Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2009-064 December 18, 2009

Selective Vectorization for Short-Vector Instructions

Samuel Larsen, Rodric Rabbah, and Saman Amarasinghe

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

Selective Vectorization for Short-Vector Instructions

Samuel Larsen* VMware
Rodric Rabbah* IBM Research
Saman Amarasinghe MIT CSAIL

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. Vector instructions introduce a data-parallel component to processors that exploit instruction-level parallelism, and present 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, however, the compiler must target these extensions with complete transparency and consistent performance.

This paper describes *selective vectorization*, a technique for balancing computation across a processor's scalar *and* vector units. 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 resources. We formulate selective vectorization in the context of software pipelining. Our approach creates software pipelines with shorter initiation intervals, and therefore, higher performance.

A key aspect of selective vectorization is its ability to manage transfer of operands between vector and scalar instructions. Even when operand transfer is expensive, our technique is sufficiently sophisticated to achieve significant performance gains. We 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×.

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 [37], MAX-2 [23], MMX [28], 3DNow! [27], SSE [29], SSE2 [13], and VMX/AltiVec [9]. 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, however, they must be completely invisible to the high-level programmer.

The primary component of a multimedia extension is a set of short vector instructions that 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 [33], and more recently in vector microprocessors such as T0 [4] and VIRAM [19]. 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.

Early multimedia extensions provided a modest set of 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,

*This research was conducted while the authors were at MIT CSAIL.

multimedia extensions have grown more 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.

The earliest method for accessing short-vector instructions in high-level languages was the use of 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 requires the programmer to have 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, initial 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 vector instructions is to use compiler technology to target them automatically. This approach 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 processor resources 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. One of the most important issues is vector code selection.

Traditional vector compilation focuses on the processor's vector resources. Many general-purpose processors dedicate similar processing power to scalar and vector execution units. Neglecting one set of resources in favor of the other can leave a significant portion of the processor underutilized. This paper introduces *selective vectorization*, a compilation technique for targeting both vector and scalar resources efficiently. The approach considers the loop's overall resource requirements, and *selectively* vectorizes only the most profitable data-parallel operations. We formulate selective vectorization in the context of software pipelining. Our technique leads to better overlap among loop iterations, and therefore, higher performance.

A unique feature of selective vectorization is that it operates in the compiler back-end. This approach contrasts with traditional vectorization, which operates on a high-level representation. A back-end approach allows us to more accurately measure the trade-offs of code selection alternatives on a specific target architecture. Knowledge of the target processor is particularly important when distributing dependent operations across vector and scalar resources. Many processors provide no uderlying support for communicating operands between vector and scalar instructions. For example, the VMX extension [16] implements separate vector and scalar register files; data transfer must occur through memory using a series of load and store instructions. Intuitively, this high cost of communication could quickly negate the beneifts of selective vectorization. As we will show, however, our technique still provides large performance gains. We 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× over existing approaches. For individual loops, selective vectorization provides speedups of up to 1.75×.

We organize the remainder of this paper as follows: Section 2 demonstrates the performance potential of selective vectorization using two simple examples. Section 3 describes the specific algorithm we have developed to perform selective vectorization. Section 4 evaluates our approach on a realistic VLIW architecture with short-vector extensions. Section 5 identifies limitations in our formulation and outlines areas for future work. Section 6 discusses related work, and Section 7 summarizes the paper.

2 Motivation

A popular approach for targeting short-vector instructions is to directly adopt technology developed for vector supercomputers. Traditional vectorization, however, is not well-suited for compilation to multimedia extensions and can 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

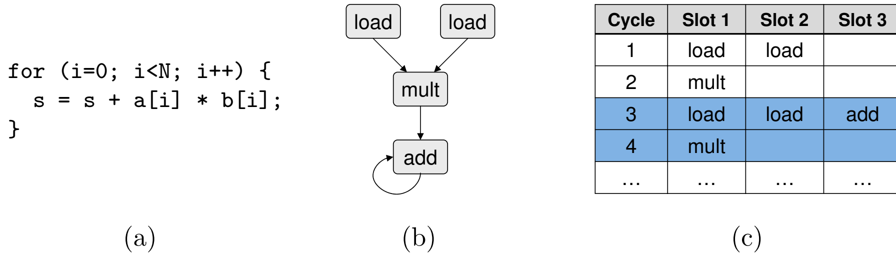

Figure 1: (a) Dot product kernel. (b) Its data-dependence graph. (c) A valid modulo schedule.

vectorizes as many operations as possible. This approach is problematic for general-purpose processors which, in contrast to 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 processor resources underutilized.

As an example of the problems created by traditional vectorization, consider the dot product kernel in Figure 1 (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* (for a review of modulo scheduling, we refer the reader to [30]). Part (c) highlights the *kernel* 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. Modulo scheduling describes the schedule length of the kernel as the *initiation interval*, or II. The schedule in part (c) has an II of 2 cycles.

If the target architecture supports short-vector instructions, the dot product in Figure 1 (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 2 (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 architecture. 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 vectors of two elements. After vectorization of the code segment in Figure 2 (a), it is reasonable to apply software pipelining to both loops in order to gain as much performance as

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

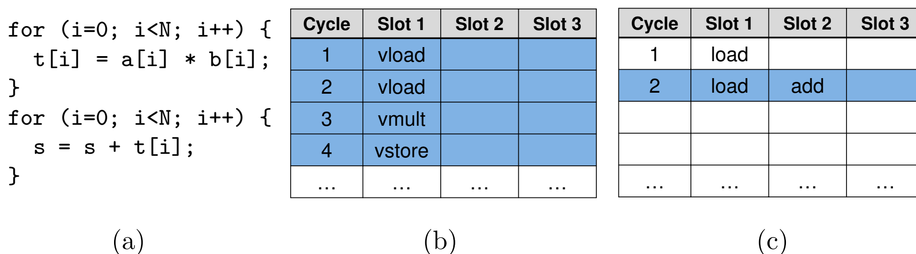

Figure 2: (a) Vectorization of a dot product using loop distribution. (b) Modulo schedule for the vector loop. (c) Modulo schedule for the scalar loop.

```
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 3: (a) Vectorized dot product without distribution. (b) Its modulo schedule.

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 2 (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, traditional vectorization does not outperform baseline modulo scheduling in this example.

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 3, 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 operation throughput 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 3 (b) provides a performance improvement over baseline modulo scheduling, but does not make the most efficient use of 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 4 illustrates this approach. The schedule in part (b) utilizes all issue slots in the kernel and executes a maximum of one vector operation each cycle.

This example illustrates the potential of balancing computation across scalar and vector resources, but overlooks a crucial detail. Namely, the software pipeline in Figure 4 (b) does not account for explicit operand communication between vector and scalar instructions. The schedule assumes that operands retrieved via

```
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 4: (a) Selectively vectorized dot product. (b) Its modulo schedule.

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

vector loads are immediately available to scalar multiplications. Modern multimedia extensions implement separate scalar and vector register files; transferring operands between register files 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 we will demonstrate, judicious code selection can still produce large performance gains.

As an example of selective vectorization for a realistic architecture, we now target a hypothetical VLIW processor, which we term the L-machine. The model is representative of contemporary designs and has a resource set similar to IBM's PowerPC 970 [15] and Intel's Itanium 2 [24]. Section 4 performs an extensive evaluation of selective vectorization using the L-machine. Figure 5 lists the high-level processor details that are pertinent to the immediate discussion.

Consider the main inner loop in `tomcatv`, an application from the SPEC CFP95 benchmark suite [35]. Figure 6 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 loop for execution on the L-machine. The data dependence graph contains no cyclic dependences, which makes the loop a prime candidate for software pipelining. In general, an absence of dependence cycles allows a software pipeline to hide operation latency, making the target processor's available resources the primary limit to 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.

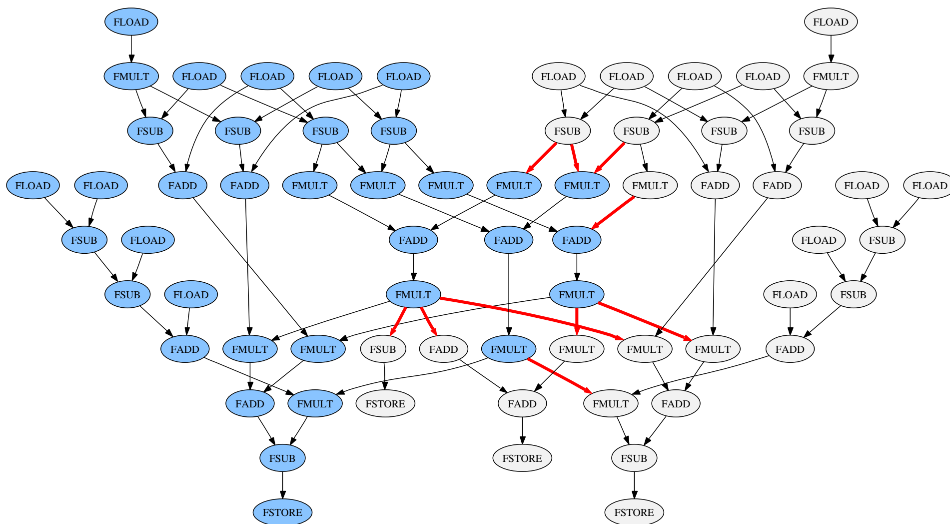

Figure 6: Data-dependence graph for the primary loop in `tomcatv`. Highlighted nodes are those chosen for vectorization using the selective vectorization algorithm described in this paper. Highlighted edges indicate dataflow between vector and scalar operations.

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

Figure 7: Resource usage and resulting initiation interval of competing compilation techniques.

As an alternative to a strictly ILP-based approach, we might choose to vectorize the loop. The absence of dependence cycles also means the loop is fully vectorizable. A traditional vectorizing compiler would replace all operations in Figure 6 with the corresponding vector instructions. After vectorization, we could software pipeline the vector loop to hide vector operation 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. Traditional vectorization combined with software pipelining therefore achieves an average initiation interval of 23 cycles, matching the performance of software pipelining alone.

Figure 6 also shows the results of selective vectorization using the algorithm described in this paper. Shaded nodes represent those chosen for vectorization; the remainder execute on scalar resources. Figure 7 summarizes the workload of the relevant processor resources resulting from the competing compilation techniques. The operation allocation of Figure 6 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 other techniques.

In Figure 6, 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 high latency of transmitting operands through memory.

3 Selective Vectorization

This section describes the details of our selective vectorization algorithm. Section 3.1 first describes the intermediate representation on which our algorithm operates. Section 3.2 describes how we extend a two-cluster partitioning heuristic to perform selective vectorization, and Section 3.3 gives a detailed example of the algorithm. The approach is applicable to architectures that export a simple instruction set architecture. Section 3.4 extends the method to support designs with complex interfaces. Section 3.5 describes a code generation technique we developed to ensure efficient communication between vector and scalar operations.

3.1 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) that should closely represent the target machine's ISA. This approach contrasts with the traditional method which operates on a source-level IR. Vectorization of 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).

[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.

- 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 [3] to extract vectorizable subexpressions.
- Selective vectorization can more accurately compare code selection alternatives. The algorithm in Section 3.2 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 and instruction selection.

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 loops explicitly. Practically, software pipelining already requires a loop identification analysis which we can leverage for selective vectorization.
- Low-level representations typically employ three-operand format (*i.e.*, two source operands and one destination operand). To produce a three-operand IR, the compiler introduces temporary registers to hold intermediate values of source-level expressions. In a naïve system, these temporaries create loop-carried dependences that prevent vectorization. Fortunately, we can easily privatize [3] scalar temporaries to remove false dependences.
- Memory dependence analysis is more effective for high-level representations. Our infrastructure performs this 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 presents the only real difficulty. An effective optimizing compiler performs induction-variable recognition and strength reduction [25] in order to reduce 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 instruction. Without it, array references require several arithmetic operations. Induction variable optimization is actually advantageous when targeting multimedia extensions. Contemporary processors only provide vector memory operations that access contiguous locations; optimized induction variables provide 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;
}
```

```
// 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 8: Partitioner pseudocode, part 1.

3.2 Algorithm Description

This section describes the specific algorithm we have implemented to perform selective vectorization. Before discussing the details, however, it is important to emphasize that the compiler performs software pipelining, in the form of modulo scheduling [30], directly following selective vectorization. The use of software pipelining has strong implications for the design of our 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, we adapt a mincut partitioning heuristic due to Fiduccia and Mattheyses [12], who credit their minimization technique to Kernighan and Lin [18]. Figures 8 and 9 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. Our 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 8, lines 6–17 constitute a complete *iteration*. Within each iteration, the algorithm repartitions each vectorizable operation exactly once. At each *step*, the heuristic

```
// 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)
29  high ← 0
30  sum ← 0
31  usage ← GET-USAGE (currPartition)
32  foreach r ∈ RESOURCES
33     cycles ← ⌈usage[r] / NUM-UNITS (r)⌉
34     high ← MAX (high, cycles)
35     sum ← sum + cycles^2
36  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)
37  foreach r ∈ RESOURCES
38     usage[r] ← 0
39  foreach op ∈ OPS
40     if currPartition[op] = SCALAR
41        for i ← 1 to VECTOR-LENGTH
42           usage ← ADD-USAGE (SCALAR-OPCODE (op), usage)
43     else
44        usage ← ADD-USAGE (VECTOR-OPCODE (op), usage)
45     if COMMUNICATION-NEEDED (op, currPartition)
46        foreach c ∈ COMMUNICATION-OPS (op, currPartition)
47           usage ← ADD-USAGE (c, usage)
48  return usage

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

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

Figure 9: Partitioner pseudocode, part 2.

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 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. Upon completion, 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 config-

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

Figure 10: Architectural model for the partitioning example in Figures 11 and 12.

uration's ResMII, or minimum initiation interval implied by resource usage [30]. Rau observed that modulo scheduling is successful in achieving the theoretical minimum II in the vast majority of cases. In his experiments, the heuristic discovered the minimum II for 96% of loops. Based on this result, we 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 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 execute instructions of type I, the weight of R is

$$\left\lceil \frac{\text{Number of operations of type } I \times \text{Occupancy of } I}{\text{Number of units in } R} \right\rceil \tag{1}$$

For now, we assume that partitioning can compute the ResMII using this method. Section 3.4 describes a scheme for extending the cost function to accommodate more complex interfaces. In Figure 9, lines 37–48 compute usage in 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 transferring 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, we 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.

3.3 Partitioning Example

To better describe the selective vectorization algorithm, we now present a simple example which details the key steps. In this section, we target the architectural model in Figure 10, with compiler-visible resources consisting of 3 memory units, 2 scalar units, and a single vector unit. Vector instructions operate on vectors of 2 elements. All operations have single-cycle throughput. We assume the target architecture requires explicit instructions to transfer 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.

We use the data dependence graph in Figure 11 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, we show the weight of each resource. The ResMII is the maximum among the memory, scalar, and vector units. We 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

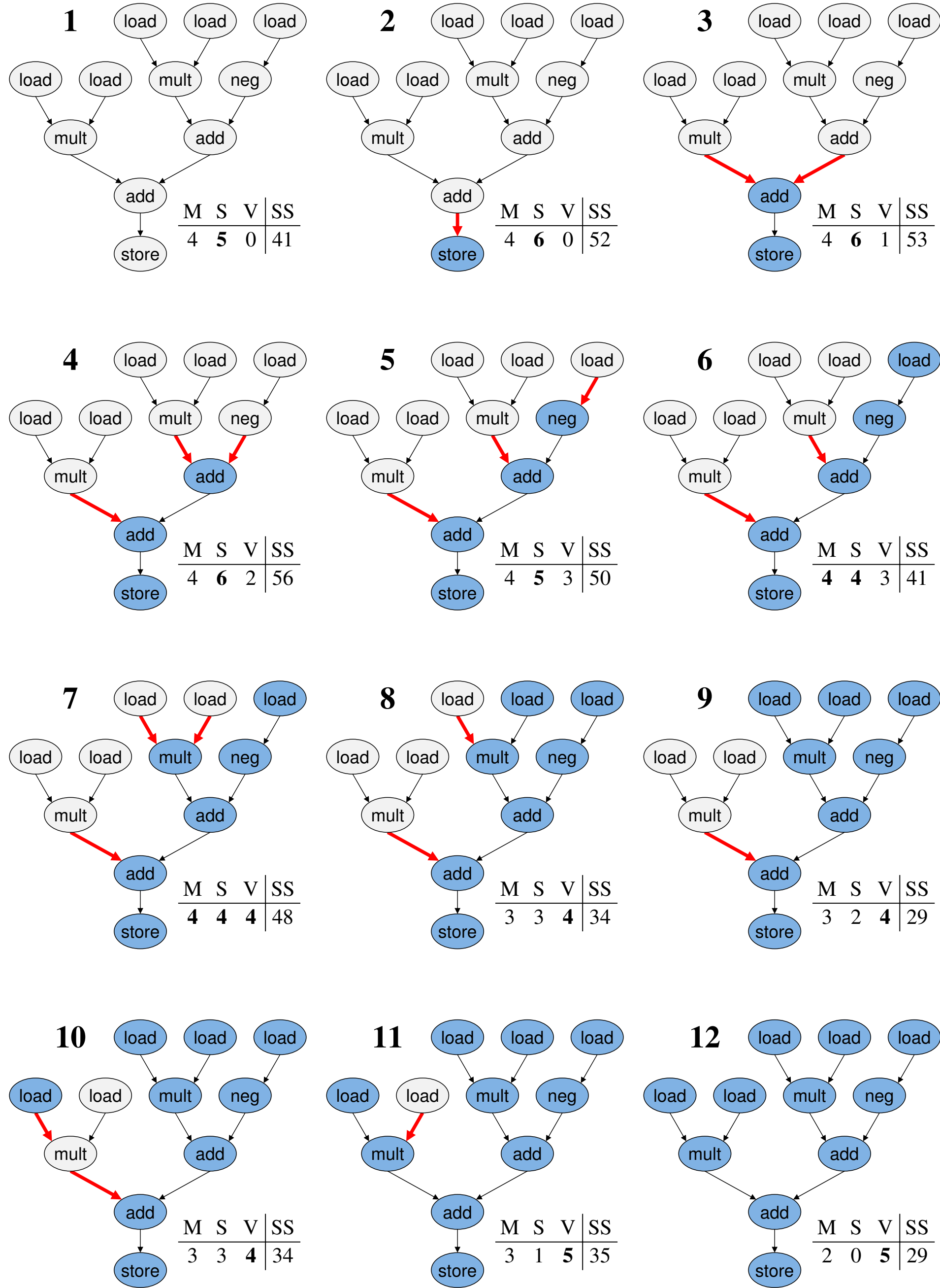

Figure 11: Example of the two-cluster partitioning algorithm for selective vectorization. This figure shows the 12 steps in the first iteration of the algorithm. Highlighted nodes designate operations currently in the vector partition. Highlighted edges indicate dataflow between scalar and vector operations. 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.

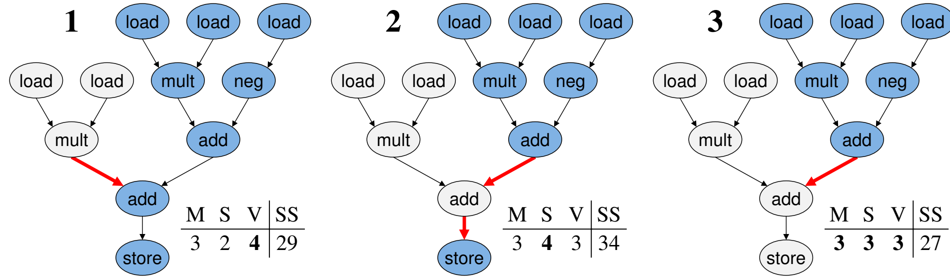

Figure 12: Second iteration for the example of Figure 11.

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, we 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 transfer operations implied by the partition. For example, the weight of the scalar unit in step 2 includes two explicit transfer instructions. These 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 11, 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 selects the alternative with the lower sum of squares value. In this case, the configuration in step 9 is the best choice. Figure 12 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 13 shows a valid modulo schedule for the final partition in Figure 12. 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.

3.4 Cost Calculation for Complex Interfaces

Many architectures provide simple interfaces that allow 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 ar-

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 13: Modulo schedule for the optimal partition in Figure 12. Numbers in parentheses indicate the pipeline stage for each operation.

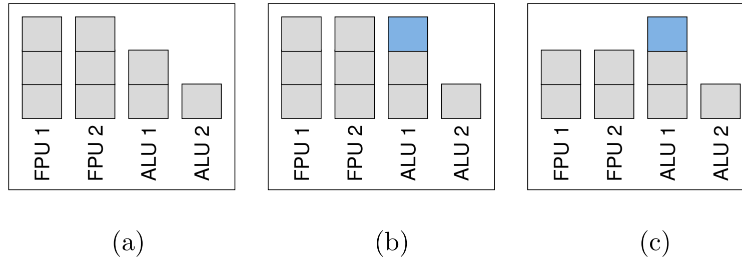

Figure 14: Bin-packing example.

chitecture provides sets of homogeneous resources that operate on strict subsets of the opcode space. Most general-purpose processors implement this interface. Nevertheless, there are many examples of architectures for which Equation 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 correct operation. Pioneers of modulo scheduling developed their approach for VLIW architectures with complex interfaces [8, 32]. 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, we leverage the ResMII calculation described by Rau for iterative modulo scheduling [30]. 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 9 so that a configuration's cost calculation and cost comparison use the same formula.

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, our experiments reveal many cases where this approach is too costly for a practical compilation system. To improve running time, we 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. This process repeats for each unlocked operation. Only after selecting and repartitioning an operation does the algorithm perform a fresh bin-packing.

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 14, 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 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. Our cost function avoids this situation by balancing operations across resources even when they do not immediately contribute to the maximum weight.

The overall result is that the bin-packing cost analysis is extremely effective in maintaining accuracy. Section 4 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 3.2. 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 our infrastructure, the algorithm's running time is comparable to modulo scheduling.

In the worst case, the modified selective vectorization algorithm 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 our benchmark suite, 85% of loops reach a solution within two iterations. Across all loops, we observe a maximum of 5 iterations.

3.5 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 arises because each sequence of transfer instructions 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;
}
```

We term this technique *modulo scalar expansion*, due to its similarity to both scalar expansion and modulo variable expansion [20]. 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 next 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[3]. With its low cost and scheduling flexibility, modulo scalar expansion is essential for absorbing the high cost of communicating through memory.

4 Evaluation

This section evaluates the selective vectorization algorithm presented in Section 3. Section 4.1 describes our compilation and simulation infrastructure. Section 4.2 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 4.3 examines performance for individual loops. We 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×.

[3] As with modulo variable expansion [20], 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.

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 15: Configuration of the L-machine.

4.1 Methodology

We used SUIF [39] as our 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, we used Trimaran, a compilation and simulation infrastructure for VLIW architectures [38]. Trimaran is one of a few publicly available infrastructures that provide a high-quality modulo scheduler. Additionally, its machine description facility allowed us to evaluate various architectural design points. With Trimaran, our evaluation focuses on statically scheduled machines. Since superscalar processors also benefit from automatic vectorization and instruction scheduling, however, we expect our techniques will be applicable to these designs as well.

For all benchmarks, we applied a suite of standard optimizations before vectorization (induction-variable recognition, register promotion, common subexpression elimination, copy propagation, constant propagation, dead code elimination, and loop-invariant code motion [25]). Among these transformations, induction variable optimization is particularly important. This optimization replaces address arithmetic with an induction variable, greatly reducing address calculations in array-based code. In our infrastructure, it also provides the mechanism for identifying unit-stride memory operations. This is important since contemporary multimedia extensions only support vector references that access contiguous memory locations.

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. We identify vectorizable operations using the classic approach described by Allen and Kennedy [3] and Wolfe [40]. As discussed in Section 3.1, however, our implementation operates on a low-level IR. The primary difficulty of this approach is obtaining accurate memory dependence information. Our infrastructure analyzes dependences at the source-level and preserves the information for the back-end.

Figure 15 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 [24] and IBM's PowerPC 970 [15]. The L-machine provides vector opcodes for all standard computational instructions. Vector instructions have the same latency as their scalar counterparts. The architecture provides one vector unit for vector floating-point computation. Since this section evaluates performance on a set of floating-point benchmarks, a vector integer unit is not relevant.

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 16: Evaluated benchmarks.

As with the VMX extension [16], the L-machine does not provide special support for transferring operands between scalar and vector registers. 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 15 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, we extended the infrastructure with rotating vector registers and predicated vector operations. For designs that do not provide rotating registers, a compiler can employ *modulo variable expansion* [20] to generate valid schedules. Predication enables the compiler to reduce code size by using the same static instructions for the kernel, prolog, and epilog [8]. It also provides a mechanism for modulo scheduling loops with control flow, but these loops are not targeted in our infrastructure.

Figure 16 summarizes the benchmarks evaluated in this paper. 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 our infrastructure was able to detect some amount of data parallelism. In order to keep simulation times practical, we evaluate performance using the SPEC training data sets. Some of the benchmarks in Figure 16 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 our compilation and simulation infrastructures do not support these techniques, however, we disregard cache performance in our simulations.

In keeping with contemporary processors, the L-machine's vector operands comprise a total of 128 bits. The benchmarks in Figure 16 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.

4.2 Selective Vectorization vs. Traditional Approaches

In this section, we compare selective vectorization to conventional compilation strategies. First, we implemented a *traditional* vectorizer in SUIF based on the scheme described by Allen and Kennedy [3]. The transformation operates on a high-level intermediate representation. To ensure highly optimized code, we 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 discussed in Section 2, a straightforward approach tends to create a series of distributed loops. Therefore, we perform loop fusion [7, 17] after vectorization to reduce overhead as much as possible. Aside from scalar expansion, we did not employ any optimizations specifically designed to expose data parallelism. Transformations such as loop interchange [2] and reduction recognition would create further opportunities for vectorization. The focus of this paper 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.

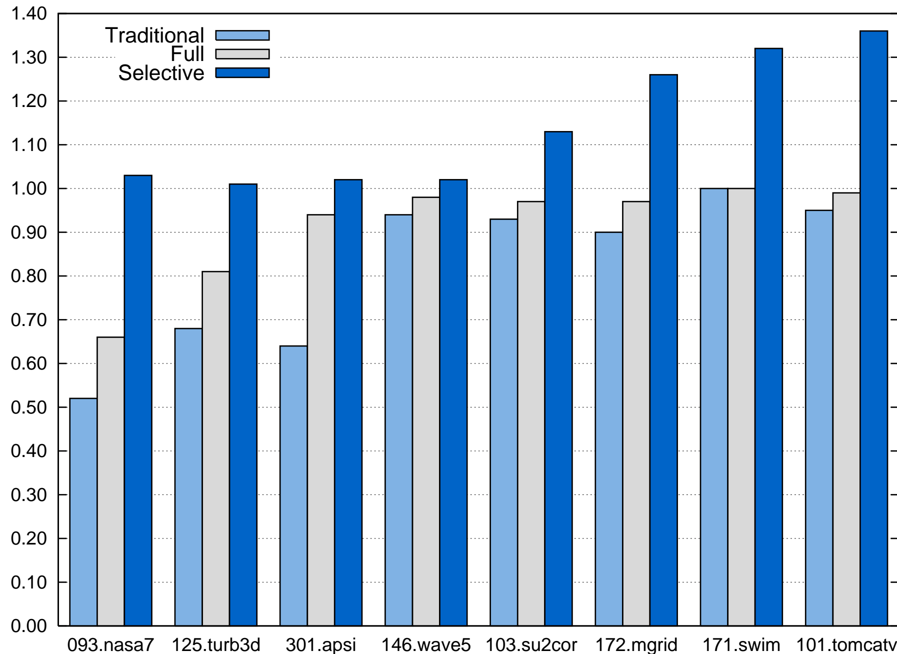

Figure 17: Speedup of vectorization techniques compared to modulo scheduling.

To study the effect of loop distribution, we also implemented a second vectorizer in the back-end. In the results that follow, we 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 3.5. For both vectorizers, we perform modulo scheduling after vectorization.

Given the heavy cost of transferring data between vector and scalar register files, we 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, we do not utilize any analyses that detect aligned memory references [5, 10, 21, 41]. For all vectorization schemes, the compiler assumes every vector memory operation is misaligned. We remove much of the resulting overhead, however, by reusing data produced in the previous iteration [10, 41].

We present all speedups relative to iterative modulo scheduling [30]. For this baseline, we unroll countable loops twice (for a vector length of 2) 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 17 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×.

The center bars show speedups for full vectorization. The improvement over traditional vectorization is due to the abandonment of loop distribution. For many loops, non-unit-stride memory operations are the primary cause for loop distribution. 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 17 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.

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. Furthermore, 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, our infrastructure does 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 altogether.

An interesting benchmark is `turb3d`, for which selective vectorization does produce improved schedules for several key loops. While these loops constitute a large portion of the benchmark's overall execution time, the innermost loops tend to have low iterations counts and therefore 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.

4.3 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 4.2 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 18 verifies that there are ample opportunities for utilizing scalar and vector resources in the same loop. For each benchmark in the figure, we 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, loops for which a recurrence does not limit performance. No resource allocation technique can improve performance in recurrence-constrained loops.

Benchmark	Number of Loops	ResMII		
		Better	Equal	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	Equal	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 18: Number of loops for which selective vectorization finds an II better than, equal to, or worse than competing techniques.

Figure 18 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 worse 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.*, [14]) place special emphasis on dependence cycles. It is likely that a more sophisticated algorithm could overcome the slight inefficiency seen here.

Figure 19 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 from selective vectorization: `su2cor`, `mgrid`, `swim`, and `tomcatv`. For each benchmark, we show results for the four targeted loops that contribute most to the benchmark's total execution time. The x-axis of Figure 19 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.

5 Future Work

Our current implementation often finds little opportunity to balance computation across vector and scalar resources in loops that contain few operations. In these situations, loop unrolling might 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];
}
```

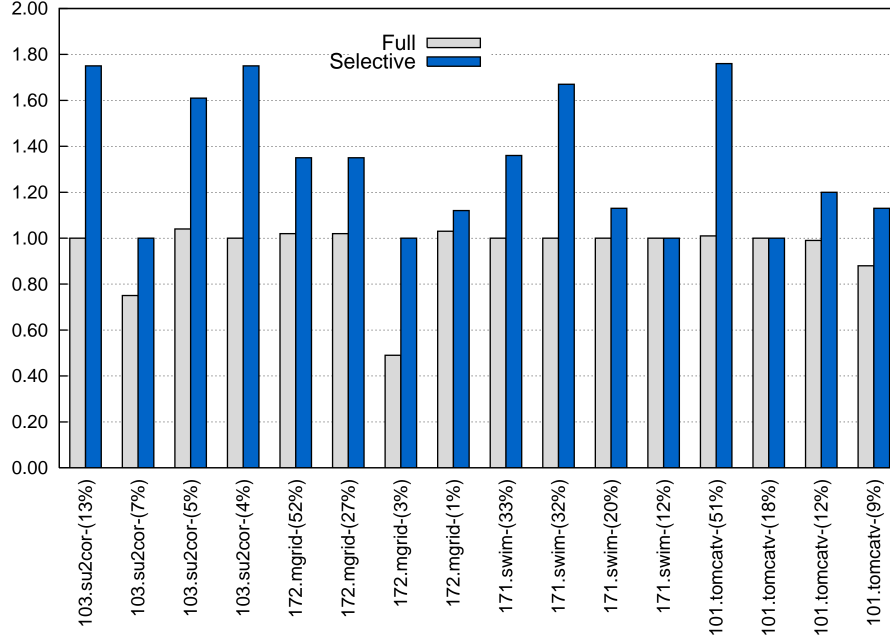

Figure 19: Speedup of individual loops using selective vectorization.

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 a full statement to either the vector or scalar unit. 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.

Currently, our 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 [3] or superblock and hyperblock formation [22]. 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 [3].

Extending selective vectorization to non-countable loops presents a more difficult problem. While modulo scheduling is possible, resolution of the branch condition limits the compilers's ability to overlap iterations. That is, a new iteration cannot begin until the branch condition from the current iteration resolves. When possible, speculative execution can relax this constraint [31]. Vectorization of non-countable loops is possible with hardware support [4].

Selective vectorization would also benefit from any transformations that expose additional data parallelism. Loop interchange [2] and reduction recognition [3] 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.

6 Related Work

Automatic extraction of parallelism is an enormous field of research. This section discusses work that is closely related to selective vectorization. To our knowledge, software pipelining of vector loops was first proposed by Tang, Davidson, and Tong [36] and Eisenbeis, Jalby, and Lichnewsky [11]. 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. 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 [1, 6, 26, 34] 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. We 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 [12, 18] described in Section 3.2 provides an intuitive match for the problem. Nonetheless, it is possible that other partitioning approaches are suitable. Regardless of the algorithm used, this paper shows that an effective technique must track resource usage carefully in order to gain top performance.

7 Conclusion

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 developed for vector supercomputers. Compared to these designs, however, multimedia extensions offer unique challenges. This paper 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 alternatives by their performance impact on a specific target architecture. An important consequence is that selective vectorization accurately accounts for explicit transfer of operands between scalar and vector instructions. Even when operand communication is costly, the technique is sufficiently sophisticated to provide large performance gains. We 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×.

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

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

[26] 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.

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

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

[29] 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.

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

[31] 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.

[32] 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.

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

[34] 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.

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

[36] 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.

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

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

[39] 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.

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

[41] 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.