteration. Since the configurations in steps 6–10 all exhibit a ResMII of 4 cycles, the heuristic

Cycle	MEM 1	MEM 2	MEM 3	ALU 1	ALU 2	VECT
n	vload (1)	vload (1)	vload (1)	mult (2)	mult (2)	vadd (2)
n+1	load (1)	load (1)	load (1)	xfer (2)	xfer (2)	vmult (1)
n+2	load (1)	store (3)	store (3)	add (2)	add (2)	vneg (1)

Figure 4-7: Modulo schedule for the optimal partition in Figure 4-6. Numbers in parentheses indicate the pipeline stage for each operation.

selects the alternative with the lower sum of squares value. In this case, the configuration in step 9 is the best choice. Figure 4-6 shows the initial steps for a second iteration of the heuristic. After two steps, the algorithm locates a configuration with a ResMII of 3 cycles. For this example, the configuration in step 3 is optimal and the algorithm uncovers no further improvements in the remaining steps (not shown). The progress in the second iteration would induce a third iteration, after which the algorithm would terminate.

Figure 4-7 shows a valid modulo schedule for the final partition in Figure 4-6. The pipeline stages, shown in parentheses, assume all operations have single-cycle latency. The schedule fully utilizes all machine resources, including the vector unit, and achieves an initiation interval of 3 cycles.

4.4 Cost Calculation for Complex Interfaces

Many architectures provide simple interfaces that lead to a closed-form solution for calculating resource weights. This section describes a method for accommodating architectures with complex interfaces in which a straightforward solution is not available. Calculating a resource's minimum usage is simple when the architecture provides sets of homogeneous resources that operate on strict subsets of the opcode space. Most general-purpose processors export this interface. Nevertheless, there are many examples of architectures for which Equation 4.1 does not apply.

Historically, VLIW designs export a low-level interface with heterogeneous functional units and operations that reserve multiple resources at specific intervals. For these architectures, the compiler must carefully orchestrate resource usage to ensure that conflicts do not occur. Pioneers of modulo scheduling developed their approach for VLIW architectures with complex interfaces [18, 73, 74, 76]. If selective vectorization is to be generally applicable to current and future designs, it is important that it be able to target any general-purpose architecture.

To perform cost analysis for complex architectural interfaces, I leverage the ResMII calculation advocated by Rau in his description of iterative modulo scheduling [73, 74]. Rau used a heuristic bin-packing algorithm to compute resource weights. Bin-packing associates a *bin* with each compiler-visible resource, and iteratively assigns operations to the bins they require for execution. The standard formulation visits operations in order of their scheduling alternatives such that those with little freedom are binned first. Given a choice of bins, the heuristic selects the option that minimizes a cost function. Iterative modulo scheduling uses the current ResMII, or maximum resource weight, to differentiate among alternatives. For selective vectorization, it makes sense to adopt the slightly modified cost function of Figure 4-2 so that a configuration's cost calculation and cost comparison use the same formula.

Figures 4-8 and 4-9 show the selective vectorization algorithm extended with a bin-packing cost function. Lines 45–49 show the top-level bin-packing routine and lines 59–69 provide the bin-selection functionality. Lines 34–39 compute the cost of a configuration using bin weights. As in Section 4.2, a configuration's cost includes the ResMII and the sum of squares of the resource weights.

The selective vectorization heuristic selects operations for repartitioning based on their effect on the total cost. Ideally, we would perform a complete bin-packing phase for each unlocked operation in order to determine the best alternative. Unfortunately, my experiments reveal many cases where this approach is too costly for a practical compilation system. To improve running time, I refine the cost analysis as follows: to compute the cost of repartitioning an operation, the algorithm checkpoints the current state of the bins, releases the resources for the operation under consideration, and reserves the set of resources required in the other partition. The algorithm then notes the cost of the configuration and discards the changes. In Figure 4-8, lines 30–33 provide this functionality. This process repeats for each unlocked operation (lines 22–29). Only after selecting and repartitioning an operation does the algorithm perform a fresh bin-packing (line 14).

A potential disadvantage of this optimization is that it could lead to a loss of accuracy during cost analysis. As it turns out, the modified cost function alleviates this concern. To see this, consider the example in Figure 4-10, which illustrates bin-packing for an architecture with 2 FPU and 2 ALU resources. Suppose that bin-packing reaches the state in part (a), and the next operation to be binned can execute on either ALU. In its original

```
// Assign each operation to a vector or scalar partition
PARTITION-OPS ()
foreach op ∈ OPS
   currPartition[op] ← SCALAR
bestPartition ← currPartition
bins ← BIN-PACK (currPartition)
minCost ← CALCULATE-COST (bins)
lastCost ← ⟨∞, ∞⟩
while lastCost ≠ minCost
   lastCost ← minCost
   locked ← ∅
   foreach vectorizable op
      bestOp ← FIND-OP-TO-SWITCH (bins, currPartition, locked)
      currPartition[bestOp] ← ¬ currPartition[bestOp]
      locked ← locked ∪ bestOp
      bins ← BIN-PACK (currPartition)
      cost ← CALCULATE-COST (bins)
      if COMPARE-COST (cost, minCost)
         minCost ← cost
         bestPartition ← currPartition
   currPartition ← bestPartition
   bins ← BIN-PACK (currPartition)
return bestPartition

// Select an unlocked operation to move to the opposing
// partition. Choose the alternative that produces the
// lowest overall cost.
FIND-OP-TO-SWITCH (bins, currPartition, locked)
minCost ← ⟨∞, ∞⟩
foreach op ∈ OPS
   if op is vectorizable ∧ op ∉ locked
      cost ← TEST-REPARTITION (op, bins, currPartition)
      if COMPARE-COST (cost, minCost)
         minCost ← cost
         bestOp ← op
return bestOp

// Compute the cost resulting from a repartitioning of "op".
// Release the resources used in the current partition and
// reserve the appropriate resources in the opposing partition.
TEST-REPARTITION (op, bins, currPartition)
bins ← RELEASE-RESOURCES (op, bins, currPartition)
currPartition[op] ← ¬ currPartition[op]
bins ← RESERVE-RESOURCES (op, bins, currPartition)
return CALCULATE-COST (bins)

// Calculate the cost of a configuration. Return a tuple
// containing the maximum weight of any resource (ResMII),
// and the sum of squares of the resource weights.
CALCULATE-COST (bins)
high ← 0
sum ← 0
foreach r ∈ RESOURCES
   high ← MAX (high, bins[r])
   sum ← sum + bins[r]^2
return ⟨high, sum⟩
```

Figure 4-8: Revised partitioner pseudocode, part 1.

```
// Compare the cost of two configurations. Return true if
// the first is "better" than the second
COMPARE-COST (cost1, cost2)
  ⟨high1, sum1⟩ ← cost1
  ⟨high2, sum2⟩ ← cost2
  if (high1 < high2) ∨ (high1 = high2 ∧ sum1 < sum2)
     return TRUE
  return FALSE

// Compute weights for all resources using a bin-packing
// heuristic. Assume operations are sorted according to their
// scheduling alternatives such that those with little freedom
// are binned first.
BIN-PACK (currPartition)
  foreach r ∈ RESOURCES
     bins[r] ← 0
  foreach op ∈ OPS
     bins ← RESERVE-RESOURCES (op, bins, currPartition)
  return bins

// Increment the bin weights of resources reserved by "op".
// Account for the fact that scalar ops will be unrolled by the
// vector length. Also account for any explicit communication
// ops implied by the configuration.
RESERVE-RESOURCES (op, bins, currPartition)
  if currPartition[op] = SCALAR
     for i ← 1 to VECTOR-LENGTH
        bins ← RESERVE-LEAST-USED (SCALAR-OPCODE (op), bins)
  else
     bins ← RESERVE-LEAST-USED (VECTOR-OPCODE (op), bins)
  if COMMUNICATION-NEEDED (op, currPartition)
     foreach c ∈ COMMUNICATION-OPS (op, currPartition)
        bins ← RESERVE-LEAST-USED (c, bins)
  return bins

// Add the resource requirements of "opcode" to the current
// bin weights. Given a choice between multiple resources,
// select the alternative that minimizes the cost function.
RESERVE-LEAST-USED (opcode, bins)
  foreach r ∈ RESOURCES-REQUIRED (opcode)
     minCost ← ⟨∞, ∞⟩
     foreach a ∈ ALTERNATIVES (r)
        temp ← bins
        temp[a] ← temp[a] + OCCUPANCY (opcode, a)
        cost ← CALCULATE-COST (temp)
        if COMPARE-COST (cost, minCost)
           minCost ← cost
           best ← a
     bins[best] ← bins[best] + OCCUPANCY (opcode, best)
  return bins
```

Figure 4-9: Revised partitioner pseudocode, part 2.

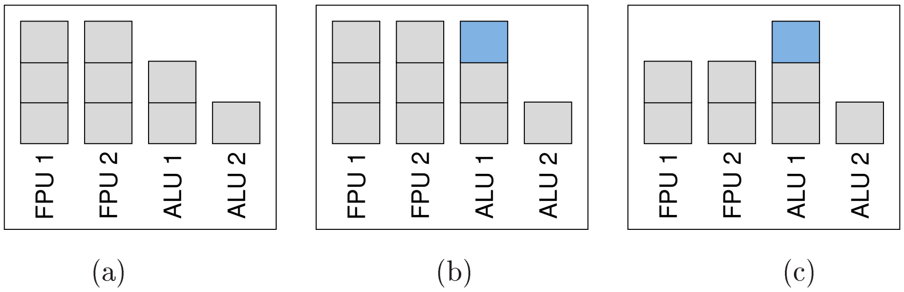


Figure 4-10: Bin-packing example.

description, the bin-packing heuristic does not prefer one alternative to the other since neither placement affects the maximum bin weight. Assume the heuristic arbitrarily chooses the first ALU, as shown in part (b). Now suppose that cost analysis removes two floating-point operations, as in part (c). Since ALU usage is unbalanced, the partitioner computes an inaccurate cost. We avoid this situation by balancing operations across resources even when they do not immediately contribute to the maximum weight. The cost function in Figure 4-8 provides this functionality.

The overall result is that the bin-packing cost analysis in Figures 4-8 and 4-9 is extremely effective in maintaining accuracy. Chapter 6 evaluates selective vectorization for a set of 8 SPEC FP benchmarks. Combined, these benchmarks contain over 300 loops that are candidates for selective vectorization (*i.e.*, they are resource-constrained and contain vectorizable operations). For every loop, the revised selective vectorization algorithm achieves the same initiation interval as the initial description in Section 4.2. The heuristic is also efficient. Compared to an approach that performs a full bin-packing at every step, the revised algorithm executes roughly 12× faster. In my infrastructure, the algorithm's running time is comparable to modulo scheduling.

In the worst case, the selective vectorization algorithm in Figures 4-8 and 4-9 is an $O(n^3)$ algorithm. Each iteration of the partitioning phase repositions every operation once and bin-packs each new configuration. Since bin-packing requires $O(n)$ steps for a loop containing n operations, each iteration has complexity $O(n^2)$. The maximum bin weight is proportional to n, and the algorithm terminates unless an iteration produces a lower-cost configuration. Therefore, the maximum number of iterations is proportional to n. In practice, however, the heuristic produces a final solution very quickly. In my benchmark suite, 85% of loops reach a solution within two iterations. Across all loops, I observe a maximum of 5 iterations.

4.5 Modulo Scalar Expansion

When the target architecture requires communication of operands through memory, intelligent code generation greatly improves loop performance. To see this, consider the following:

```
for (i=0; i<N; i++) {
  s = s + a[i] + b[i];
  p = p + c[i] + d[i];
}
```

Suppose we fully vectorize the loop for a vector length of two, using a single temporary array to transfer operands:

```
for (i=0; i<N; i+=2) {
  t[0:1] = a[i:i+1] + b[i:i+1];
  s = s + t[0];
  s = s + t[1];
  t[0:1] = c[i:i+1] + d[i:i+1];
  p = p + t[0];
  p = p + t[1];
}
```

This solution is efficient in terms of memory usage, but severely limits ILP. Since every communication sequence uses the same memory location, the compiler must completely sequentialize the corresponding load and store operations. A better approach uses a distinct location for each transfer:

```
for (i=0; i<N; i+=2) {
  t[0:1] = a[i:i+1] + b[i:i+1];
  s = s + t[0];
  s = s + t[1];
  v[0:1] = c[i:i+1] + d[i:i+1];
  p = p + v[0];
  p = p + v[1];
}
```

This method exposes more parallelism in the loop since the first three statements are fully independent of the last three. While this approach allows the compiler to reorder operations within a loop, however, it prevents overlap *among* iterations. This restriction results from the fact that a particular transfer sequence uses the same location in every dynamic iteration. Therefore, a new iteration cannot initiate until values produced in the previous iteration are consumed.

To provide complete flexibility to the software pipeliner, another approach is to perform full scalar expansion, albeit without loop distribution:

```
for (i=0; i<N; i+=2) {
  t[i:i+1] = a[i:i+1] + b[i:i+1];
  s = s + t[i];
  s = s + t[i+1];
  v[i:i+1] = c[i:i+1] + d[i:i+1];
  p = p + v[i];
  p = p + v[i+1];
}
```

Scalar expansion allows for maximum overlap among iterations since every dynamic transfer sequence utilizes a unique set of memory locations. This approach has two drawbacks, however. First, the additional memory overhead is potentially large since temporary arrays must be long enough to accommodate all iterations. Second, each expanded array introduces additional address arithmetic, which could degrade loop performance if those operations use critical resources.

In fact, we can achieve the scheduling flexibility afforded by scalar expansion without the large memory and operation overhead:

```
t = &SCRATCH_PAD;
j = 0;
for (i=0; i<N; i+=2) {
  t[0:1] = a[i:i+1] + b[i:i+1];
  s = s + t[0];
  s = s + t[1];
  t[2:3] = c[i:i+1] + d[i:i+1];
  p = p + t[2];
  p = p + t[3];
  j = j + 4;
  j = j % SCRATCH_SIZE;
  t = j + &SCRATCH_PAD;
}
```

I term this technique *modulo scalar expansion*, due to its similarity to both scalar expansion [6] and modulo variable expansion [54]. The technique uses a circular scratch pad buffer to support all operand transfers. Within an iteration, each communication sequence accesses a unique location in memory. Furthermore, the pointer increment at the end of the loop provides fresh locations for each iteration. Once the consumers of a communication sequence receive their data, it is safe to reuse those buffer locations. In order to ensure that it does not overwrite any values too early, the compiler must simply ensure that a given loop has a buffer length of

$$t * V * (s + 1)$$

where t is the number of transfers in the loop body, V is the vector length in bytes, and s is the maximum number of pipeline stages separating a communication sink from its source.

Modulo scalar expansion is very efficient in terms of operation overhead. Rounding the buffer length to the nearest power of 2 allows the compiler to replace the `mod` operation with a bitwise `and`. Also, as long as the target architecture provides base+offset addressing, buffer accesses do not require additional address arithmetic. Overall, operation overhead amounts to 3 ALU instructions[1]. With its low cost and the scheduling flexibility it provides, modulo scalar expansion is essential for absorbing the high cost of communicating through memory.

4.6 Code Generation

When selective vectorization opts to vectorize operations, the next step is to transform the loop. This section details the procedure. Figure 4-11 outlines the important high-level steps. The top-level function, CODEGEN, creates a series of basic blocks which replace the sequential loop. On line 2, the procedure first inserts a test to ensure there are sufficient iterations to enter the vector loop. In loops with fewer than *vector length* iterations, control transfers directly to a cleanup loop. Line 3 inserts an initialization block to reset the scratch pad pointer (for modulo scalar expansion) and create vectors for loop invariants.

Lines 7–18 of the pseudocode construct the vector loop. The algorithm assumes strongly connected components, or piblocks, are available from a traditional analysis, as described in Chapter 3. It further assumes that piblocks are in topological sort order. A topological sort is necessary to preserve dependences between piblocks and is analogous to the loop distribution phase in traditional vectorization. Piblocks selected for vectorization contain a single operation. Line 11 replaces the operation with the appropriate vector opcode. Lines 12–14 emit scalar operations V times, where V is the vector length. To honor depen-

[1] As with modulo variable expansion [54], we could unroll the modulo-scheduled loop and avoid any computation overhead associated with buffer management. As long as ALU operations are not a performance bottleneck, however, it is best to avoid unrolling since it puts undue stress on the instruction cache.

```
CODEGEN ()
01  R ← ∅
    // Test for iterations less than the vector length
02  R ← INSERT-VECTOR-TEST (R)
    // Initialize scratch pad pointer and loop invariants
03  R ← INSERT-VECTOR-INIT (R)
    // Construct the vectorized loop
04  R ← INSERT-VECTOR-LOOP (R)
    // Loop to execute any remaining iterations
05  R ← INSERT-POST-LOOP (R)
06  return R

INSERT-VECTOR-LOOP (R)
    // Piblocks are in topological sort order
07  foreach p ∈ PIBLOCKS
      // Insert induction ops once, with strides updated
08    if op ∈ p is induction op
09      R ← INSERT-STRIDED-OP (R, op)
10    elsif op ∈ p selected for vectorization
11      R ← INSERT-VECTOR-OP (R, op)
12    else for i ← 1 to vector length
        // ops in p are in original program order
13      foreach op ∈ p
14        R ← INSERT-SCALAR-OP (R, op, i)
15  if loop requires communication
16    R ← INSERT-SCRATCHPAD-UPDATE (R)
17  R ← INSERT-TEST-AND-BRANCH (R)
18  return R

INSERT-VECTOR-OP (R, op)
19  R ← R • VECTOR-OP (op)
20  if HAS-SCALAR-FLOW-DEPS (op)
21    R ← R • VECTOR-SCALAR-TRANSFER (op)
22  return R

INSERT-SCALAR-OP (R, op, i)
23  newOp ← COPY-OP (op)
    // Use base+offset addressing for scalar memops
24  if op is strided memop
25    newOp ← ADD-OFFSET (newOp, i)
26  R ← R • newOp
27  if i = vector length ∧ HAS-VECTOR-FLOW-DEPS (op)
28    R ← R • SCALAR-VECTOR-TRANSFER (op)
29  return R
```

Figure 4-11: Code generation pseudocode.

dences among scalar operations, the procedure emits operations within a piblock in original program order.

The code generation algorithm pays special attention to induction variable increments (lines 8–9). In contrast to other scalar operations, it inserts these instructions only once and updates their strides to reflect the unroll factor. For simplicity, the pseudocode assumes that the main induction variable appears only in the loop's branch test. This restriction is not fundamental, however, and can be removed with more meticulous code generation. For strided scalar memory operations, the compiler can insert a single address increment for all unrolled versions of the instruction as long as the target processor provides base+offset addressing (lines 24–25).

Whenever a loop requires communication between vector and scalar operations, code generation emits the appropriate transfer instructions (lines 20–21 and 27–28). Operand communication requires that the compiler rename privatizable scalars so that transfer operations can reference a distinct register for each unrolled version of a scalar operation. Although it is not explicit in Figure 4-11, the pseudocode assumes appropriate renaming for newly created instructions. Scalar renaming is generally useful since it removes false dependences and allows greater flexibility in scheduling.

At the end of the vector loop, the pseudocode inserts instructions to update the scratch pad pointer (line 20), and to perform the loop test and branch (line 23). Finally, the top-level code generation routine inserts a scalar loop to execute any iterations not performed in the vector loop (line 5).

4.7 Related Work

Automatic extraction of parallelism is an enormous field of research. This section discusses work that is closely related to selective vectorization. To my knowledge, software pipelining of vector loops was first proposed by Tang, Davidson, and Tong [87] and Eisenbeis, Jalby, and Lichnewsky [25]. These researchers advocate software pipelining as a method for achieving higher performance on the Cray-2 vector supercomputer. The Cray-2 does not support chaining, and incurs a long latency between dependent vector operations. Software pipelining hides this latency in the same way it hides latency between scalar operations. Mangione-Smith *et al.* [60, 61] extended this work to show that software pipelining lessens the need for long vector registers. Since software pipelining tends to increase register lifetimes, vector designs more readily benefit from a large number of shorter registers. Compared to selective vectorization, existing research on software pipelining for vector computers does not propose to balance computation across scalar and vector resources. Such mixed-mode operation may provide little improvement for supercomputers which dedicate the vast majority of processor resources to vector execution.

Software pipelining for clustered VLIWs [2, 16, 67, 81, 99] is closely related to selective vectorization. A clustered design consists of a number of processing clusters, each containing its own functional units and register file. Typically, these designs require explicit transfer instructions to communicate data among clusters. As in selective vectorization, balancing operations across all functional units can result in better resource utilization and lead to software pipelines with lower initiation intervals. In clustered architectures, the compiler is usually responsible for managing communication between clusters.

In some aspects, selective vectorization is simpler than partitioning for clustered machines. I have shown that selective vectorization is free to concentrate on resource usage alone since operations utilizing vector functional units rarely lie on dependence cycles. In contrast, proposals that target clustered architectures typically emphasize partition assignment for operations in dependence cycles.

Selective vectorization is more difficult than compilation for clustered architectures for two reasons. First, operand communication may introduce load and store operations that compete for resources with existing operations. By comparison, a clustered architecture usually employs an operand network with dedicated resources for communicating among clusters. A specialized communication network simplifies partitioning since communication operations compete for resources only among themselves. Second, selective vectorization must perform instruction selection and resource allocation simultaneously. Specifically, vectorizing an operation introduces a new opcode that may have entirely different resource requirements than its scalar version. Selective vectorization monitors these requirements closely in order to accurately gauge the trade-offs of scalar versus vector execution.

The partitioning phase of selective vectorization distributes operations between two clusters. The two-cluster partitioning heuristic [27, 46] described in this chapter provides an intuitive match for the problem. Nonetheless, it is possible that other partitioning approaches are suitable. Regardless of the algorithm used, this thesis shows that an effective technique must track resource usage carefully in order to gain top performance.

Chapter 5

Address Alignment

In contemporary general-purpose architectures, the alignment of vector memory operations affects code generation and performance. A memory operation is *aligned* if its address is a multiple of the vector length. Some architectures restrict memory references to aligned regions only. In this case, it is the compiler's responsibility to explicitly merge data in two aligned regions, leading to an overhead which can diminish performance in vectorized code sequences. If the processor supports misaligned references directly, these accesses still require extra latency, resource usage, or both.

This chapter presents a number of compiler techniques that reduce misalignment overhead. Section 5.1 describes the alignment issue in greater detail and outlines methods for coping with it in the compiler. Sections 5.2 and 5.3 introduce a dataflow analysis that extracts alignment information at compile-time. While the approach is effective at uncovering aligned references where they exist, misalignment is usually pervasive unless the compiler takes a more proactive approach. Section 5.4 describes a memory layout transformation that can improve the effectiveness of alignment analysis in loop nests that reference multidimensional arrays.

As an alternative to static analysis, Sections 5.5 through 5.8 describe code transformations that detect alignment at runtime. The central scheme, dynamic loop peeling, can *enforce* alignment for a portion of a loop's memory references. In order to be practical, however, the technique relies on the profiling method described in Section 5.7. For a set of SPEC FP benchmarks, the profile-based transformation enforces alignment for roughly 75% of dynamic references.

While innovative approaches can reduce the number of misaligned references, some degree of misalignment is unavoidable. In these situations, the compiler can reduce overhead through intelligent code generation. Section 5.9 demonstrates a technique similar to scalar replacement which reduces the cost of misalignment in vector loops.

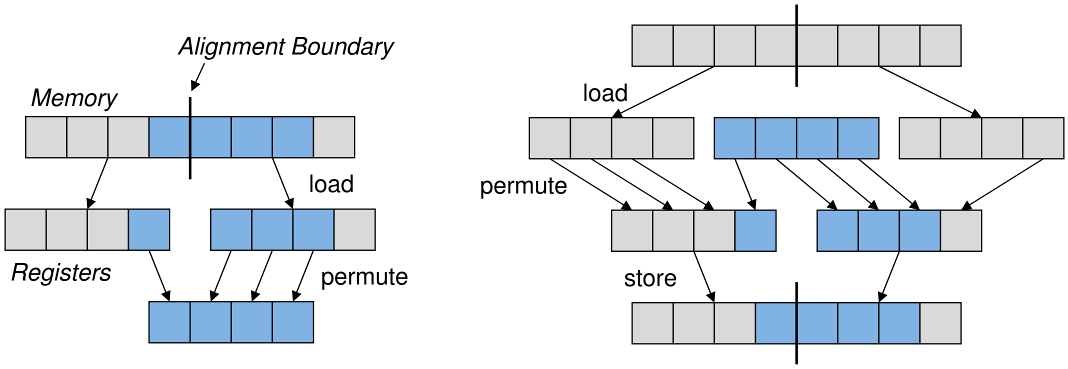
Much of the technology described in this chapter originally appeared in [56]. Other researchers [9, 10, 11, 12, 24, 51, 97] have made significant contributions both concurrently and subsequently. Rather than separating the work of others from my own, this chapter presents the breadth of alignment research in a cohesive discussion. Section 5.10 gives a historical perspective on developments in the field. Finally, Section 5.11 discusses alignment issues in the context of selective vectorization.

5.1 Background

Formally, an address A is *aligned* if $A \bmod W = 0$, where W is the width in bytes of the accessed data. Some literature terms this property *natural* or *size alignment*. When a memory reference is misaligned, the value $A \bmod W$ specifies the *offset* from alignment. The alignment restriction is not peculiar to vector memory references. Most modern RISC processors require alignment for basic data types as well. Even Intel's IA-32, which supports misaligned references directly, is more efficient when accessing aligned regions.

Alignment issues result from the fact that a misaligned reference accesses two aligned segments in memory. To support misaligned operations in hardware, the processor must implement a rotation or merge network between memory and registers, potentially increasing the latency of a vector load. Vector supercomputers tolerate this latency with longer vectors [7]. Cache-based systems have the added complexity that a misaligned access might require two tag checks and two TLB lookups, depending on whether the data cross a cache-line or page boundary, respectively.

Because of the complexity of supporting misaligned references directly, designers of the VMX/AltiVec extension [37] opted to relegate the responsibility to software. The VMX ISA ignores the low-order bits of a vector memory address, enforcing access to aligned regions. The compiler performs misaligned accesses by merging data from aligned regions. Figure 5-1 illustrates this process. The misaligned load in part (a) requires two aligned loads followed by an explicit merge operation. The store in part (b) is even more troublesome since the



(a) (b)

Figure 5-1: (a) A misaligned vector load handled explicitly in software. (b) A misaligned vector store. Highlighted regions represent data loaded or stored.

compiler must splice data with existing values in memory. Chapter 5.9 shows how careful code generation can remove much of this overhead in vector loops.

In contrast to VMX, Intel's IA-32 extensions provide direct hardware support for accessing misaligned data. Memory references are more efficient when accessing aligned regions, however, particularly if the data cross a cache-line boundary [41]. Recent extensions to IA-32 provide aligned and unaligned memory opcodes. The aligned versions are more efficient, but are valid only when the compiler can guarantee alignment. For a simple loop that copies one array to another, my experiments reveal that a version utilizing aligned opcodes executes roughly 2× faster on a Pentium 4 than a version using unaligned opcodes.

VMX and SSE represent two points in the design space of multimedia memory systems. In either model, it is advantageous to reduce the overhead of misalignment as much as possible. The compiler can achieve this reduction using the following methods:

- **Static detection of aligned references**. Compile-time alignment information is particularly important for architectures like VMX, which suffers a performance penalty whenever the compiler is unable to guarantee alignment. Static information is also useful for IA-32, where the compiler can employ more efficient opcodes when it is safe.

- **Code transformations that enforce alignment**. Such techniques are useful for both models since aligned references are more efficient than unaligned references, regardless of the underlying hardware support.

- **Intelligent code generation**. In many cases, enforcing alignment for all memory operations is impossible. Judicious code generation can reduce the overhead of the references that are misaligned.

In the following sections, most of my discussion assumes that the compiler enforces *base alignment* for arrays, meaning that the first element of an array is aligned to the vector length. Current compilers already enforce alignment at smaller boundaries since most systems require standard data types to be aligned. To support base alignment at larger boundaries, we simply need to modify existing conventions. Enforcing alignment for global arrays is trivial. For stack-allocated data, the compiler is responsible for layout within a stack frame, and can guarantee base alignment as long it allocates frames in multiples of the vector length. For heap-allocated data, enforcing base alignment is possible if we modify memory allocation routines (e.g., `malloc`) to return aligned pointers. Recent multimedia extensions support vectors totaling 16 bytes. Unless otherwise noted, the examples in this chapter assume that vector memory operations must be 16-byte aligned, and that the compiler can guarantee 16-byte base alignment for local and global arrays.

5.2 Alignment Analysis

This section describes a dataflow analysis I developed [56] to extract alignment information from programs. The goal of alignment analysis is to identify static memory references that are aligned on all dynamic executions. In a conventional setting, the compiler would likely perform alignment analysis after vectorization as part of the code generation phase. My infrastructure applies the analysis early so that selective vectorization can account for misalignment overhead during partitioning. In this case, the objective is to identify memory references having an offset of zero on entry to a loop. Consider the following example:

```
int main() {
  float a[N], b[N];
  float *pa = &a[4];
  float *pb = &b[1];
  ...
  for (i=0; i<N-4; i++) {
    *pa = *pb;
  }
}
```

```
// Compute alignment information within a procedure
// using a standard dataflow analysis.
ANALYZE-PROC (proc)
    // initialize data for each variable to ⊤
    foreach b ∈ basic blocks in proc
        worklist ← worklist ∪ b
        foreach v ∈ variables in proc
            data[b][v] ← ⊤
    // iterate until a fixed point is reached
    while worklist ≠ ∅
        b ← block in worklist
        worklist ← worklist − b
        old ← data[b]
        data[b] ← ANALYZE-BLOCK (b, proc, data)
        if data[b] ≠ old
            foreach s ∈ successors of b
                worklist ← worklist ∪ s
    return data

// Compute dataflow values within a basic block using
// the transfer functions in Figure 5-3.
ANALYZE-BLOCK (b, proc, data)
    // first merge data from predecessor blocks
    foreach v ∈ variables in proc
        curr[v] ← ⊤
        foreach p ∈ predecessors of b
            curr[v] ← curr[v] ⊓ data[p][v]
    // visit instructions in program order
    foreach op ∈ operations in b
        if op is add
            curr[op.dest] ← curr[op.src1] + curr[op.src2]
        elsif op is sub
            curr[op.dest] ← curr[op.src1] − curr[op.src2]
        elsif op is mult
            curr[op.dest] ← curr[op.src1] ∗ curr[op.src2]
        elsif op is load constant
            curr[op.dest] = ⟨V, op.src mod V⟩
        if op is array base
            curr[op.dest] = ⟨V, 0⟩
        elsif op has dest
            curr[op.dest] ← ⟨1, 0⟩
    return curr
```

Figure 5-2: Alignment analysis pseudocode.

On the first iteration of the loop, `pa` falls on a 16-byte boundary, but `pb` has an offset of 4 bytes. If we vectorize the loop for a vector length of 4 (*i.e.*, 16 bytes), the store to `pa` will be aligned, but the load from `pb` will not.

To extract alignment information from arbitrary control-flow, I developed a static dataflow analysis similar to constant propagation. The algorithm characterizes pointer variables with a *stride* and *offset*, $\langle s, o \rangle$, which indicate that the variable only sees values from the subset $sn + o$, where n is a non-negative integer. After vectorization, a memory reference is

meet	$\langle s_1, o_1 \rangle$	$\sqcap$	$\langle s_2, o_2 \rangle$	$=$	$\langle \gcd(s_1, s_2, \lvert o_1 - o_2 \rvert), o_1 \bmod s_1 \rangle$
addition	$\langle s_1, o_1 \rangle$	$+$	$\langle s_2, o_2 \rangle$	$=$	$\langle \gcd(s_1, s_2), (o_1 + o_2) \bmod s_1 \rangle$
subtraction	$\langle s_1, o_1 \rangle$	$-$	$\langle s_2, o_2 \rangle$	$=$	$\langle \gcd(s_1, s_2), (o_1 - o_2) \bmod s_1 \rangle$
multiplication	$\langle s_1, o_1 \rangle$	$*$	$\langle s_2, o_2 \rangle$	$=$	$\langle \gcd(s_1 s_2, s_1 o_2, s_2 o_1, V), o_1 o_2 \bmod s_1 \rangle$

Figure 5-3: Transfer functions for alignment analysis. V is the vector length in bytes.

guaranteed to be aligned if the analysis can prove its address has the value $\langle V, 0 \rangle$ on entry to a loop. If $s = V$, and $o \neq 0$, then o specifies the offset from alignment. Two references with values $\langle s_1, o_1 \rangle$ and $\langle s_2, o_2 \rangle$ are *relatively* aligned if $s_1 = s_2 = V$ and $o_1 = o_2$.

Alignment analysis uses a standard worklist algorithm, as shown in the pseudocode of Figure 5-2. At every point in a procedure, the analysis associates a stride and offset with any variable involved in an address calculation. Initially, all variables are assigned the value $\top$, indicating they have yet to be analyzed (lines 1–4). The algorithm then places all basic blocks in a worklist (lines 1–2), and removes them iteratively for analysis. Whenever analysis of a basic block modifies its dataflow information, the algorithm reinserts all successors into the worklist (lines 10–12). The process terminates when the worklist empties.

Within a basic block, alignment analysis propagates dataflow values using the transfer functions listed in Figure 5-3. As with any forward dataflow analysis, the first step merges the values of variables from all predecessor blocks using the *meet* operator (lines 14–17). The algorithm then visits operations within a basic block in original program order. Lines 19–24 compute dataflow values using the transfer functions for addition, subtraction, and multiplication, which are the primary operations in address arithmetic. For all transfer functions, combining an element $\langle s, o \rangle$ with $\top$ results in $\langle s, o \rangle$. Otherwise, the analysis computes new dataflow values using the vector length, V, and the current values of the operation's two sources. Any operation not listed in Figure 5-3 results in the value $\langle 1, 0 \rangle$ (lines 29–30), indicating an absence of information. For an integer constant, C, the analysis assigns the value $\langle V, C \bmod V \rangle$ (lines 25–26). Approximating a constant value with a stride and offset leads to no loss of precision and allows us to use one set of transfer functions. Finally, the algorithm assumes base alignment for local and global arrays (lines 27–28).

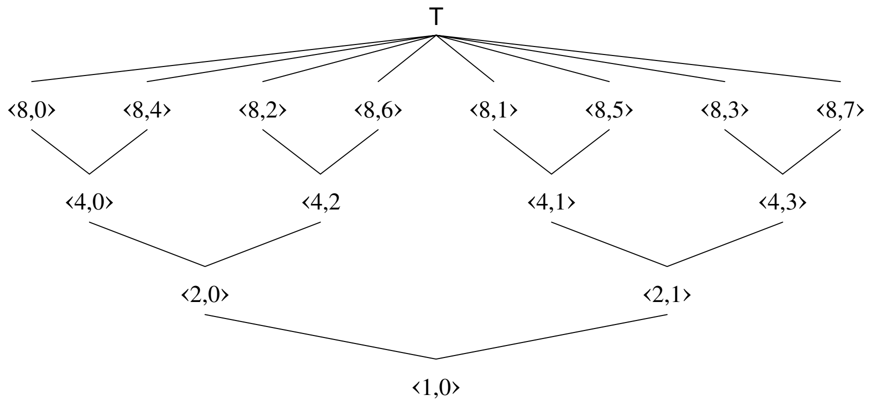


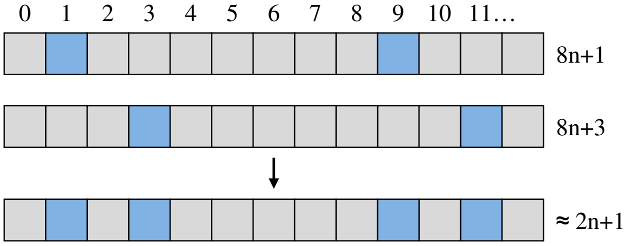
Figure 5-4: Alignment lattice for an 8-byte vector length.

Dataflow theory [1, 64] often represents the function of the meet operator with a lattice, which depicts a partial order on the set of possible dataflow values. Figure 5-4 shows the lattice for an 8-byte alignment analysis. Elements appearing higher in the lattice offer more precise information. For example, the set of values $\langle 8, 0\rangle$ is more specific than the set of values $\langle 4, 0\rangle$. The $\perp$ element familiar in dataflow analysis is equivalent to $\langle 1, 0\rangle$, and represents the case where the analysis has no information about a variable's value.

Graphically, the meet operator merges dataflow values by locating the common root in the lattice. Figure 5-5 helps to illustrate this point. Assuming an alignment boundary of 8 bytes, the pointer variable `pa` in part (a) has the value $\langle 8, 1\rangle$ in one branch of control, and the value $\langle 8, 3\rangle$ in the other. When these paths meet, the analysis must assume the variable can access either set of locations. The union of $\langle 8, 1\rangle$ and $\langle 8, 3\rangle$ is best approximated with the value $\langle 2, 1\rangle$, as illustrated in part (b). In Figure 5-4, $\langle 2, 1\rangle$ is the common root of $\langle 8, 1\rangle$ and $\langle 8, 3\rangle$.

```
char a[N];
char* pa;
if (...) {
  pa = &a[1];
}
else {
  pa = &a[3];
}
...
```

(a)



(b)

Figure 5-5: Illustration of the meet operator. (a) `pa` has a value of $\langle 8, 1\rangle$ in one branch of control and a value of $\langle 8, 3\rangle$ in the other. (b) The meet of these locations is best approximated with $\langle 2, 1\rangle$.

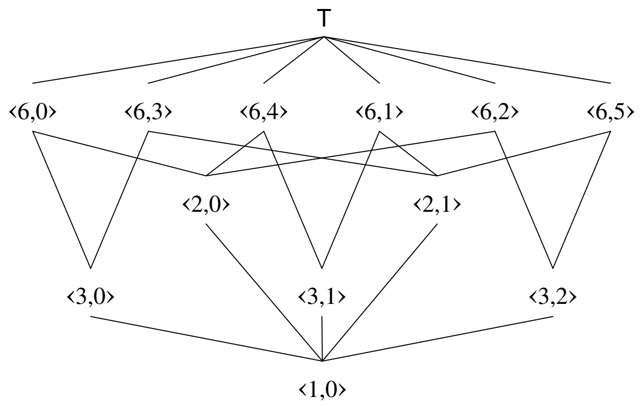


Figure 5-6: Alignment lattice for a 6-byte vector length.

The lattice's height and structure depend on the largest value relative to which we need to uncover alignment information. Recent multimedia extensions operate on vectors of 16 bytes, and require a 16-byte lattice with the same structure as the 8-byte lattice. The transfer functions in Figure 5-3 are valid for any integral value. For example, Figure 5-6 shows the lattice for a width of 6 bytes. Given the structure of modern memory systems, it is difficult to imagine a realistic design that requires alignment to boundaries that are not a power of two. In this case, Bik [10] shows how to simplify the transfer functions.

5.3 Interprocedural Alignment Analysis

The analysis described in Section 5.2 operates at the procedure level. Unfortunately, many vector loops access arrays passed as pointers to the enclosing function. This practice complicates alignment analysis since languages like C and Fortran allow the programmer to alias arbitrary elements in an array. In this case, alignment depends on the calling context. Figure 5-7 shows an example. Here, a procedure-level analysis cannot uncover alignment information in `copy` since it has no information about the base alignment of `x` or `y`. If an application's entire source code is available at compile-time, one solution for obtaining alignment information is to employ a whole-program analysis.

Extending alignment analysis to the program level is straightforward in the absence of recursive function calls. After analyzing a calling procedure f, we simply propagate dataflow values computed in f to the formal parameters of a called procedure, g. When a program contains multiple calls to a procedure, a whole-program analysis must merge the different calling contexts using the meet operator. As an example, consider the calls

```
void copy(float* x, float* y) {
  int i;
  for (i=0; i<N; i++) {
    x[i] = y[i];
  }
}

int main() {
  float a[N], b[N+4];
  copy(a, b+1);
  copy(a, b+4);
}
```

Figure 5-7: Alignment of the memory references in `copy` depends on the calling context in `main`.

from `main` in Figure 5-7. On the first call, parameters `x` and `y` have the values $\langle 16, 0\rangle$ and $\langle 16, 4\rangle$, respectively. On the second call, both parameters have the value $\langle 16, 0\rangle$. Merging the two calling contexts produces $\langle 16, 0\rangle$ for `x` and $\langle 4, 0\rangle$ for `y`. Therefore, the compiler can guarantee 16-byte alignment for the store to `x`, but rightly assumes misalignment for the load from `y`.

In order to ensure correct dataflow information, the compiler must analyze all callers to a procedure before analyzing the procedure itself. This constraint precludes analysis of recursive functions. To support recursive calls, Bik [10] proposes an extension to interprocedural constant propagation [13]. The approach uses an iterative worklist algorithm to compute alignment information for pointer parameters. The algorithm initiates by placing all functions in a worklist. At each step, it removes a function f and computes values for passed arguments using *jump functions*, which are represented in terms of the function's formal parameters. The algorithm updates values for the parameters of a called function, g, by merging their current dataflow values with those produced by the jump functions. If any values change, the algorithm reinserts g into the worklist. The process terminates when dataflow values reach a fixed point and the worklist empties. At this point, the compiler can perform a procedure-level alignment analysis using dataflow information gained for the formal parameters.

In interprocedural constant propagation, the accuracy and runtime of the algorithm depend on the complexity of the jump functions [31]. In the extreme case, a jump function could involve a full symbolic interpretation of the procedure. For alignment analysis, Bik advocates the following jump functions as sufficient to capture most cases that occur in practice:

For a call to g at call site s in function f, the jump function $J_s(y_j)$ for formal parameter y_j is given as follows:

1. If argument a_j in f, corresponding to parameter y_j in g evaluates to

   (a) an address with a statically known value $\langle s, o\rangle$, then set $J_s(y_j) = \langle s, o\rangle$.

   (b) an integer constant C, then set $J_s(y_j) = \langle V, C \bmod V\rangle$.

2. If $a_j = x_i + C$ for a constant C and x_i is a formal parameter of f with no other definitions of x_i reaching s, then set

$$J_s(y_j) = \begin{cases} \top & \text{if value of } x_i = \top \\ \langle s, (o + C) \bmod V\rangle & \text{if value of } x_i = \langle s, o\rangle \\ \perp & \text{if value of } x_i = \perp \end{cases}$$

3. Otherwise, set $J_s(y_j) = \perp$.

These jump functions uncover alignment information for pass-through parameters with constant offsets. For example,

```
void f(float* x) {
  g(x, x+1, x+2);
}
```

They fail to locate alignment when definitions within a procedure reach its call sites, as in

```
void f(float* x) {
  float* p = x;
  g(p, p+1, p+2);
}
```

I tested the effectiveness of whole-program alignment analysis for the SPEC FP benchmarks I use in the evaluation of Chapter 6. Since these benchmarks have no recursion, I perform a full analysis of a procedure before propagating dataflow information to callees. Figure 5-8 shows the percentage of dynamic memory references for which the analysis can guarantee alignment. The figure accounts for unit-stride memory operations in inner loops since they are the only candidates for vectorization. I show results for three alignment

Benchmark	V = 16	V = 32	V = 64
093.nasa7	16.70%	16.70%	16.51%
101.tomcatv	0.01%	0.00%	0.00%
103.su2cor	67.52%	52.64%	41.33%
125.turb3d	25.33%	12.99%	12.99%
146.wave5	10.84%	10.88%	10.88%
171.swim	0.02%	0.02%	0.02%
172.mgrid	0.00%	0.00%	0.00%
301.apsi	0.04%	0.04%	0.04%

Figure 5-8: Percentage of dynamically aligned memory references detected by whole-program alignment analysis.

boundaries: 16, 32, and 64 bytes. 32 and 64-byte boundaries do not appear in contemporary machines, but provide a better measure of the effectiveness of alignment analysis. The benchmarks in Figure 5-8 operate on 64-bit floating-point data, which are otherwise aligned to 8 bytes. For a 16-byte boundary, vector memory operations have two possible offsets: 0 and 8. The 32- and 64-byte boundaries allow 4 and 8 possible offsets, respectively.

Although whole-program analysis is able to detect a degree of alignment in these benchmarks, its ability is limited. A major impediment is that a particular reference must be aligned in all contexts in order for the analysis to guarantee alignment. For the benchmarks in Figure 5-8, this property is rare. To make matters worse, whole-program analysis is limited if the entire code base is not available. In later sections, I show that a local transformation combined with profiling can both detect and enforce aligned references.

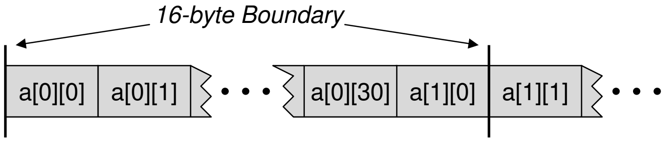
5.4 Array Padding

One source of misalignment derives from the use of multidimensional arrays. Misaligned references occur for multidimensional accesses when the size of the low-order dimension is not a multiple of the vector length. Figure 5-9 shows an example. In part (a), the offset of `a[i][j]` depends on the outer loop iteration. For instance, reference `a[0][0]` is 16-byte aligned, but reference `a[1][0]` has an offset of 12 bytes. Part (b) shows the array's layout in memory. In some cases, the compiler can fix an array's offset to its low-order dimension by padding [56]. Adding an extra element to the low-order dimension of the array in Figure 5-9 (a) produces the layout in part (c), where all accesses `a[i][0]` are aligned.

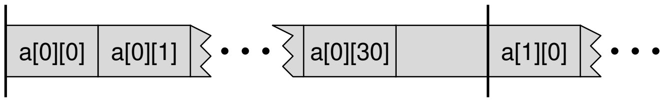
The complication in array padding is ensuring the transformation is safe. Languages like C and Fortran allow the programmer to access an array using dimensions different from

```
float a[32][31];
for (i=0; i<32; i++) {
  for (j=0; j<31; j++) {
     a[i][j] = 0.0;
  }
}
```

(a)



(b)



(c)

```
pa = &a;
for (i=0; i<32*31; i++) {
   pa[i] = 0.0;
}
```

(d)

Figure 5-9: (a) Loop nest accessing a two-dimensional array with 31 4-byte elements in the low-order dimension. (b) Data layout of the array. Without padding, reference `a[0][0]` is aligned to a 16-byte boundary, but reference `a[1][0]` is not. (c) After padding, all references `a[i][0]` are aligned to a 16-byte boundary. (d) Padding is illegal if the array is accessed with dimensions different from its declaration.

its declaration. In C, for example, it is legal to access array `a` linearly, as in Figure 5-9 (d). In this case, padding would cause the loop to access the array incorrectly. To determine when the transformation is safe, the compiler must identify all potential references to an array and ensure they are consistent with its declaration. If the array is visible in other procedures, the compiler must perform a whole-program dataflow analysis.

To test the benefits of multidimensional array padding, I implemented a whole-program analysis and padding transformation. Since padding alone does not provide any data, I extracted alignment information using the alignment analysis described in Section 5.3. Figure 5-10 shows the number of dynamic references for which the analysis can guarantee alignment after padding. Compared to the results in Figure 5-8, `tomcatv` and `su2cor` see a minor improvement with array padding.

Benchmark	V = 16	V = 32	V = 64
093.nasa7	16.70%	16.70%	16.51%
101.tomcatv	24.50%	12.25%	12.25%
103.su2cor	76.45%	66.04%	59.10%
125.turb3d	25.33%	12.99%	12.99%
146.wave5	10.84%	10.88%	10.88%
171.swim	0.02%	0.02%	0.02%
172.mgrid	0.00%	0.00%	0.00%
301.apsi	0.04%	0.04%	0.04%

Figure 5-10: Percentage of dynamically aligned memory references detected by whole-program alignment analysis after array padding.

```
void copy(double* x, double* y) {
  ...
  // Offset of x and y relative to 16 bytes
  unsigned offx = (unsigned)x & 15;
  unsigned offy = (unsigned)y & 15;
   if (offx == 0 && offy == 0) {
    // Both x and y aligned
    for (i=0; i<N; i++) { x[i] = y[i]; }
  }
  else if (offx == 0 && offy == 8) {
    // x aligned, y misaligned
    for (i=0; i<N; i++) { x[i] = y[i]; }
  }
  else if (offx == 8 && offy == 0) {
    // x misaligned, y aligned
    for (i=0; i<N; i++) { x[i] = y[i]; }
  }
  else {
    // x and y misaligned
    for (i=0; i<N; i++) { x[i] = y[i]; }
  }
}
```

Figure 5-11: Multi-versioning to detect alignment at runtime.

5.5 Multi-Versioning

As an alternative to static analysis, another approach for uncovering alignment information is to use runtime tests to identify aligned references dynamically [10, 51]. Figure 5-11 illustrates this approach. Within each test, the compiler can vectorize the loop for a specific set of conditions. In some cases, the exact offset of a misaligned reference is unimportant, and it is sufficient to determine which references are aligned. In VMX, for example, the instruction sequence for accessing misaligned data does not depend on the offset. For IA-32 processors, offset information is useful. When data cross a cache line, Bik [10] shows that accessing a vector in memory is more efficient with a series of short-width operations. He describes a *cache line split optimization*, in which he unrolls loops by a factor of C/V, where C is the cache line size. Among the unrolled versions of a misaligned memory operation, he replaces the one crossing a cache line boundary with a code sequence optimized for a specific offset.

Figure 5-12 shows the percentage of dynamic references for which multi-versioning can detect alignment in my benchmark suite. Compared to static analysis, multi-versioning detects a much larger number of aligned references. Unfortunately, the technique is only

Benchmark	V = 16	V = 32	V = 64
093.nasa7	75.51%	69.20%	65.76%
101.tomcatv	50.04%	25.02%	12.49%
103.su2cor	67.80%	52.93%	50.48%
125.turb3d	100.00%	100.00%	95.23%
146.wave5	95.94%	95.04%	95.04%
171.swim	50.01%	25.03%	12.52%
172.mgrid	61.10%	30.55%	15.28%
301.apsi	65.26%	57.01%	55.01%

Figure 5-12: Percentage of dynamically aligned memory references detected with exhaustive multi-versioning.

feasible for loops that reference a small number of pointer variables. A loop containing n variables, each having m possible offsets, requires m^n loop versions to capture all possibilities. If we ignore specific offsets and test simply for alignment, we still require 2^n loop versions to account for all cases. The results in Figure 5-12 overlook this complication and assume exhaustive testing.

Figure 5-13 illustrates the impracticality of exhaustive multi-versioning. For each benchmark, the center column shows the number of inner loops that contain unit-stride memory references. The rightmost column shows the number of additional loop versions required to support exhaustive runtime testing. Even if we disregard the execution overhead of dynamic testing, the exponential increase in code size is far too great to consider for a practical system.

Another shortcoming of multi-versioning is that it is simply a technique for detecting which references are aligned at runtime. It is not difficult to envision situations where all references are misaligned. As I will show in the following section, a loop peeling scheme can often enforce aligned references when they do not exist.

Benchmark	# original inner loops	w/ exhaustive multi-versioning
093.nasa7	23	6.77×10^7
101.tomcatv	7	4.20×10^6
103.su2cor	39	4.40×10^{12}
125.turb3d	19	1.35×10^3
146.wave5	47	1.10×10^{12}
171.swim	10	1.10×10^6
172.mgrid	17	1.07×10^9
301.apsi	35	1.72×10^4

Figure 5-13: Number of inner loops with unit-stride memory operations, and additional loop versions required for exhaustive multi-versioning.

```
void add(float* a, float* b, float* c, int n) {
  int i;
  for (i=0; i<n; i++)
    a[i] = b[i] + c[i];
}

int main() {
  float a[N], b[N], c[N];
  ...
  add(a+1, b+1, c+1,  25);
  add(a+2, b+2, c+2,  50);
  add(a+3, b+3, c+1, 100);
}
```

(a)

```
void add(float* a, float* b, float* c, int n) {
  ...
  // Pre-loop to align a[i] to 16-byte boundary
  unsigned offa = (unsigned)&a[0] & 15;
  peel = offa ? (16 - offa) / sizeof(float) : 0;
  for (i=0; i < min(n, peel); i++) {
    a[i] = b[i] + c[i];
  }

  // a[i] guaranteed to have offset 0
  for (; i<n; i++) {
    a[i] = b[i] + c[i];
  }
}
```

(b)

Figure 5-14: Dynamic loop peeling to enforce runtime alignment. (a) Original loop and its calling context. (b) A pre-loop to enforce alignment for `a[i]`.

5.6 Dynamic Loop Peeling

To enforce aligned references in vector loops, the compiler can employ dynamic loop peeling [10, 12, 56]. Figure 5-14 illustrates this approach. Given the calls from `main` in part (a), the parameters of `add` are never aligned to 16-byte boundaries. Moreover, the offsets of the arguments vary among the different calls. Part (b) introduces a pre-loop, which executes enough iterations to align `a[i]` to a 16-byte boundary[1]. With the pre-loop in place, the compiler can vectorize the subsequent loop assuming alignment for the store to `a[i]`.

In essence, the pre-loop shifts the iterations executed in parallel in a vector loop. For example, in the first call from `main` in Figure 5-14 (a), parameter `a` has an offset of 4 bytes.

[1]The code in Figure 5-14 assumes basic data types are aligned to their data width.

```
void add(float* a, float* b, float* c, int n) {
  ...
  // Pre-loop to align a[i] to 16-byte boundary
  unsigned offa = (unsigned)&a[0] & 15;
  peel = offa ? (16 - offa) / sizeof(float) : 0;
  for (i=0; i < min(n, peel); i++) {
    a[i] = b[i] + c[i];
  }
  // Check b[i] and c[i] for alignment
  if (((unsigned)&b[i] & 15) == 0 &&
      ((unsigned)&c[i] & 15) == 0) {
    // All memops guaranteed to have offset 0
    for (; i<n; i++) {
      a[i] = b[i] + c[i];
    }
  }
  // Default case: only a[i] is aligned
  else {
    // a[i] guaranteed to have offset 0
    for (; i<n; i++) {
      a[i] = b[i] + c[i];
    }
  }
}
```

Figure 5-15: Dynamic loop peeling combined with a runtime check.

At runtime, the pre-loop in part (b) would peel three iterations. For a vector length of 4, the first iteration of the subsequent vector loop would then execute iterations 4–7 in parallel, in which `a[i]` is properly aligned to a 16-byte boundary.

The pre-loop in Figure 5-14 guarantees alignment for one pointer variable. Notice, however, that enforcing alignment for `a` also enforces alignment for any references that are relatively aligned to `a`. The compiler can exploit relative alignment by adding a runtime test, as shown in Figure 5-15. In this example, the test allows the compiler to vectorize the subsequent loop assuming alignment for all memory references. In the case that `a`, `b`, and `c` are not relatively aligned at runtime, control transfers to a default loop in which only the store to `a[i]` has known alignment.

This transformation is essentially a combination of dynamic loop peeling and multi-versioning. In this example, however, the runtime test captures only one of several scenarios: the case where all references are relatively aligned. Given the third call in `main`, an optimal approach (from an alignment perspective) would also include a separate test for the relative alignment of `a` and `b` alone. For some loops, it is possible to include a test for all possibilities. Bik [10] proposes to enable or disable multi-versioning based on the degree of code expansion

Iterations	Vector iterations	a mod 16	b mod 16	c mod 16
25	6	4	4	4
50	12	8	8	8
100	25	12	12	4

Figure 5-16: Alignment profile for the code in Figure 5-14. *Vector iterations* assumes the compiler will vectorize the loop for a vector length of 4.

it entails. Combining peeling with multi-versioning is clearly superior to multi-versioning alone. Unfortunately, the number of possible loop versions is still prohibitive for loops with more than a few pointer variables. Ideally, we would test only those conditions that occur frequently at runtime. This is the topic of the next section.

5.7 Alignment Profiling

I propose runtime profiling as a means to guide pre-loop construction. Given a profile, my goal is to maximize the number of dynamic references with an initial offset of zero. With *full* vectorization, this strategy creates the greatest number of aligned references. The approach may be suboptimal for *selective* vectorization, however, since that algorithm may elect to leave some references in scalar form. Section 5.11 discusses this issue in more detail.

To obtain profiling information, I instrument an application with code to record the initial offset of all memory references in an inner loop. I limit profiling to unit-stride memory operations since they are the only candidates for vectorization. The profiling phase also records the iteration count associated with each set of offsets. Figure 5-16 shows a profile for the example in Figure 5-14. For ease of discussion, I refer to a set of offsets and the associated iteration count as a *stripe*. The profile in Figure 5-16 has 3 stripes.

After acquiring an alignment profile, the next step is to derive a set of runtime tests. In order to limit code expansion, I advocate *single-version peeling*, which combines loop peeling with a single runtime test. Figure 5-15 exemplifies the approach. That example tests alignment for all references. Assuming vectorization for a length of 4, this strategy enforces 79 aligned references: 6×3 in the first call, 12×3 in the second call, and 25×1 in the third call. The best solution omits `c` from the runtime test, leading to $6 \times 2 + 12 \times 2 + 25 \times 2 = 86$ aligned references. Ideally, the compiler would examine all possible tests in order to uncover the optimal solution. Unfortunately, an exhaustive search proves too costly for many loops. Therefore, I have developed a greedy variable-selection heuristic.

```
// Select the pointer variables to include in the
// runtime test.
SELECT-VARIABLES (loop)
    // sort stripes in descending number of iterations
    stripes ← sort(loop.stripes)
    foreach v ∈ loop.vars
        curr ← curr ∪ v
    best ← 0
    foreach s ∈ stripes
        temp ← LARGEST-SUBSET (s, curr)
        count ← NUM-DYNAMIC-REFS (loop, temp)
        if count > best
            curr ← temp
            best ← count
    return curr

// Return the largest subset of variables in the set
// "curr" that are relatively aligned in "stripe".
LARGEST-SUBSET (stripe, curr)
    most ← 0
    best ← ∅
    found ← ∅
    foreach v ∈ curr
        if v ∉ found
            // get offset of v in this stripe
            o_v ← OFFSET (v, stripe)
            temp ← {v}
            found ← found ∪ v
            foreach w ∈ curr
                if w ∉ found ∧ o_v = OFFSET (w, stripe)
                    found ← found ∪ w
                    temp ← temp ∪ w
            if |temp| > most
                most ← |temp|
                best ← temp
    return best

// Compute the number of dynamic references
// enforced in the profile data set if the variables
// in "curr" form the runtime test.
NUM-DYNAMIC-REFS (loop, curr)
    count ← 0
    foreach s ∈ loop.stripes
        if STRIPE-MATCHES (s, curr)
            count ← count + (s.iters × |curr|)
    return count

// Return true if the variables in the set "curr"
// are relatively aligned in "stripe".
STRIPE-MATCHES (stripe, curr)
    o_v ← OFFSET (v, stripe) where v ∈ curr
    foreach w ∈ curr
        if o_v ≠ OFFSET (w, stripe)
            return false
    return true
```

Figure 5-17: Greedy variable selection algorithm.

Benchmark	V = 16		V = 32		V = 64	
	zero offset	% max	zero offset	% max	zero offset	% max
093.nasa7	95.29%	n/a	95.28%	n/a	93.80%	n/a
101.tomcatv	71.44%	100.0%	63.27%	100.0%	63.27%	100.0%
103.su2cor	67.80%	100.0%	52.93%	100.0%	48.10%	100.0%
125.turb3d	100.0%	100.0%	100.0%	100.0%	90.46%	100.0%
146.wave5	97.69%	100.0%	97.61%	100.0%	97.61%	100.0%
171.swim	71.42%	100.0%	63.25%	100.0%	63.25%	100.0%
172.mgrid	62.51%	100.0%	31.94%	100.0%	18.11%	100.0%
301.apsi	28.15%	40.86%	26.75%	39.82%	25.68%	38.85%

Figure 5-18: Percentage of dynamically aligned memory references detected with single-version peeling.

Figure 5-17 shows the pseudocode for the algorithm. The heuristic first sorts the stripes in descending order of their iteration counts (line 1). Starting with the first stripe, it then computes the largest subset of variables that are relatively aligned (lines 12–27). This subset has an associated cost, which I define to be the number of dynamically aligned memory references the subset would enforce if it were to form the runtime test. The algorithm computes this cost by summing the iteration counts of all stripes in which the variables in the subset are relatively aligned, and multiplying by the size of the subset (lines 28–37).

From the initial subset, the heuristic iteratively removes variables as it deems profitable (lines 5–10). It considers each stripe in sort order. From the working subset, the algorithm extracts the largest subset of variables that are relatively aligned in the stripe. It then computes the cost of the resulting subset, replacing the current group with the smaller subset whenever the latter produces a better overall cost (lines 8–10).

I examined the effectiveness of this heuristic for the SPEC FP benchmarks studied earlier in this chapter. For all benchmarks, I compared the runtime tests generated by the heuristic to those created by an exhaustive search[2]. In all cases, the greedy heuristic obtained the same solution.

I next evaluate the overall effectiveness of the profile-based transformation. In general, profiling is only useful when optimizations guided by the profile perform well on other data sets. To test the viability of the technique, I generated runtime tests using the SPEC training data and gathered results using the reference data. Figure 5-18 shows the percentage of dynamic references for which single-version peeling can guarantee alignment. The table also reports these numbers as a percentage of the maximum achievable with a perfect profile. I do

[2]Profiles for these benchmarks were sufficiently small to allow a comparison of every possible test. The runtime of the exhaustive search, however, was not practical for a real system.

Benchmark	Single-Version	Multi-Version	Increase
V = 16			
093.nasa7	95.29%	97.08%	1.88%
101.tomcatv	71.44%	71.44%	0.00%
103.su2cor	67.80%	67.80%	0.00%
125.turb3d	100.0%	100.0%	0.00%
146.wave5	97.69%	97.69%	0.00%
171.swim	71.42%	71.42%	0.00%
172.mgrid	62.51%	62.51%	0.00%
301.apsi	28.15%	79.15%	**181.2%**
V = 32			
093.nasa7	95.28%	95.63%	0.37%
101.tomcatv	63.27%	63.27%	0.00%
103.su2cor	52.93%	52.93%	0.00%
125.turb3d	100.0%	100.0%	0.00%
146.wave5	97.61%	97.61%	0.00%
171.swim	63.25%	63.25%	0.00%
172.mgrid	31.94%	31.94%	0.00%
301.apsi	26.75%	76.36%	**185.5%**
V = 64			
093.nasa7	93.80%	94.57%	0.82%
101.tomcatv	63.27%	63.27%	0.00%
103.su2cor	48.10%	51.15%	6.34%
125.turb3d	90.46%	97.62%	7.92%
146.wave5	97.61%	97.61%	0.00%
171.swim	63.25%	63.25%	0.00%
172.mgrid	18.11%	18.11%	0.00%
301.apsi	25.68%	75.75%	**195.0%**

Figure 5-19: Percentage of dynamically aligned memory references detected with single-version peeling and peeling combined with exhaustive multi-versioning.

not show results for `nasa7` since that benchmark does not have a training data set. In almost every other case, the profiling scheme obtains the best solution. The poor performance in `apsi` is due to variation between the benchmark's two data sets. Overall however, the results suggest that profiling offers a practical solution for constructing a runtime test. Across all benchmarks, single-version peeling enforces alignment for an average of 74%, 66%, and 62% of dynamic references for 16-, 32-, and 64-bit boundaries, respectively.

I next explore the potential of combining peeling with full multi-versioning. Whereas multi-versioning alone detects only those references that are aligned at runtime, supplementing the technique with dynamic peeling can enforce alignment for a subset of relatively aligned references. The best approach detects the largest subset that are relatively aligned. While multi-version peeling is still impractical in terms of code expansion, it provides a means of gauging the effectiveness of the single-version method. Figure 5-19 compares the

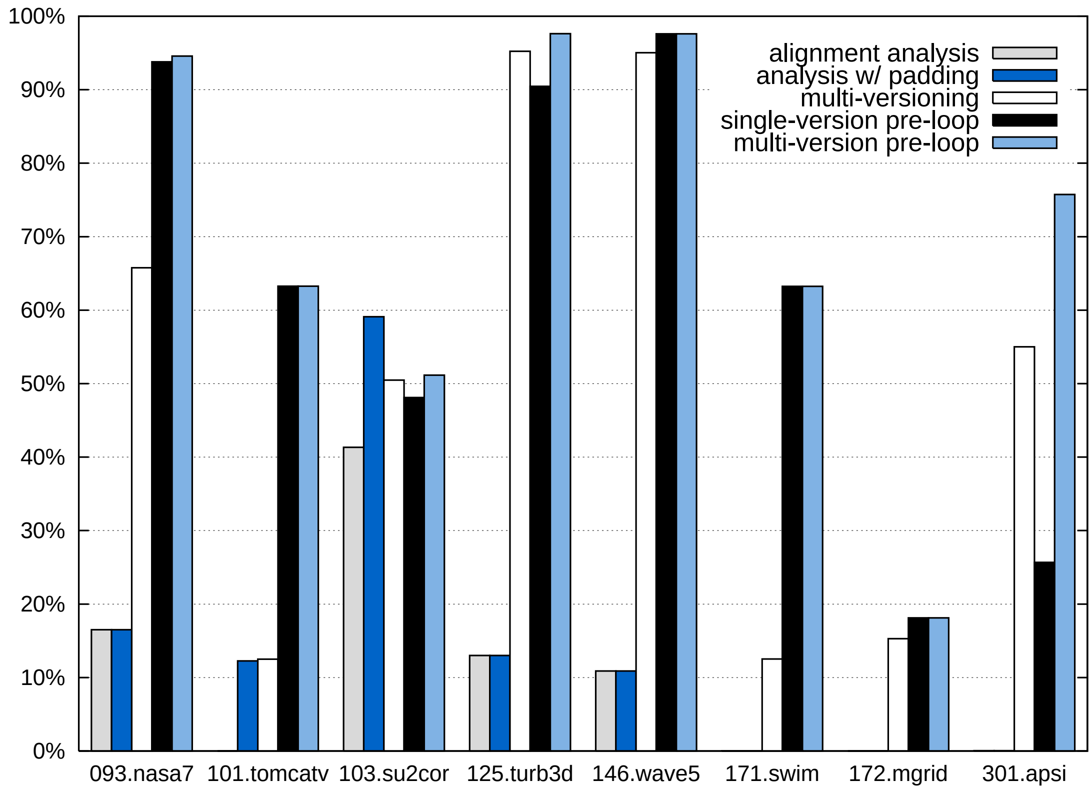


Figure 5-20: Percentage of dynamically aligned memory references detected by alignment analysis with and without padding, single-version peeling, and multi-version peeling for a 64-byte boundary.

two approaches. In columns 2 and 3, I show the number of dynamic memory references for which each technique enforces alignment. Column 4 shows the improvement of exhaustive testing over a single profile-generated test. In most cases, the methods produce identical results. The only benchmark benefiting substantially from multi-version peeling is `apsi`.

Figure 5-20 compares single-version and multi-version peeling to the techniques discussed earlier in this chapter for a 64-byte boundary. For most benchmarks, single-version peeling is among the top performers. Furthermore, the technique is the most practical of those listed in the figure since it does not require whole-program analysis and limits code expansion to an acceptable level.

5.8 Per-statement Peeling

In loops that contain multiple statements, a further reduction in misaligned references is possible using *per-statement* peeling. Figure 5-21 shows an example. Suppose we wish to vectorize the loop in part (a). Here, a pre-loop can enforce alignment for only half of the

```
void func() {
  float a[N], b[N], c[N], d[N];
  ...
  for (i=0; i<N; i++) {
    a[i+1] = b[i+1];
    c[i+2] = d[i+2];
  }
  ...
}
```

(a)

```
void func() {
  float a[N], b[N], c[N], d[N];
  ...
  for (i=0; i<N; i++) {
    a[i+1] = b[i+1];
  }
  for (i=0; i<N; i++) {
    c[i+2] = d[i+2];
  }
  ...
}
```

(b)

```
void func() {
  float a[N], b[N], c[N], d[N];
  ...
  // Separate pre-loop for each statement
  for (j=0; j < min(3, N); j++) {
    a[j+1] = b[j+1];
  }
  for (k=0; k < min(2, N); k++) {
    c[k+2] = d[k+2];
  }
  // All memops guaranteed aligned
  for (i=max(2,3); i<N; i++, j++, k++) {
    a[j+1] = b[j+1];
    c[k+2] = d[k+2];
  }
  // Separate post-loops
  for (; j<N; j++) {
    a[j+1] = b[j+1];
  }
  for (; k<N; k++) {
    c[k+2] = d[k+2];
  }
  ...
}
```

(c)

Figure 5-21: (a) A loop with relative alignment within each statement. (b) Loop distribution facilitates per-statement peeling. (c) Per-statement peeling without loop distribution.

references. Since the two statements are independent, however, they can be distributed and peeled separately, as shown in part (b). Bik [10] takes this approach in the Intel compiler. In this example, distribution and peeling ensure alignment for all references. The disadvantage is that loop distribution eliminates ILP between the statements.

In fact, Eichenberger, Wu, and their colleagues [24, 97] show how to peel each statement separately while keeping the main loop body intact. Figure 5-21 (c) illustrates this approach for the example loop. Here, we peel each statement individually using separate pre-loops, but execute the statements together in the main loop. The technique also requires separate post-loops for cases where the pre-loops peel a different number of iterations.

A potential optimization would combine per-statement peeling with runtime testing. In the example of Figure 5-21 (c), the compiler assumes the memory operations within each statement are relatively aligned. If these instructions were to reference pointer arguments, this assumption may not be possible. As with single-version peeling, a dynamic test can verify the necessary conditions at runtime. Profiling is certainly beneficial for this scheme as well, but I leave an examination of the approach to future work.

5.9 Code Generation

The analyses and transformations discussed in this chapter offer several methods for increasing the number of aligned references. When a statement contains memory references that are relatively misaligned, some overhead is unavoidable. If the processor does not support misaligned references directly, the compiler is responsible for generating the proper realignment code. In this case, an approach similar to scalar replacement [6] can remove much of the misalignment overhead in vectorized loops [24, 97]. The technique reuses data produced in the previous iteration to eliminate the extra memory operations otherwise required. This approach reduces the overhead of a misaligned memory reference to a merge and copy. Figure 5-22 shows an example for a vector add, in which the compiler assumes all memory operations are misaligned. The *merge_and_shift_left* and *merge_and_shift_right* pseudo operations provide data merging functionality. The *merge_and_shift_left* operation computes the high-order portion of shifting left the concatenation of its first two arguments, and the *merge_and_shift_right* operation delivers the low-order portion of shifting right. For both operations, the third argument determines the shift amount.

```
void add(float* a, float* b, float* c, int n) {
  ...
  i = 0;
  if (0 + 3 < n) {
    // Enforce access to 16-byte regions
    pa = (float*)((unsigned)&a[0] & ~15);
    pb = (float*)((unsigned)&b[0] & ~15);
    pc = (float*)((unsigned)&c[0] & ~15);

    // First store may need splice data with existing values
    a0 = vec_ld(pa);
    a0 = merge_and_shift_left(a0, a0, a);

    b0 = vec_ld(pb);
    c0 = vec_ld(pc);
    pb += 4;
    pc += 4;

    // Vector loop (vector length = 4)
    for (; i < n-3; i+=4) {
      // Load and merge with data from last iteration
      b1 = vec_load(pb);
      b2 = merge_and_shift_left(b0, b1, b);
      b0 = b1;
      c1 = vec_load(pc;
      c2 = merge_and_shift_left(c0, c1, c);
      c0 = c1;
      a1 = vec_add(b2, c2);

      // Merge before storing
      a2 = merge_and_shift_right(a0, a1, a);
      vec_store(a2, pa);
      a0 = a1;

      pa += 4;
      pb += 4;
      pc += 4;
    }
    // If necessary, splice remaining values with existing data
    if (((unsigned)&a[0] & 15) != 0) {
      a1 = vec_load(pa);
      a1 = merge_and_shift_left(a1, a1, a);
      a2 = merge_and_shift_right(a0, a1, a);
      vec_store(a2, pa);
    }
  }
  // Execute remaining iterations
  for (; i<n; i++) {
    a[i] = b[i] + c[i];
  }
}
```

Figure 5-22: Scalar replacement to reduce misalignment overhead.

```
void func() {
  float a[N], b[N], c[N], d[N];
  ...
  for (i=0; i<N; i++) {
    a[i] = b[i] + c[i+1] * d[i+1]
  }
  ...
}
```

(a)

```
void func() {
  float a[N], b[N], c[N], d[N];
  ...
  i = 0;
  if (0 + 3 < N) {
    // Perform first multiply
    c0 = vec_ld(c[0]);
    c1 = vec_ld(d[0]);
    m0 = vec_mult(c0, d0);

    // Vectorized loop (vector length = 4)
    for (; i < N-3; i+=4) {
      // Load, multiply and merge with data
      // computed on last iteration
      c0 = vec_load(c[i+4]);
      d0 = vec_load(d[i+4]);
      m1 = vec_mult(c0, d0);
      m2 = merge_and_shift_left(m0, m1, 4);
      m0 = m1;

      b0 = vec_load(b[i]);
      a0 = vec_add(b0, m2);
      vec_store(a[i], a0);
    }
  }
  // Execute remaining iterations
  for (; i<N; i++) {
    a[i] = b[i] + c[i+1] * d[i+1];
  }
  ...
}
```

(b)

Figure 5-23: Reducing misalignment overhead with delayed realignment.

A further reduction in misalignment overhead is possible for subexpressions that have relatively aligned memory references. Consider the loop in Figure 5-23 (a). Here, `a[i]` and `b[i]` are aligned, but `c[i+1]` and `d[i+1]` require explicit realignment. Since `c[i+1]` and `d[i+1]` are relatively aligned, however, merging can take place after the multiplication, as shown in part (b). For this example, the scheme reduces misalignment overhead to one merge and copy operation per loop iteration. Eichenberger, Wu, and their colleagues [24, 97] were the first to describe this optimization. They develop a framework called a *data reorganization graph* that allows the compiler to optimize the amount of explicit realignment. Eichenberger and Wu examine several heuristics for doing so; I refer the reader to their publications for details.

In order to exploit relative alignment, the compiler must be able to detect it. Eichenberger and Wu do not discuss methods for identifying relative alignment among memory references. Since the task is nearly identical to extracting natural alignment, the techniques described in this chapter offer a range of possible solutions.

5.10 Historical Perspective

To my knowledge, the first description of compiler techniques that address the alignment of vector memory references is due to Krall and Lelait [51]. These researchers introduced the multi-versioning transformation described in Section 5.5 as part of a framework for targeting the VIS [90] extension.

Bik [9, 10, 11, 12] was the first to advocate loop peeling as a method to enforce aligned references at runtime. He proposed static peeling for situations in which the compiler can determine offsets at compile-time, and also introduced dynamic peeling for cases in which the compiler has no information. Bik's description of loop peeling is very similar to the discussion in Section 5.6. By itself, dynamic loop peeling guarantees alignment for one pointer variable. Bik combined multi-versioning with loop peeling as a method to extract additional alignment information at runtime. In his description, the compiler uses a simple heuristic to determine when multi-versioning leads to unacceptable code expansion.

I was the first to advocate profiling as a method to capture alignment information while limiting code expansion. In [56], I combined profiling with loop peeling and runtime testing. The goal of this research was to detect the exact offset of pointer variables. In Section 5.7, I specialize the technique for enforcement of aligned vector memory references.

I was also the first to develop a static analysis to compute alignment information. I first utilized an alignment analysis in conjunction with superword level parallelization [55], and formalized the approach in [56]. Bik later described a similar analysis [10], and extended the technique to operate at the program level. He also optimized the transfer functions for power-of-two vector lengths. Pryanishnikov, Krall, and Horspool [70] developed a similar interprocedural alignment analysis based on pointer analysis.

Eichenberger, Wu, and their colleagues [24, 97] pioneered the per-statement peeling approach discussed in Section 5.8. They also described the code generation technique outlined in Section 5.9, which is crucial for reducing the overhead of explicit software realignment.

5.11 Alignment and Selective Vectorization

Wherever alignment information is available, the selective vectorization algorithm described in Chapter 4 incorporates it naturally. Simply, the cost analysis accounts for any additional resources required by misaligned vector memory operations. The approach described in this thesis performs alignment transformations before selective vectorization. If selective vectorization elects to skip vectorization for a particular memory operation, there is no advantage to including its address in the runtime test. One solution to this problem is to refine the test after selective vectorization. A more extensive solution would perform the loop transformation directly in the back-end. I leave this study to future work.

Chapter 6

Evaluation

This chapter evaluates the selective vectorization algorithm presented in Chapter 4. Section 6.1 describes my compilation and simulation infrastructure. Section 6.2 compares the performance of selective vectorization to traditional vectorization and software pipelining. On a realistic architectural model, selective vectorization achieves whole-program speedups of up to 1.35× over the conventional approaches. Section 6.3 examines performance for individual loops. I demonstrate that selective vectorization subsumes existing approaches by naturally identifying those situations where a traditional strategy provides the best performance. More importantly, selective vectorization uncovers ample opportunities for distributing computation across scalar and vector resources. In some cases, the technique achieves speedups of 1.75×, which approaches the theoretical maximum of 2×.

The bulk of my evaluation targets the L-machine, the hypothetical machine model introduced in Chapter 1. In Section 6.4, I study the performance of selective vectorization for an architecture that supports direct transmission of operands between vector and scalar register files. A specialized operand network provides further opportunities for performance improvements. Compared to the baseline L-machine, however, the speedups presented in Section 6.4 are modest. This phenomenon is due to the capacity of selective vectorization to provide high performance without specialized hardware support. Software pipelining is also instrumental in hiding the long latency of communication sequences.

Section 6.5 examines the performance impact of alignment information. Using the single-version peeling scheme presented in Chapter 5, selective vectorization provides an additional speedup of up to 1.10×. Finally, Section 6.6 examines the performance effect of increased

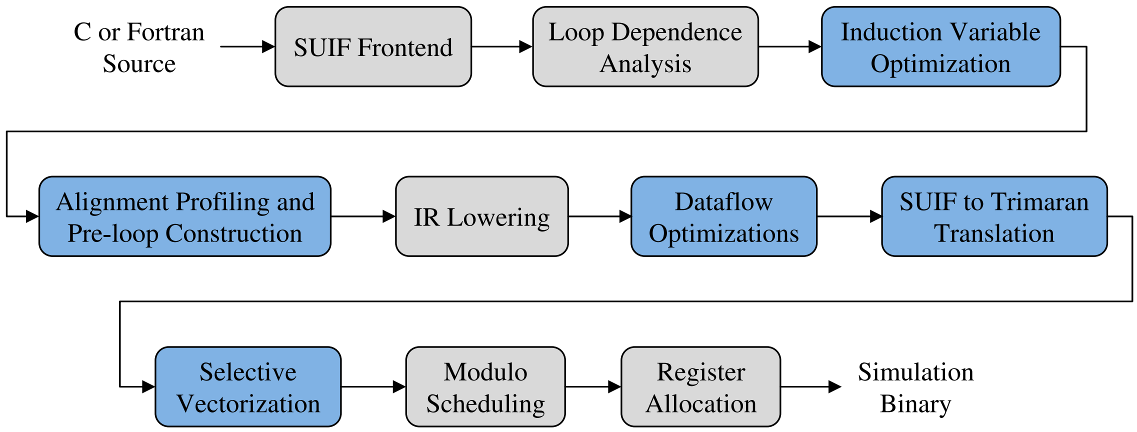


Figure 6-1: Compiler toolchain. Highlighted modules represent the compilation phases I implemented for this thesis.

vector processing power. I show that additional vector operation throughput diminishes the advantage of utilizing both scalar and vector resources. This result is not unexpected, as selective vectorization realizes the largest performance gains for architectures that provide similar scalar and vector operation throughput. Nonetheless, selective vectorization remains an effective method for targeting short-vector instructions.

6.1 Methodology

Figure 6-1 shows the major components of my toolchain. I used SUIF [95] as the compiler front-end. SUIF includes a dependence-analysis package which is crucial for identifying data parallelism. Additionally, it provides a suite of existing dataflow optimizations. For the compiler back-end and simulation system, I used Trimaran, a compilation and simulation infrastructure for VLIW architectures [91]. Trimaran is one of a few publicly available infrastructures that provide a high-quality modulo scheduler. Additionally, its machine description facility allowed me to evaluate various architectural design points. With Trimaran, my evaluation focuses on statically scheduled machines. Since superscalar processors also benefit from automatic vectorization and instruction scheduling, however, I expect my techniques will be applicable to these designs as well.

For all benchmarks, I applied a suite of standard optimizations before vectorization, including induction-variable recognition, register promotion, common subexpression elimination, copy propagation, constant propagation, dead code elimination, and loop-invariant code motion [64]. Among these transformations, induction variable optimization is particu-

Processor Parameter	Value
Issue width	6
Branch units	1
Integer units	4
Floating-point units	2
Memory units (scalar & vector)	2
Vector floating-point units	1
Vector merge units	1
Vector length (64-bit elements)	2
Scalar integer registers	128
Scalar floating-point registers	128
Vector registers	64
Predicate registers	64

Op Type	Latency	Op Type	Latency
Int ALU	1	FP Add/Sub	4
Int Multiply	3	FP Multiply	4
Int Divide	36	FP Divide	32
Load	3	Branch	1

Figure 6-2: Configuration of the L-machine.

larly important. This optimization replaces address arithmetic with an induction variable, greatly reducing address calculations in array-based code. In my infrastructure, it also provides the mechanism for identifying unit-stride memory operations. Recall from Chapter 2 that contemporary multimedia extensions only support vector references that access contiguous data.

The compilation toolchain applies selective vectorization and modulo scheduling to countable loops (*i.e.*, *do* loops) without control flow or function calls. In general, these are the code sequences to which both software pipelining and vectorization are most applicable. I identify vectorizable operations using the classic approach described by Allen and Kennedy [6] and Wolfe [96]. As discussed in Chapter 4, however, my implementation operates on a low-level IR. The primary difficulty of this approach is obtaining accurate memory dependence information. My infrastructure analyzes dependences at the source-level and preserves the information for the back-end using annotations.

Figure 6-2 shows the details of the simulated architecture, the L-machine. The processor is representative of modern designs and contains a resource set similar to Intel's Itanium 2 [62] and IBM's PowerPC 970 [36]. The L-machine provides vector opcodes for all standard computational instructions. In addition, vector instructions have the same latency as their scalar counterparts. The architecture provides one vector unit for vector floating-point com-

Benchmark	Source	Description
093.nasa7	CFP92	7 kernels used in NASA applications
101.tomcatv	CFP95	Vectorized mesh generation
103.su2cor	CFP95	Monte-Carlo method
125.turb3d	CFP95	Turbulence modeling
146.wave5	CFP95	Maxwell's equations
171.swim	CFP2000	Shallow water modeling
172.mgrid	CFP2000	Multi-grid solver: 3D potential field
301.apsi	CFP2000	Meteorology: pollutant distribution

Figure 6-3: Evaluated benchmarks.

putation. Since this chapter evaluates performance on a set of floating-point benchmarks, a vector integer unit is unnecessary.

As with VMX, the L-machine does not provide specialized support for transferring operands between scalar and vector register files. The compiler must communicate data through memory using a series of load and store instructions. The L-machine also restricts vector memory operations to aligned regions. To access misaligned data, the compiler must merge data from adjacent, aligned segments. The merge unit in Figure 6-2 is analogous to the vector permute unit in VMX and provides the necessary merging functionality.

Trimaran's modulo scheduler assumes a target architecture with rotating registers and predicated execution. Therefore, I extended the infrastructure with rotating vector registers and predicated vector operations. For designs that do not provide rotating registers, the compiler can employ *modulo variable expansion* [54] to generate valid schedules. Predication enables the compiler to use the same static instructions for the kernel, prolog, and epilog [18]. It also provides a mechanism for modulo scheduling loops with control flow, but these loops are not targeted in my infrastructure.

Figure 6-3 summarizes the benchmarks evaluated in this chapter. They consist of eight scientific applications gathered from the SPEC 92, 95, and 2000 floating-point suites. These benchmarks represent the subset of SPEC benchmarks for which the compiler was able to detect some amount of data parallelism. In order to keep simulation times practical, I evaluate performance using the SPEC training data sets. Some of the benchmarks in Figure 6-3 have large memory footprints which can negatively impact performance in cache-based systems. Without techniques such as prefetching or loop blocking, load latency can become a performance bottleneck. Since my compilation and simulation infrastructures do not support these techniques, however, I disregard cache performance in my simulations.

In keeping with contemporary processors, the L-machine's vector operands comprise a total of 128 bits. The benchmarks in Figure 6-3 operate on 64-bit data, leading to a vector length of 2. For this vector length, the L-machine provides equivalent floating-point operation throughput on vector and scalar resources. Barring secondary effects such as register pressure, selective vectorization can achieve a maximum speedup of $2\times$ for any loop in this study.

6.2 Selective Vectorization vs. Traditional Approaches

In this section, I compare selective vectorization to conventional compilation strategies. First, I implemented a *traditional* vectorizer in SUIF based on the scheme described by Allen and Kennedy [6]. The transformation operates on a high-level intermediate representation. To ensure highly optimized code, I have extended all back-end optimizations to support vector opcodes. When loops contain a mix of vectorizable and non-vectorizable operations, the traditional vectorizer employs loop distribution and scalar expansion. As described in Chapter 3, a straightforward approach tends to create a series of distributed loops. Therefore, I perform loop fusion [17, 45] after vectorization to reduce overhead as much as possible. Aside from scalar expansion, I did not employ any optimizations specifically designed to expose data parallelism. Transformations such as loop interchange [4] and reduction recognition would create further opportunities for vectorization. The focus of this thesis is not the identification of data parallelism, but how to exploit it to create highly efficient schedules. It is worth noting that selective vectorization would also benefit from transformations that expose data parallelism since they would allow greater flexibility in code selection.

To study the effect of loop distribution, I also implemented a second vectorizer in the back-end. In the results that follow, I refer to this technique as *full* vectorization. As in the traditional vectorizer, the approach vectorizes all data parallel operations. In order to exploit ILP among vector and scalar operations, however, full vectorization does not perform loop distribution. Instead, the approach unrolls scalar operations inside the vector loop. While the traditional vectorizer uses scalar expansion to communicate operands, the full vectorizer uses modulo scalar expansion, described in Section 4.5. For both vectorizers, I perform modulo scheduling after vectorization.

Given the heavy cost of communication between vector and scalar register files, I make one improvement to the baseline vectorizers: neither technique vectorizes an operation unless it has at least one vectorizable predecessor or successor. Doing otherwise is almost certainly unprofitable as the system must transmit all source and destination operands through memory just to execute a single vector instruction. Selective vectorization avoids this scenario automatically.

In this section, I do not utilize any alignment transformations or analyses. For all vectorization schemes, the compiler assumes every vector memory operation is misaligned. I remove much of the resulting overhead using the code generation technique described in Section 5.9. With this optimization, the overhead of a misaligned memory operation consists of the cost of explicit merging. Section 6.5 explores the performance advantage of compiler-enforced alignment.

I present all speedups relative to iterative modulo scheduling [74]. For this baseline, I unroll countable loops V times (for vector length V) in order to match the reduced address arithmetic of vector loops. Vector memory operations benefit from fewer address calculations since a single address references multiple locations. The same reduction in instructions is available in scalar loops for processors that provide base+offset addressing by unrolling and folding the pointer increments into memory operations.

Figure 6-4 shows the speedup of each vectorization technique over modulo scheduling. In the graph, the leftmost bar for each benchmark shows the performance of traditional vectorization. For `swim` and `tomcatv`, the majority of targeted loops are fully vectorizable, and traditional vectorization is able to closely match the performance of modulo scheduling. This result is expected since fully vectorizable loops do not induce distribution and require no communication between vector and scalar operations. In contrast, many of the loops in the other benchmarks are only partially vectorizable. For these applications, traditional vectorization underperforms the baseline by a wide margin. For example, `nasa7` sees a slowdown of almost $2\times$.

One reason for this loss in performance is the high cost of communicating operands between scalar and vector register files. The traditional vectorizer transfers operands using scalar expansion. We see this overhead in Figure 6-5, which lists the dynamic instruction counts resulting from each technique. We would expect vectorization to decrease total memory operations (scalar + vector) since vector accesses transfer two operations per in-

Benchmark	Traditional	Full	Selective
093.nasa7	0.52	0.66	1.03
125.turb3d	0.68	0.81	1.01
301.apsi	0.64	0.94	1.02
146.wave5	0.94	0.98	1.02
103.su2cor	0.93	0.97	1.13
172.mgrid	0.90	0.97	1.26
171.swim	1.00	1.00	1.32
101.tomcatv	0.95	0.99	1.35

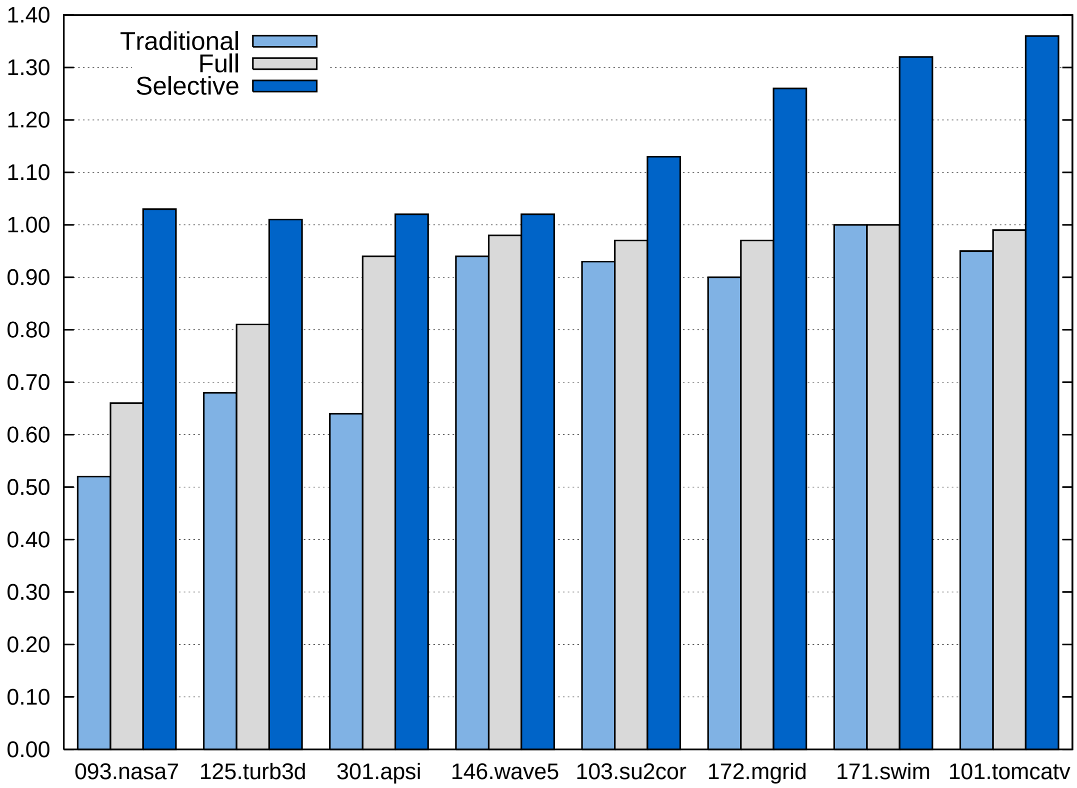


Figure 6-4: Speedup of vectorization techniques compared to modulo scheduling.

struction. As the table shows, traditional vectorization actually increases the number of memory operations in the first four benchmarks. The additional memory operations also increase address arithmetic. For `nasa7`, traditional vectorization uses nearly 3× as many ALU instructions as modulo scheduling. While the other vectorization schemes also communicate operands through memory, modulo scalar expansion does not significantly increase the number of ALU operations.

The center bars in the graph of Figure 6-4 show speedups for full vectorization. The improvement over traditional vectorization results from the abandonment of loop distribution. For many loops, non-unit-stride memory operations are the primary cause for loop distribu-

Benchmark	Mode	Scalar ALU	Scalar FPU	Vector FPU	Scalar Mem	Vector Mem	Total	IPC
093.nasa7	swp	17.87	20.91	0.00	23.62	0.00	64.74	3.69
	trad	52.47	1.43	10.38	29.39	12.87	127.08	3.78
	full	21.50	2.49	9.09	30.00	10.44	79.41	2.99
	sel	18.85	17.11	1.98	20.06	2.54	65.07	3.83
125.turb3d	swp	47.91	26.66	0.00	34.81	0.00	114.09	2.54
	trad	80.46	0.81	16.77	32.66	15.73	171.99	2.62
	full	59.62	0.83	18.60	42.55	16.39	151.63	2.75
	sel	48.16	25.64	0.54	33.44	1.34	114.86	2.59
301.apsi	swp	65.94	47.66	0.00	52.22	0.00	171.32	1.56
	trad	161.53	21.12	22.18	75.89	25.80	348.48	2.02
	full	68.68	14.15	16.37	55.67	14.32	180.83	1.54
	sel	68.80	37.49	5.43	48.95	5.72	175.97	1.63
146.wave5	swp	6.34	4.21	0.00	5.06	0.00	16.57	1.32
	trad	8.51	1.45	1.43	4.12	1.48	19.75	1.48
	full	6.50	1.34	1.44	4.00	1.36	16.55	1.29
	sel	6.47	3.08	0.57	4.12	0.81	16.68	1.36
103.su2cor	swp	26.28	28.88	0.00	25.13	0.00	83.24	2.41
	trad	32.10	2.95	12.98	12.67	9.06	82.81	2.23
	full	27.96	3.04	12.92	12.94	9.01	76.40	2.15
	sel	26.91	16.15	6.38	19.58	5.64	82.91	2.72
172.mgrid	swp	58.11	98.76	0.00	96.29	0.00	255.55	4.46
	trad	69.22	0.13	48.62	11.66	46.27	226.55	3.58
	full	58.98	0.13	48.59	12.13	46.23	212.19	3.58
	sel	63.53	31.93	33.60	38.66	32.38	233.16	5.12
171.swim	swp	15.21	37.56	0.00	27.55	0.00	81.41	3.98
	trad	16.21	3.23	17.19	1.63	12.97	65.56	3.29
	full	15.23	3.24	17.19	1.67	12.96	64.34	3.15
	sel	16.86	15.43	11.13	13.33	10.34	77.46	5.01
101.tomcatv	swp	9.31	21.69	0.00	17.46	0.00	49.32	3.38
	trad	11.09	1.03	10.52	2.29	8.43	43.01	2.79
	full	10.37	1.07	10.40	2.69	8.43	41.75	2.84
	sel	10.90	8.64	6.62	10.61	6.14	48.60	4.53

Figure 6-5: Dynamic instruction counts $\times$ 100 million instructions and overall IPC. An *instruction* refers to individual operations as opposed to VLIW instructions. Total instruction count includes operations not listed, such as branches. For each benchmark, **swp** = baseline modulo scheduling. **trad** = traditional vectorization, **full** = full vectorization, **select** = selective vectorization.

tion. Since scatter/gather and strided memory operations are unavailable, the traditional vectorizer creates scalar loops to shuffle these operands in memory. A potential optimization would omit loop distribution and implement these memory operations with scalar code sequences. Performance would not exceed that of the full vectorization scheme, however, which never distributes loops. The `swim` benchmark is the only case for which traditional vectorization performs no loop distribution. For all other benchmarks, full vectorization provides an improvement over traditional vectorization. Nevertheless, performance does not match baseline modulo scheduling. This fact is due entirely to the overhead of operand communication. In fact, full vectorization matches the baseline in all cases if the simulator ignores this overhead. In other words, blind vectorization of all data parallel operations can saturate communication resources and degrade performance.

The rightmost bars in the graph of Figure 6-4 show the performance obtained with selective vectorization. For all benchmarks, the technique yields the best performance. By carefully monitoring resource usage, selective vectorization avoids unnecessary resource saturation and the resulting loss in performance. Better yet, there are several benchmarks for which selective vectorization produces a substantial performance improvement by utilizing both scalar and vector resources. In the best case, `tomcatv` achieves a speedup of 1.35× over modulo scheduling alone.

The dynamic instruction counts in Figure 6-5 offer two insights into the advantage of selective vectorization. First, notice that the approach tends to utilize the same number of overall memory instructions as baseline modulo scheduling. For `nasa7`, `turb3d`, and `apsi`, selective vectorization avoids unduly burdening the memory units with excessive communication. In contrast, the traditional and full vectorizers greatly increase the number of memory operations utilized by these benchmarks. For `su2cor`, `mgrid`, `swim`, and `tomcatv`, selective vectorization employs more memory operations than the other vectorization schemes. For these benchmarks, the technique takes advantage of underutilized memory units to distribute computation across vector and scalar resources. This leads us to the second insight, which is the operation balance across the scalar and vector floating-point units. In Figure 6-5, selective vectorization provides better resource utilization than the other techniques. The last four benchmarks see the most balanced usage and hence, the biggest performance gains. It is interesting to note that selective vectorization actually increases the total operation count in these benchmarks. The approach offsets this increase with a much improved instruction per cycle ratio (IPC), leading to an overall performance improvement.

Three primary factors explain the cases for which selective vectorization fails to provide a significant performance improvement. The first is dependence cycles. Loop-carried dependences limit selective vectorization because operations on a recurrence must execute on scalar hardware. Even more problematic is that these operations must execute sequentially. If a long dependence cycle constrains a loop's initiation interval, intelligent resource allocation cannot improve loop performance. Dependence cycles are the primary cause for the limited performance gain in `apsi`.

Another factor is the existence of non-unit-stride memory operations, which place a double burden on the memory units. Non-unit-stride memory operations cannot take advantage of more efficient vector opcodes. Even worse, the compiler must shuffle these operands between register files in order to use them in vector computation. This overhead means that loops containing a large number of non-vectorizable memory operations are better executed without vector opcodes. Several critical loops in `nasa7` and `wave5` contain non-unit-stride memory operations.

Finally, selective vectorization and modulo scheduling do not target several important code sequences in the benchmark suite. The most common examples are loops containing function calls or internal control flow. Several of these loops populate `wave5`. This benchmark also contains an important function that lacks loops of any kind.

An interesting benchmark is `turb3d`, for which selective vectorization does produce improved schedules for several key loops. Unfortunately, these loops tend to have low iteration counts and do not benefit from software pipelining. In general, a lower initiation interval increases the number of stages in a software pipeline, and hence, the schedule length of the prolog and epilog. With short iteration counts, the prolog and epilog can dominate execution time and actually diminish performance. In the case of `turb3d`, the slowdown in the prolog and epilog offset the performance gained with selective vectorization.

6.3 Opportunities for Selective Vectorization

Selective vectorization is adept at uncovering those cases in which full vectorization or no vectorization provide the best performance. It is possible that the performance gains presented in Section 6.2 are the result of selecting between these two alternatives on a per-loop basis. Such an approach would provide an advantage over a technique that applies the same method to every loop. Figure 6-6 verifies that there are ample opportunities for utilizing scalar and vector resources in the same loop. For each benchmark in the figure, I list the number of loops for which selective vectorization finds a schedule better than, equal to, or worse than the competing methods (*i.e.*, modulo scheduling, traditional vectorization, and full vectorization). The figure only considers loops that are resource-limited; that is,

Benchmark	Number of Loops	ResMII Better	ResMII Equal	ResMII Worse
093.nasa7	29	7 (24.1%)	22 (75.9%)	0 (0.0%)
101.tomcatv	6	4 (66.7%)	2 (33.3%)	0 (0.0%)
103.su2cor	33	25 (75.8%)	8 (24.2%)	0 (0.0%)
125.turb3d	13	6 (46.2%)	7 (53.8%)	0 (0.0%)
146.wave5	128	55 (43.0%)	73 (57.0%)	0 (0.0%)
171.swim	10	5 (50.0%)	5 (50.0%)	0 (0.0%)
172.mgrid	14	5 (35.7%)	9 (64.3%)	0 (0.0%)
301.apsi	77	18 (23.4%)	59 (76.6%)	0 (0.0%)

Benchmark	Number of Loops	II Better	II Equal	II Worse
093.nasa7	29	4 (13.8%)	25 (86.2%)	0 (0.0%)
101.tomcatv	6	4 (66.7%)	2 (33.3%)	0 (0.0%)
103.su2cor	33	25 (75.8%)	8 (24.2%)	0 (0.0%)
125.turb3d	13	6 (46.2%)	7 (53.8%)	0 (0.0%)
146.wave5	128	51 (39.8%)	73 (57.0%)	4 (3.1%)
171.swim	10	5 (50.0%)	5 (50.0%)	0 (0.0%)
172.mgrid	14	5 (35.7%)	9 (64.3%)	0 (0.0%)
301.apsi	77	18 (23.4%)	59 (76.6%)	0 (0.0%)

Figure 6-6: Number of loops for which selective vectorization finds an II better, equal to, or worse than competing techniques.

loops for which the ResMII is larger than the RecMII. No vectorization technique can improve performance in recurrence-constrained loops.

Figure 6-6 separates results into the resource-constrained II (ResMII), as computed by the modulo scheduler, and the final II obtained after scheduling. As the table shows, there are a significant number of loops for which selective vectorization provides an advantage. Furthermore, there are no loops for which it produces a ResMII higher than competing techniques. Modulo scheduling does produce inferior schedules in four loops from `wave5`. This aberration is due to the fact that iterative modulo scheduling is a heuristic. Improving resource utilization does not *guarantee* that the algorithm will produce a better schedule. Interestingly, these four loops contain long dependence cycles and heavy resource requirements. It appears that Trimaran's modulo scheduler is struggling in this situation. Some extensions to iterative modulo scheduling (*e.g.*, [35]) place special emphasis on dependence cycles. It is likely that a more sophisticated algorithm could overcome the slight inefficiency seen here.

Figure 6-7 shows the speedup of full vectorization and selective vectorization for individual loops. The loops in the figure originate from the four benchmarks that benefit most

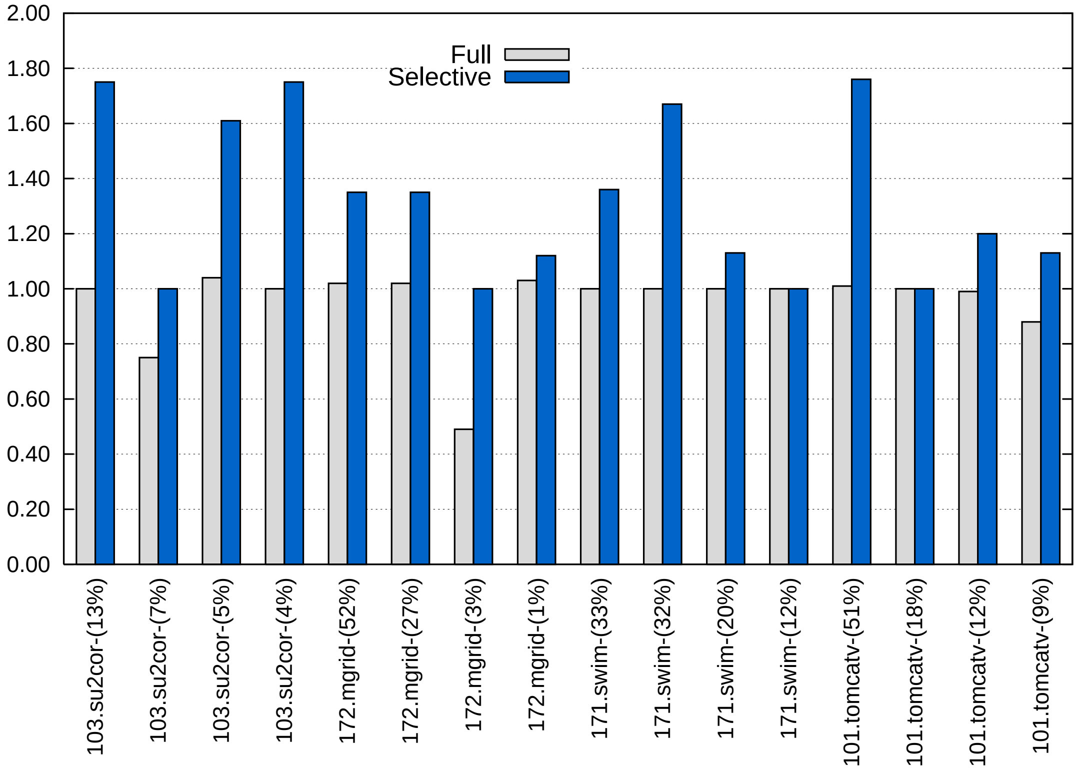


Figure 6-7: Speedup of individual loops using selective vectorization.

from selective vectorization: `su2cor`, `mgrid`, `swim`, and `tomcatv`. For each benchmark, I show results for the four targeted loops that contribute most to the benchmark's total execution time. The x-axis of Figure 6-7 shows each loop's contribution using baseline modulo scheduling. The results here reaffirm that selective vectorization offers a significant performance advantage for individual loops. In the best case, the algorithm achieves a 1.75× speedup out of a maximum of 2×.

6.4 Hardware Support for Operand Transfer

Sections 6.2 and 6.3 evaluate performance on a machine model that provides no specialized support for operand communication. This section considers two variations of the L-machine that do not require communication through memory. The first, L^c, provides two functional units which transfer single elements between the vector and scalar register files. The second, L^0, assumes no overhead for communication. In all other aspects, L^c and L^0 are identical to the baseline architecture. L^0 is a hypothetical design which provides an upper bound on

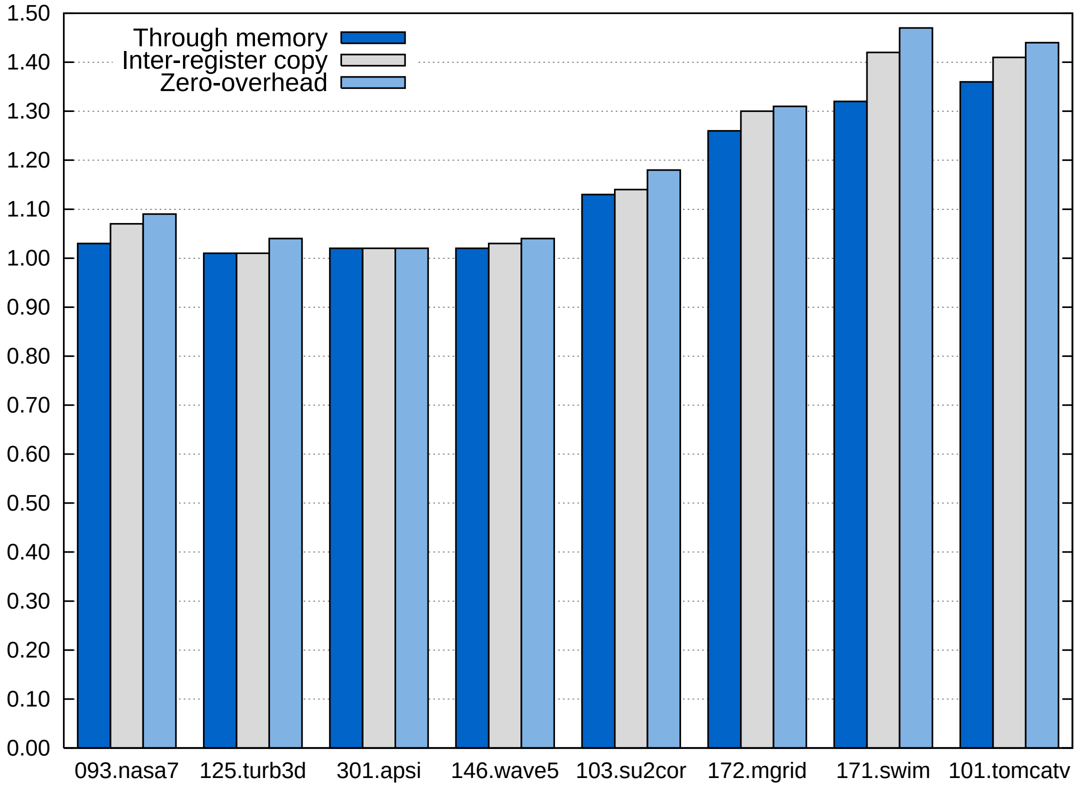


Figure 6-8: Speedup of selective vectorization for different communication models.

the performance potential of hardware communication. A practical implementation that might approach this ideal is one that overlaps the vector and scalar registers so that scalar operations can address vector elements directly. The disadvantage of this design is that it requires more register file ports to satisfy the processor's functional units.

Figure 6-8 shows the speedup of selective vectorization for the three architectural models. For consistency with previous sections, I report speedups relative to baseline modulo scheduling. The leftmost bar for each benchmark shows results for the baseline L-machine. The center and rightmost bars show speedups for the L^c and L^0 models, respectively. In the best case, L^c achieves a 1.07× speedup over the baseline architecture; L^0 achieves a maximum speedup of 1.11×.

Figure 6-9 compares individual loop performance for the different communication models. The loops shown here are the same as those in Figure 6-7. The leftmost bars show the speedup of selective vectorization for the L-machine. The center and rightmost bars show speedup for L^c and L^0, respectively. Relative to the L-machine, both models see a maximum per-loop speedup of 1.2×.

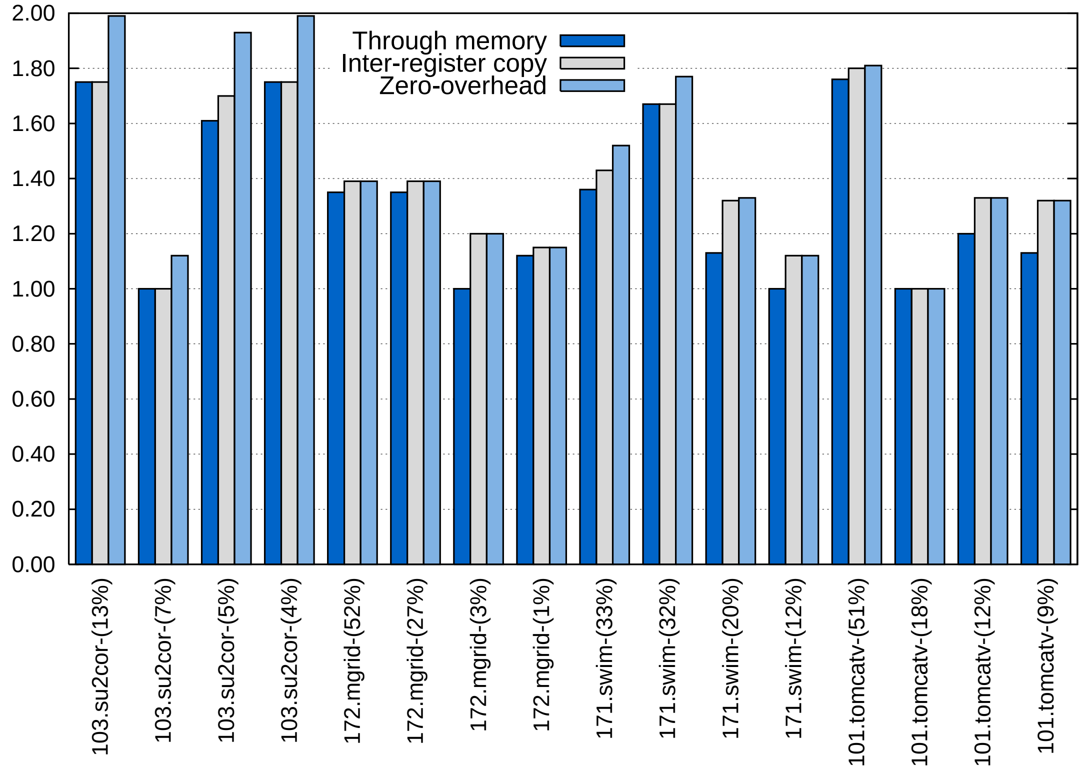


Figure 6-9: Speedup of individual loops for different communication models.

As these experiments show, a specialized communication network does not provide large performance gains for these benchmarks. A primary reason for the lack of improvement is that software pipelining hides the L-machine's long latency communication sequences. The primary benefit of a communication network, therefore, is a reduction in memory unit pressure. In many cases, however, the L-machine's two memory units suffice to carry out its communication requirements, particularly when the compiler employs more efficient vector memory operations.

Specialized communication support is more critical for an architecture with fewer memory units. For example, Figure 6-10 shows the speedup gained from selective vectorization when the L-machine supports only one memory unit. For this model, the addition of a communication network has a more drastic effect on overall performance. In the best case, L^c (with one memory unit) provides a maximum speedup of $1.23\times$ over a design that requires communication through memory.

Another factor limiting performance is issue slot pressure. In Figure 6-9, two loops achieve the maximum speedup of $2\times$. In most cases, however, performance falls short of

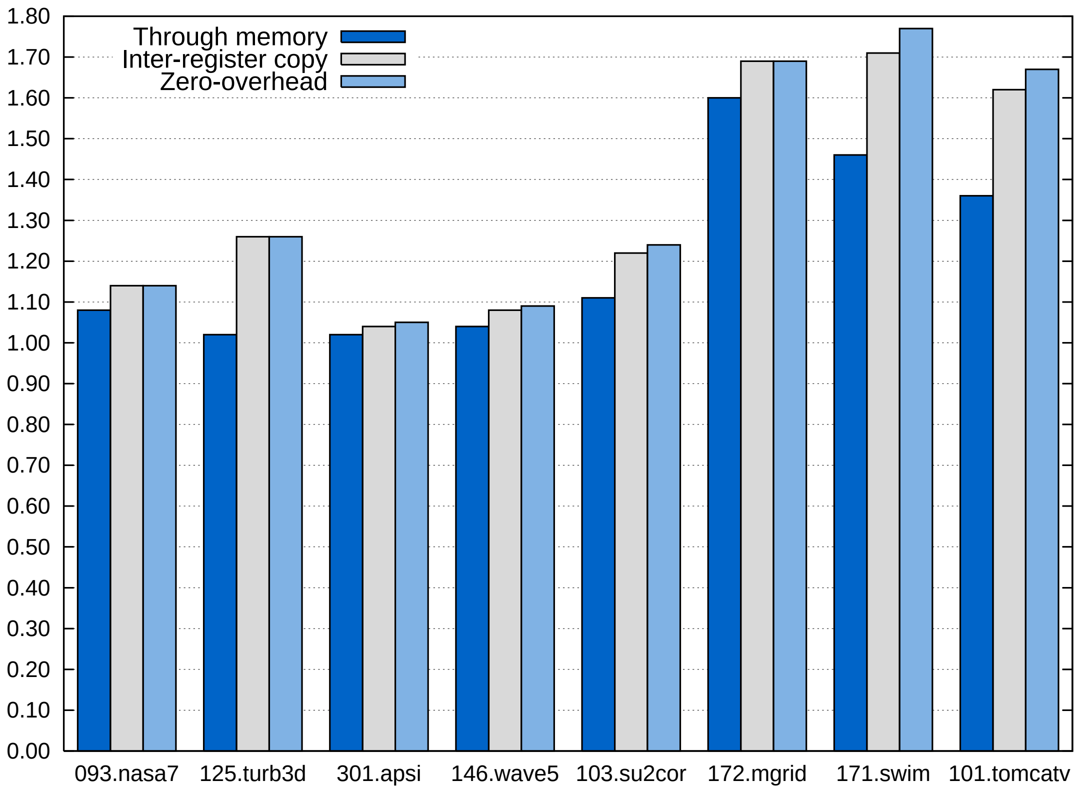


Figure 6-10: Speedup of selective vectorization for different communication models using an L-machine with one memory unit.

this ideal. With no communication overhead, we might expect L^0 to achieve the maximum speedup in more cases. Most of the loops in Figure 6-9 are limited by available issue slots. Since vector and scalar instructions share issue slots, selective vectorization has no opportunity to balance computation across separate scalar and vector versions. Since vector instructions specify multiple independent operations, issue slots achieve the highest utilization with vector opcodes. As demonstrated in Section 6.2, however, full vectorization can underutilize scalar execution resources. If issue slots are heavily-used, they can become saturated before selective vectorization can fully balance computation across the processor's other resources.

Figure 6-11 examines the performance impact of issue slots. The figure presents speedups of selective vectorization over modulo scheduling using a processor model with no communication overhead and no issue slot restriction. With this design, most of the loops achieve optimal or near-optimal speedup. For the remaining loops, a lack of vectorizable operations precludes maximum speedup. In several cases, dependence cycles inhibit selective vectoriza-

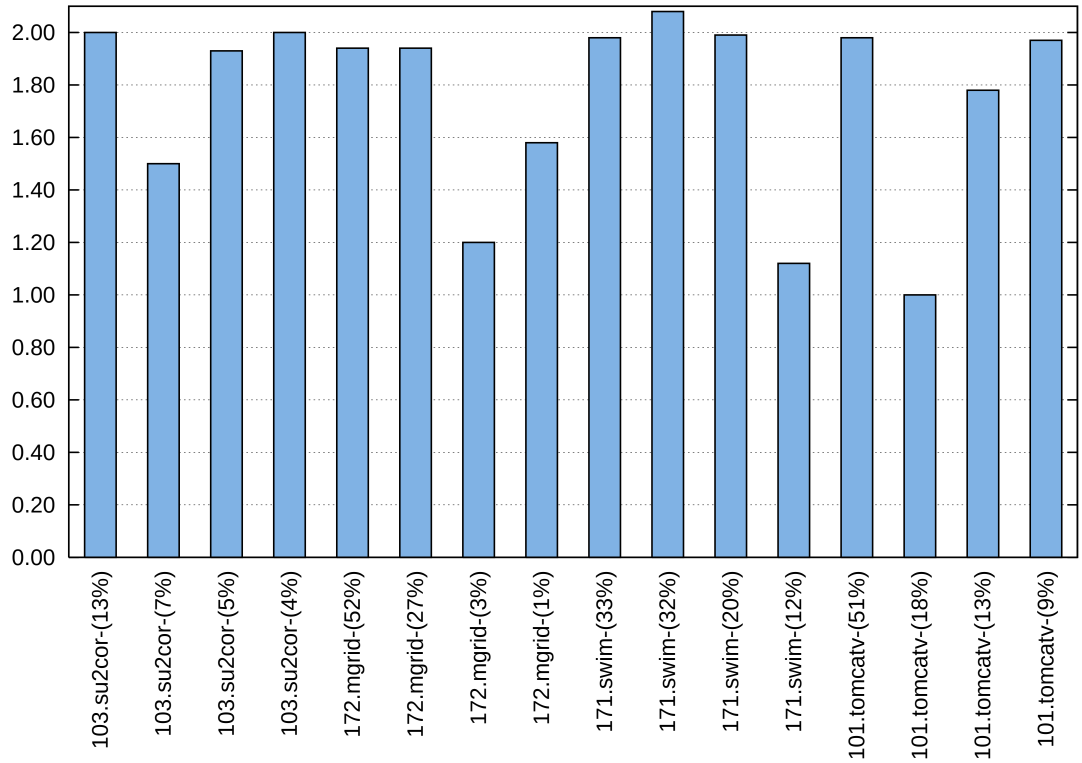


Figure 6-11: Speedup of individual loops with zero-overhead communication and unrestricted issue slot usage.

tion. For example, a long dependence cycle constrains the initiation interval for the second loop in `tomcatv`. Two loops in Figure 6-11 contain non-unit-stride memory references which the compiler must execute with scalar instructions. An interesting anomaly occurs for one loop in `swim`, which achieves a speedup above the maximum of 2×. This seeming impossibility is due to the modulo scheduling heuristic, which produces a suboptimal schedule for the baseline.

6.5 Performance Impact of Alignment Information

This section examines the effect of memory alignment on the speedups attained with selective vectorization. In the previous sections, the compiler assumed all vector memory references were misaligned. In this section, I report speedups for selective vectorization when the compiler enforces alignment with code transformation. To gain alignment information, I use the single-version peeling scheme developed in Chapter 5. In order to measure an upper bound on performance, I also introduce a new machine model, L^a, which incurs no

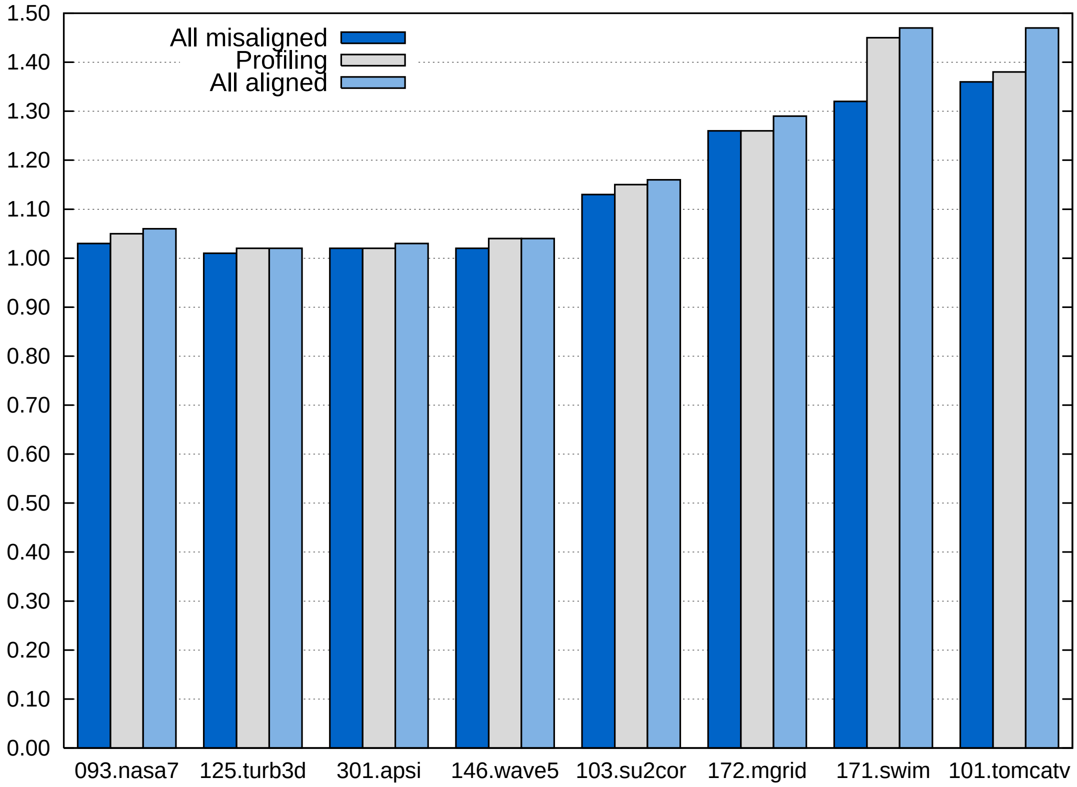


Figure 6-12: Speedup of selective vectorization without alignment information, with profiling, and with zero-overhead realignment.

overhead when executing misaligned memory references. As described in Chapter 5, single-version peeling relies on profiling to generate pre-loops. Since I use the SPEC training data sets for simulation, I gather profile information using the reference data. As demonstrated in Chapter 5, the choice of data sets is not a major concern since the training and reference data sets produce nearly identical alignment profiles. The only benchmark with a significant deviation between data sets is `apsi`, but this application does not achieve a significant performance gain from selective vectorization.

Figure 6-12 plots the speedup of selective vectorization over modulo scheduling for the different systems. For each benchmark, the leftmost bar shows speedup for the L-machine when the compiler assumes all references are misaligned. The center bars show performance using single-version peeling. Note that these results account for the overhead of dynamic peeling and runtime testing. The rightmost bars show performance for L^a. In the best case, peeling achieves an additional 1.10× speedup for `swim`, nearly matching the upper bound performance of L^a.

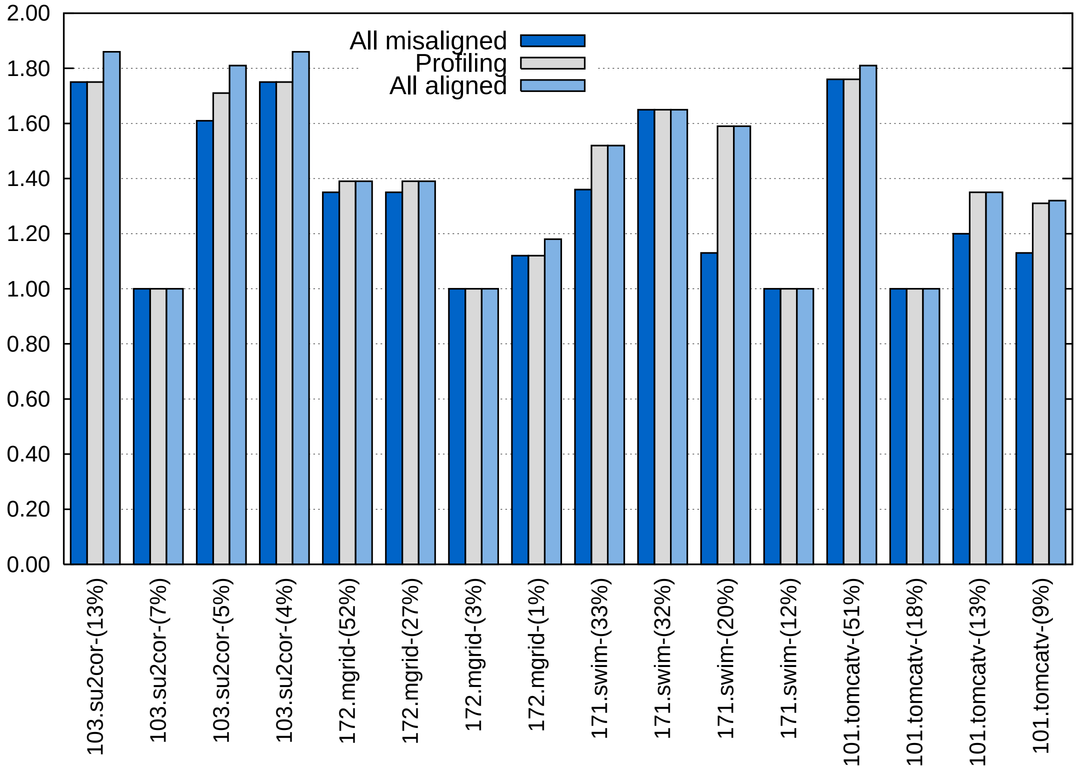


Figure 6-13: Speedup of individual loops without alignment information, with profiling, and with zero-overhead realignment.

Figure 6-13 shows individual loop performance for the different alignment systems. The loops represented here are the same as those first presented in Figure 6-7. In most cases, aligned memory references do not cause a significant performance increase. There are a few loops in `tomcatv` and `swim`, however, that see a substantial improvement. In the best case, single-version peeling produces a 1.40× speedup over an approach that assumes total misalignment.

6.6 Vector vs. Scalar Operation Throughput

This section examines the effect of vector processing power on performance. For this study, I introduce a new processor model, L^4. The design is identical to the L-machine except that it operates on vectors of four 64-bit elements. As with the L-machine, I assume the vector unit provides single-cycle throughput.

Figure 6-14 shows the speedup over modulo scheduling of traditional, full, and selective vectorization using L^4. Comparing these result to those gained for the L-machine in Fig-

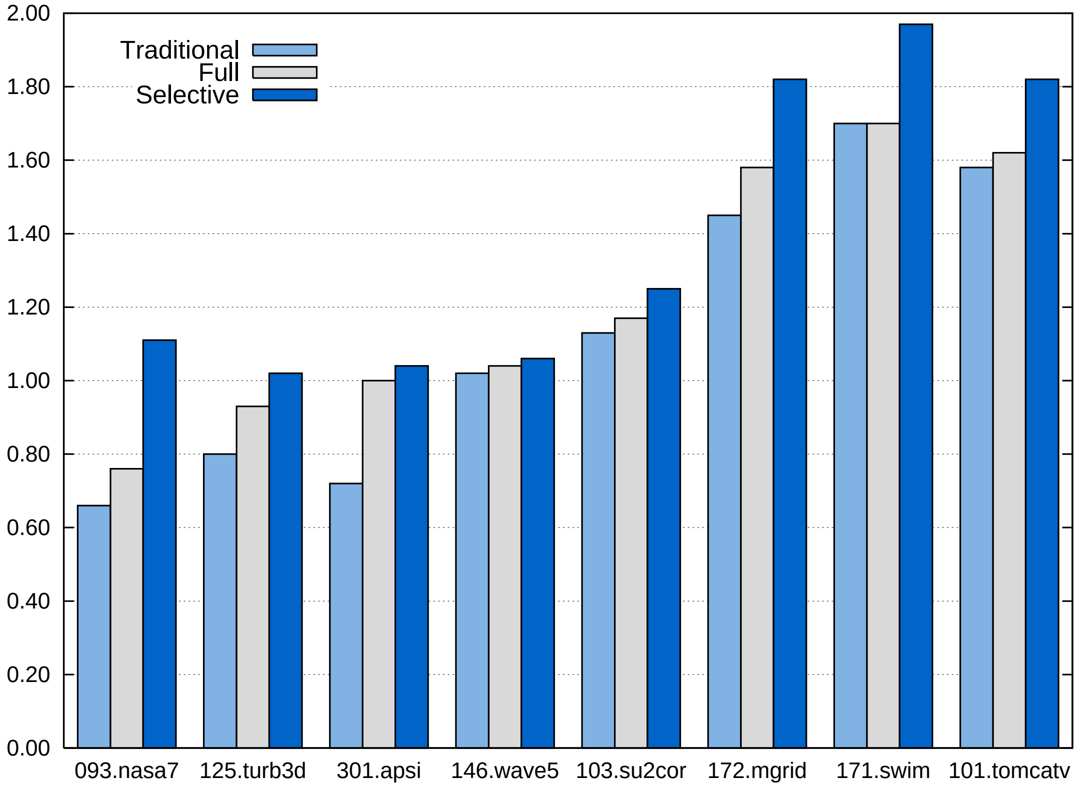


Figure 6-14: Speedup of several vectorization techniques compared to modulo scheduling for a vector length of 4.

ure 6-4, I make two observations. First, increasing the effective operation throughput of the vector unit provides further opportunities for performance improvements. In the best case, `swim` achieves a whole-program speedup of almost 2×. By comparison, the baseline L-machine reaches a maximum speedup of 1.35×. In addition, traditional and full vectorization become more attractive as we increase the processing power of the vector unit. Compared to the L-machine, fewer benchmarks experience a slowdown, and four benchmarks achieve a significant speedups.

The second observation is that the relative advantage of selective vectorization diminishes as we increase the ratio of vector to scalar operation throughput. Figure 6-15 illustrates this phenomenon more clearly. Here, I show the speedup of selective vectorization over full vectorization for the L-machine and L^4. For all benchmarks, the performance differential is less for L^4. This result is not unexpected, since increasing the operation throughput of the vector unit compared to the scalar units decreases the relative advantage of utilizing both sets of resources.

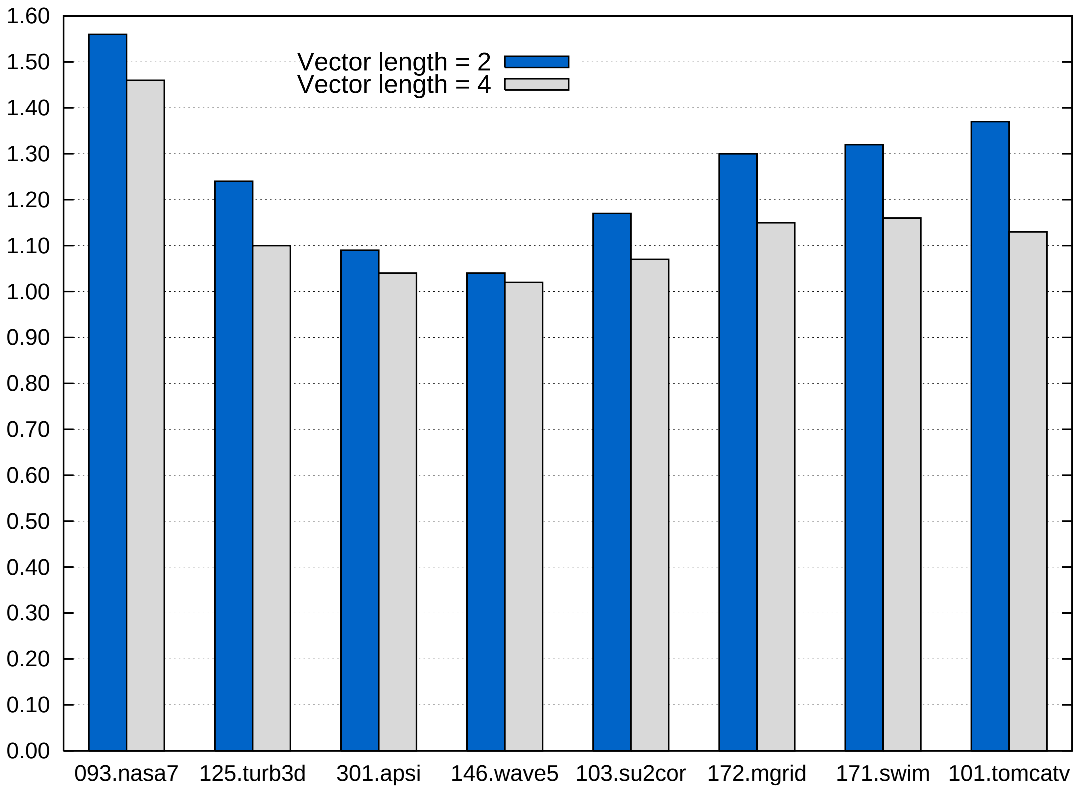


Figure 6-15: Speedup of selective vectorization over full vectorization for the L-machine and L^4.

Figure 6-16 shows the maximum achievable speedup of selective vectorization over full vectorization for various vector lengths. The figure assumes one vector operation and two scalar operations can execute each cycle. At a vector length of 2, scalar and vector operation throughput are equal, allowing a maximum speedup of 2×. As vector length increases, the attainable speedup diminishes.

Although designers have steadily increased the capabilities of short-vector extensions, there is a practical limit to the number of elements an architecture can process in parallel. Future designs are unlikely to see much increase in vector operation throughput, if they see any increase at all. For example, the Pentium 4 already executes double-precision vector operations as consecutive 64-bit computations. It is likely, therefore, that selective vectorization will remain an important compilation technique for targeting tomorrow's general-purpose designs.

Despite its diminished potential at longer vector lengths, selective vectorization still proves to be an effective method for targeting general-purpose processors with short-vector instructions. In Figure 6-14, selective vectorization is the top-performing technique in all

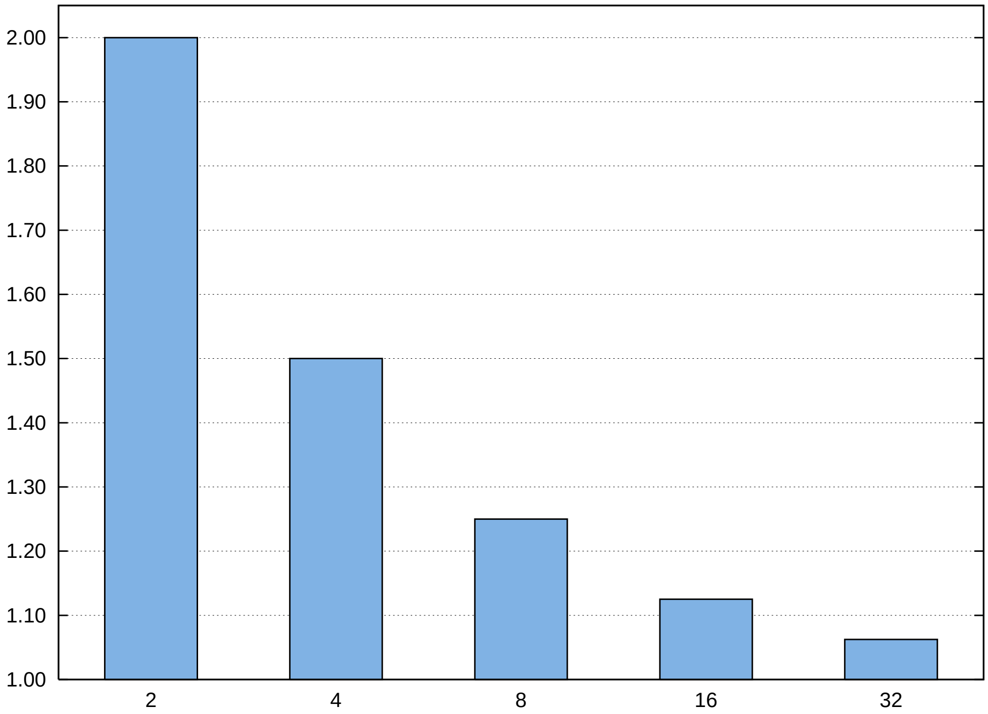


Figure 6-16: Maximum speedup of selective vectorization over full vectorization as vector length increases. Speedups assume a maximum of one vector operation and two scalar operations can execute each cycle.

cases. For benchmarks that do not benefit from vectorization, the approach avoids unnecessary resource saturation. For the remaining benchmarks, selective vectorization provides a respectable performance advantage.

Chapter 7

Conclusion

This chapter concludes the thesis by identifying limitations in my approaches and outlining areas for improvement. Section 7.1 describes the conditions for which selective vectorization is most applicable. In short, the technique is most effective when compiling fully vectorizable loops for architectures that provide equivalent operation throughput on scalar and vector resources. Section 7.2 suggests methods to increase the effectiveness of my compilation toolchain by extending the class of loops it targets. Finally, Section 7.3 summarizes the major results and contributions of the thesis.

7.1 Limitations

Selective vectorization is an effective method for improving performance in loops with data parallelism. Even when vectorizable operations exist, however, the presence of a dependence cycle can diminish opportunities for speedup. Selective vectorization boosts performance by balancing computation across vector and scalar resources. Intelligent resource allocation cannot reduce execution time if a dependence cycle constrains a loop's initiation interval. Long latency operations exacerbate the problem since the II must equal or exceed the sum of latencies on the cycle. Of course, these difficulties are not peculiar to vectorization and software pipelining, as dependence cycles fundamentally limit parallelism.

Resource sharing between vector and scalar instructions also limits selective vectorization since the approach has no opportunity to balance computation across different sets of resources. In contemporary designs, examples of shared resources include issue slots and memory units. When a shared resource limits performance, full vectorization tends to be most beneficial since vector instructions typically employ these resources more efficiently. In this sense, selective vectorization does not hold the same advantage over a conventional approach. It is important to point out, however, that selective vectorization subsumes traditional vectorization since it naturally uncovers those situations where full vectorization is most profitable.

As demonstrated in Chapter 6, selective vectorization produces the largest improvements for designs that provide equivalent vector and scalar processing power. If the operation throughput of one set of resources overwhelms the other, there is less advantage to utilizing both sets. Even when vector resources offer relatively higher throughput than scalar resources, full vectorization does not provide the best solution in all cases. For example, if appropriate vector opcodes are not available, the overheard of assembling and disassembling vector operands can eliminate the advantage of vectorization. In many cases, the trade-offs of a vector versus scalar execution are unclear. The framework described in this thesis provides a straightforward and accurate mechanism for predicting the performance of different code generation strategies.

7.2 Future Work

Loops containing few operations often present little opportunity to balance computation across vector and scalar resources. In these situations, loop unrolling can increase the prospects for selective vectorization. Consider the dot product kernel, which occurs frequently in multimedia and DSP applications:

```
for (i=0; i<N; i++) {
  s = s + a[i] * b[i];
}
```

A common optimization performs multiple partial summations and combines the results when the loop completes. This approach allows for full vectorization:

```
t[0:1] = 0;
for (i=0; i<N; i+=2) {
  t[0:1] = t[0:1] + a[i:i+1] * b[i:i+1];
}
s = s + t[0] + t[1];
```

Unrolling the vectorized loop by a factor of 2 yields

```
t[0:3] = 0;
for (i=0; i<N; i+=4) {
  t[0:1] = t[0:1] + a[i+0:i+1] * b[i+0:i+1];
  t[2:3] = t[2:3] + a[i+2:i+3] * b[i+2:i+3];
}
s = s + t[0] + t[1] + t[2] + t[3];
```

This transformation exposes additional computation in the loop body and provides greater flexibility for selective vectorization. In the absence of loop-carried dependences, the unrolled versions of a statement are independent. In this case, one approach to code selection might assign entire statements to one set of resources. Furthermore, the unroll factor need not be a multiple of the vector length. With an unroll factor of 3, for example, operations in iterations $3i$ and $3i + 1$ could execute on vector resources, while operations in iterations $3i + 2$ could execute on scalar resources. There are at least two difficulties with the approach, however. First, unroll factors that are not a multiple of the vector length guarantee misaligned vector memory references. Second, the compiler must discover an unroll factor that leads to the best resource usage. If opportunities for selective vectorization already exist, additional unrolling may be useless. In general, the compiler should employ unrolling judiciously since the transformation can place additional pressure on the instruction cache and instruction fetch unit.

Another area for future work concerns register pressure. Contemporary multimedia extensions provide separate scalar and vector registers. For these architectures, selective vectorization can reduce spilling by utilizing both register files. Higher performance might be available if the selective vectorization algorithm could predict the implications of register pressure during partitioning.

Currently, my infrastructure supports countable loops without internal control flow. An obvious extension would target a broader class of loops. Modulo scheduling can accommodate loops with control flow using if-conversion [5] or superblock and hyperblock formation [57]. These transformations convert control dependences into data dependences and allow the modulo scheduler to operate normally on straightline code. If-conversion is also a standard mechanism for vectorizing operations in control flow [6].

Extending selective vectorization to non-countable loops (*i.e.*, *while* loops) presents a more difficult problem. While modulo scheduling is possible, resolution of the branch condition limits the scheduler's ability to overlap iterations. In other words, a new iteration cannot begin until the branch condition from the current iteration resolves. When possible, speculative execution can relax this constraint [75]. Vectorization of non-countable loops is possible with hardware support [7].

Selective vectorization would also benefit from any transformations that expose additional data parallelism. Loop interchange [4] and reduction recognition [6] would be especially beneficial. In cases where a loop nest does not access consecutive locations of a multidimensional array, loop interchange might reorder the loops to create unit-stride references in the inner loop. Reduction recognition enables vectorization of reductions.

Finally, the evaluation in Chapter 6 does not employ the most sophisticated alignment transformations. For example, the per-statement peeling scheme discussed in Section 5.8 might enable further improvements, particularly if we combine that approach with profiling and runtime testing. Also, my current formulation performs pre-loop construction and selective vectorization in separate phases. As a result, the compiler may generate runtime tests for memory operations that it does not vectorize. A more sophisticated approach might combine alignment transformations and selective vectorization in a single phase.

7.3 Summary

Short-vector extensions are prevalent in general-purpose and embedded microprocessors. In order to provide transparency to the programmer, the compiler must target these extensions automatically and efficiently. Current solutions typically adopt technology first developed for vector supercomputers. Compared to these designs, multimedia extensions offer unique challenges. One of the most important issues is the alignment restriction placed on vector memory references. This thesis describes a suite of tools to manage this complexity, including a dataflow analysis to extract alignment information at compile-time, as well as transformations that detect alignment at runtime. The most practical method combines profiling with dynamic peeling and runtime testing, and is able to enforce alignment for roughly 75% of dynamic memory references.

This thesis also introduces selective vectorization, a method for balancing computation across vector and scalar resources. Compared to traditional techniques, selective vectorization provides improved resource utilization and leads to software pipelines with shorter initiation intervals. Selective vectorization operates in the back-end, where it measures vectorization decisions by their performance impact on a specific target architecture. An important consequence is that selective vectorization accurately accounts for explicit communication of operands between scalar and vector instructions. Even when operand communication requires memory operations, the technique is sufficiently sophisticated to provide large performance gains. I compare selective vectorization to traditional techniques using a realistic VLIW processor model. On a set of SPEC FP benchmarks, the technique achieves whole-program speedups of up to 1.35×. For individual loops, selective vectorization provides speedups of up to 1.75×.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Alex Aletà, Josep M. Codina, Jesús Sánchez, and Antonio González. Graph-Partitioning Based Instruction Scheduling for Clustered Processors. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 150–159, Austin, TX, December 2001.

[3] Jr. Alexander Vanderburgh. *TX-2 Users Handbook, Lincoln Manual No. 45.* Massachusetts Institute of Technology Lincoln Laboratory, Lexington, MA, July 1961.

[4] John R. Allen and Ken Kennedy. Automatic Loop Interchange. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 233–246, Montreal, Quebec, June 1984.

[5] J.R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, Austin, TX, January 1983.

[6] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, San Francisco, California, 2001.

[7] Krste Asanović. *Vector Microprocessors.* PhD thesis, University of California at Berkeley, May 1998.

[8] Chris Basoglu, Woobin Lee, and John Setel O'Donnell. The MAP1000A VLIW Mediaprocessor. *IEEE Micro*, 20(2):48–59, March 2000.

[9] Aart Bik, Milink Girkar, Paul Grey, and Xinmin Tian. Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems. *Intel Technology Journal Q1*, 2001. http://developer.intel.com.

[10] Aart J.C. Bik. *The Software Vectorization Handbook: Applying Multimedia Extensions for Maximum Performance.* Intel Press, Hillsboro, OR, 2004.

[11] Aart J.C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Experiments with Automatic Vectorization for the Pentium 4 Processor. In *Proceedings of the 9th Workshop on Compilers for Parallel Computers*, pages 1–10, Edinburgh, Scotland, June 2001.

[12] Aart J.C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic Intra-Register Vectorization for the Intel Architecture. *International Journal of Parallel Programming*, 30(2):65–98, April 2002.

[13] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural Constant Propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 152–161, Palo Alto, CA, June 1986.

[14] David A. Carlson, Ruben W. Castelino, and Robert O. Mueller. Multimedia Extensions for a 550-MHz RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 32(11):1618–1624, November 1997.

[15] Gerald Cheong and Monica Lam. An Optimizer for Multimedia Instruction Sets. In *Second SUIF Compiler Workshop*, August 1997.

[16] Josep M. Codina, Jesús Sánchez, and Antonio González. A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, pages 175–184, Barcelona, Spain, September 2001.

[17] Alain Darte. On the complexity of loop fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 149–157, Newport Beach, CA, October 1999.

[18] James C. Dehnert, Peter Y.T. Hsu, and Joseph P. Bratt. Overlapped Loop Support in the Cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, Boston, MA, April 1989.

[19] Derek J. DeVries. A Vectorizing SUIF Compiler: Implementation and Performance. Master's thesis, University of Toronto, June 1997.

[20] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, March 2000.

[21] Alexandre E. Eichenberger and Edward S. Davidson. Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 180–191, Ann Arbor, MI, November 1995.

[22] Alexandre E. Eichenberger and Edward S. Davidson. Efficient Formulation for Optimal Modulo Schedulers. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 194–205, Las Vegas, NV, June 1997.

[23] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing Compiler for the CELL Processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Saint Louis, MO, September 2005.

[24] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 82–93, Washington, DC, June 2004.

[25] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Squeezing More CPU Performance out of a CRAY-2 by Vector Block Scheduling. In *Proceedings of Supercomputing '88*, pages 237–246, Orlando, FL, November 1988.

[26] Roger Espasa, Mateo Valero, and James E. Smith. Vector Architectures: Past, Present and Future. In *Proceedings of the 12th International Conference on Supercomputing*, pages 425–432, Melbourne, Australia, July 1998.

[27] C.M. Fiduccia and R.M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Conference on Design Automation*, pages 175–181, 1982.

[28] Jose Fridman and Zvi Greenfield. The TigerSHARC DSP Architecture. *IEEE Micro*, 20(1):66–76, January 2000.

[29] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Professional, 2005.

[30] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 85–94, San Jose, CA, November 1994.

[31] Dan Grove and Linda Torczon. Interprocedural Constant Propagation: A Study of Jump Function Implementations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, Albuquerque, NM, June 1993.

[32] Craig Hansen. MicroUnity's MediaProcessor Architecture. *IEEE Micro*, 16(4):34–41, August 1996.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, pages G2–G54. Morgan Kaufman Publishers, third edition, 2003.

[34] Glenn Hinton, Dave Sager, Mike Upton, Darell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processors. *Intel Technology Journal*, 5(1), February 2001.

[35] Richard A. Huff. Lifetime-Sensitive Modulo Scheduling. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 258–267, Albuquerque, NM, June 1993.

[36] IBM Corporation. *IBM PowerPC 970FX RISC Microprocessor*, 2005.

[37] IBM Corporation. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*, 2005.

[38] IBM XL C/C++ and Fortran compilers.
http://www-306.ibm.com/software/awdtools/xlcpp/.

[39] ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, 1985.

[40] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 1: Instruction Set Reference*, October 2002.

[41] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*, June 2005.

[42] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, June 2005.

[43] ISO/IEC 9899:TC2, Programming Languages — C, May 2005.

[44] Paul Kalapathy. Hardware-Software Interactions on Mpact. *IEEE Micro*, 17(2):20–26, March 1997.

[45] Ken Kennedy and Kathryn S. McKinley. Typed Fusion with Applications to Parallel and Sequential Code Generation. Technical Report CRPC-TR94646, Rice University, 1994.

[46] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49:291–307, February 1970.

[47] Patrick Knebel, Barry Arnold, Mick Bass, Wayne Kever, Joel D. Lamb, Ruby B. Lee, Paul L. Perez, Stephen Undy, and Will Walker. HP's PA7100LC: A Low-Cost Superscalar PA-RISC Processor. In *Compcon Spring '93, Digest of Papers*, pages 441–447, San Francisco, CA, February 1993.

[48] Les Kohn and Neal Margulis. Introducing the Intel i860 64-Bit Microprocessor. *IEEE Micro*, 9(4):15–30, August 1989.

[49] Christos Kozyrakis and David Patterson. Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 283–293, Istanbul, Turkey, November 2002.

[50] Christos Kozyrakis and David Patterson. Overcoming the Limitations of Conventional Vector Processors. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 399–409, San Diego, CA, June 2003.

[51] Andreas Krall and Sylvain Lelait. Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming*, 28(4):347–361, August 2000.

[52] Alexei Kudriavtsev and Peter Kogge. Generation of Permutations for SIMD Processors. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 147–156, Chicago, IL, June 2005.

[53] Ashok Kumar. The HP PA-8000 RISC CPU. *IEEE Micro*, 17(2):27–32, March 1997.

[54] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[55] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC, June 2000.

[56] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, Charlottesville, VA, September 2002.

[57] Daniel M. Lavery and Wen-mei W. Hwu. Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 126–137, Paris, France, December 1996.

[58] Ruby Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[59] Josep Llosa, Eduard Ayguadé, Antonio Gonzalez, Mateo Valero, and Jason Eckhardt. Lifetime-Sensitive Modulo Scheduling in a Production Environment. *IEEE Transactions on Computers*, 50(3):234–249, March 2001.

[60] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. Vector Register Design for Polycyclic Vector Scheduling. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 154–163, Santa Clara, CA, April 1991.

[61] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. Register Requirements of Pipelined Processors. In *Proceedings of the 6th International Conference on Supercomputing*, pages 260–271, Washington, D.C., July 1992.

[62] Cameron McNairy and Don Soltis. Itanium 2 Processor Microarchitecture. *IEEE Micro*, 23(2):44–55, March 2003.

[63] MIPS Technologies, Inc. *MIPS Extension for Digital Media with 3D*, March 1997.

[64] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, California, 1997.

[65] Dorit Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[66] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a SIMdD DSP Architecture. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 2–11, San Jose, CA, October 2003.

[67] Erik Nystrom and Alexandre E. Eichenberger. Effective Cluster Assignment for Modulo Scheduling. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 103–114, Dallas, TX, December 1998.

[68] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow! Technology: Architecture and Implementations. *IEEE Micro*, 19(2):37–48, March 1999.

[69] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[70] Ivan Pryanishnikov, Andreas Krall, and Nigel Horspool. Pointer Alignment Analysis for Processors with SIMD Instructions. In *Proceedings of the 5th Workshop on Media and Streaming Processors*, pages 50–57, San Diego, CA, December 2003.

[71] Steve Purcell. The Impact of Mpact2. *IEEE Signal Processing Magazine*, 15(2):102–107, March 1998.

[72] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, 20(4):47–57, July 2000.

[73] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, San Jose, CA, November 1994.

[74] B. Ramakrishna Rau. Iterative Modulo Scheduling. Technical Report HPL-94-115, Hewlett Packard Company, November 1995.

[75] B. Ramakrishna Rau, Michael S. Schlansker, and P.P. Tirumalai. Code Generation Schema for Modulo Scheduled Loops. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, Portland, OR, December 1992.

[76] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions and Trade-offs. *Computer*, 22(1), January 1989.

[77] B.R. Rau, M. Lee, P.P. Tirumalai, and M.S. Schlansker. Register Allocation for Software Pipelined Loops. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 283–299, San Francisco, CA, June 1992.

[78] Norman J. Rohrer, Miles Canada, Erwin Cohen, Mat Ringler, Mike Mayfield, Peter Sandon, Paul Kartschoke, Jay Heaslip, James Allen, Peter McCormick, Thomas Pfluger, Jeff Zimmerman, Cedric Lichtenau, Tobias Werner, Gerard Salem, Mike Ross, David Appenzeller, and Dana Thygesen. PowerPC 970 in 130nm and 90nm Technologies. In *IEEE International Solid-State Circuits Conference*, pages 68–69, San Francisco, CA, February 2004.

[79] Richard M. Russel. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[80] John Ruttenberg, G.R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 1–11, Philadelphia, PA, May 1996.

[81] Jesús Sánchez and Antonio González. Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 124–133, Monterey, CA, December 2000.

[82] Harsh Sharangpani. Itanium Processor Microarchitecture. *IEEE Micro*, 20(5):24–43, September 2000.

[83] Jaewook Shin, Jacqueline Chame, and Mary Hall. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architecture. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 45–55, Charlottesville, VA, September 2002.

[84] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 165–175, San Jose, CA, March 2005.

[85] Standard Performance Evaluation Corporation. http://www.spec.org/cpu95.

[86] N. Sreraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming*, 28(4):363–400, August 2000.

[87] Ju-ho Tang, Edward S. Davidson, and Johau Tong. Polycyclic Vector Scheduling vs. Chaining on 1-Port Vector Supercomputers. In *Proceedings of Supercomputing '88*, pages 122–129, Orlando, FL, November 1988.

[88] Robert E. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[89] Shreekant Thakkar and Tom Huff. Internet Streaming SIMD Extensions. *IEEE Computer*, 32(12):26–34, December 1999.

[90] Marc Tremblay, Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, August 1996.

[91] Trimaran Research Infrastructure. http://www.trimaran.org.

[92] J.T.J. van Eijndhoven, F.W. Sijstermans, K.A. Vissers, E.J.D. Pol, M.J.A. Tromp, P. Struik, R.H.J. Bloks, P. van der Wolf, A.D. Pimentel, and H.P.E. Vranken. TriMedia CPU64 Architecture. In *International Conference on Computer Design*, pages 580–585, Austin, TX, October 1999.

[93] VAST-C/AltiVec. http://www.crescentbaysoftware.com.

[94] Codeplay VectorC. http://www.codeplay.com.

[95] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[96] Michael J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, California, 1996.

[97] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 153–164, San Jose, CA, March 2005.

[98] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An Integrated Simdization Framework Using Virtual Vectors. In *Proceedings of the 19th ACM International Conference on Supercomputing*, pages 169–178, Cambridge, MA, June 2005.

[99] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 160–169, Austin, TX, December 2001.