Exploiting Vector Parallelism in Software Pipelined Loops

Samuel Larsen, Rodric Rabbah and Saman Amarasinghe
MIT Computer Science and Artificial Intelligence Laboratory
{slarsen,rabbah,saman}@mit.edu

Abstract

An emerging trend in processor design is the addition of short vector instructions to general-purpose and embedded ISAs. Frequently, these extensions are employed using traditional vectorization technology first developed for supercomputers. In contrast, scalar hardware is typically targeted using ILP techniques such as software pipelining. This paper presents a novel approach for exploiting vector parallelism in software pipelined loops. The proposed methodology (i) *lowers the burden on the scalar resources by offloading computation to the vector functional units,* (ii) *explicitly manages communication of operands between scalar and vector instructions,* (iii) *naturally handles misaligned vector memory operations, and* (iv) *partially (or fully) inhibits the optimization when vectorization will decrease performance.*

Our approach results in better resource utilization and allows for software pipelining with shorter initiation intervals. The proposed optimization is applied in the compiler backend, where vectorization decisions are more amenable to cost analysis. This is unique in that traditional vectorization optimizations are usually carried out at the statement level. Although our technique most naturally complements statically scheduled machines, we believe it is applicable to any architecture that tightly integrates support for instruction and data level parallelism. We evaluate our methodology using nine SPEC FP benchmarks. In comparison to software pipelining, our approach achieves a maximum speedup of $1.38\times$*, with an average of* $1.11\times$*.*

1. Introduction

Increasingly, modern general-purpose and embedded processors provide short vector instructions that operate on elements of packed data [11, 14, 23, 24, 29, 37]. Vector instructions are desirable because the vector functional units operate on multiple operands in parallel. Thus, vector instructions increase the amount of concurrent execution while maintaining a compact instruction encoding. In addition, the performance advantage is realized with moderate architectural complexity and cost.

Short vector instructions are predominantly geared toward improving the performance of multimedia and DSP codes. However, today's vector extensions also afford a significant performance potential for a large class of data parallel applications, such as floating-point and scientific computations. In these applications, as in multimedia and DSP codes, a large extent of the processing is embedded within loops that vary from fully parallel to fully sequential.

Loop-intensive programs with an abundance of data parallelism can be software pipelined, essentially converting the available parallelism to ILP. Software pipelining overlaps instructions from different loop iterations and derives a schedule that attempts to maximize resource utilization. Without explicit instruction selection that vectorizes operations, a machine's vector resources are unused and software pipelining cannot fully exploit the potential of a multimedia architecture.

Compilers that target short vector extensions typically employ technology previously pioneered for vector supercomputers. However, traditional vectorization is not ideal for today's microprocessors since it tends to diminish ILP. When loops contain a mix of vectorizable and non-vectorizable operations, the conventional approach generates separate loops for the vector and scalar operations. In the vectorized loops, scalar resources are not well used, and in the scalar loops, vector resources remain idle. In modern processors, a reduction in ILP may significantly degrade performance. This is especially problematic for VLIW processors (e.g., Itanium) because they do not dynamically reorder instructions to rediscover parallelism.

In this paper, we show that targeting both scalar and vector resources presents novel problems, leading to a new algorithm for automatic vectorization. We formulate these problems in the context of software pipelining, with an emphasis on VLIW processors. Better utilization of both scalar and vector resources leads to greater overlap among iterations, thus improving performance. Our approach remains cognizant of the loop's overall resource requirements and *selectively* vectorizes only the most profitable data parallel computations. As a result, the algorithm effectively balances computation across all machine resources.

The goal of selective vectorization is to divide operations between scalar and vector resources in a way that maximizes performance when the loop is software pipelined. Conventional strategies vectorize all data parallel operations. In loops with a large number of vector operations, this can leave scalar resources idle. Moving some operations to the scalar units can provide a more compact schedule. In other situations, full vectorization may be appropriate. This could occur when the overhead of transferring data between vector and scalar resources negates the benefit of vectorization. Alternatively, it may be advantageous to omit vectorization altogether in loops with little data parallelism. The most efficient choice depends on the underlying architectural resources, the number and type of operations in the loop, and the dependences between them.

Selective vectorization is further complicated when the compiler is responsible for satisfying complex scheduling requirements. Most architectures provide a set of heterogeneous functional units. It is not unusual for these units to support overlapping subsets of the ISA. Furthermore, the compiler may be responsible for multiplexing shared resources such as register file ports and operand networks. Also, scalar and vector operations may compete for the same resources. This is usually the case for memory operations since the same resources execute vector and scalar versions.

In this paper, we describe a union of ILP via software pipelining and DLP (data level parallelism) via short vector instructions (Section 3). We demonstrate that even fully vectorizable loops benefit from selective vectorization (Section 4). Our algorithm operates on a low-level IR. This approach is more suitable for emerging complex architectures since the impact of the optimization is measured with respect to actual machine resources. A backend approach also allows us to examine the interaction between vectorization and other backend optimizations, specifically software pipelining. This leads to a natural combination of two techniques that are generally considered alternatives. To our knowledge, no literature exists that proposes the partial vectorization we advocate in this paper.

We have implemented selective vectorization in Trimaran, a compilation and simulation infrastructure for VLIW architectures. Trimaran includes a large suite of optimizations geared toward improving ILP. It also includes a parametric cycle-accurate simulation environment. Our approach offers significant performance gains on the various architectural configurations we simulated. Compared to Trimaran's optimizations, which include software pipelining using modulo scheduling [31], our optimization yields a $1.11\times$ speedup on a set of SPEC FP benchmarks.

2. Motivating Example

We use the dot product in Figure 1(a) to illustrate the potential of selective vectorization. The data dependence graph is shown in part (b). For clarity, we omit address calculations. Consider a target architecture with three issue slots as the only compiler-visible resources, and single-cycle latencies for all operations. A modulo schedule for the loop is shown in part (c). In a modulo schedule, the *initiation interval* (II) measures the constant throughput of the software pipeline. In the schedule of part (c), two cycles are needed to execute four instructions, resulting in an II of 2.0.

Often, reductions similar to that shown in Figure 1 are vectorizable using multiple partial summations that are combined when the loop completes. Since this scheme reorders the additions, it is not valid in all cases (e.g., with floating point data). For this example, assume parallelization of the reduction is illegal, preventing vectorization of the add.

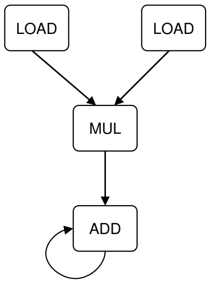
Now consider an extension to the example architecture that allows for the execution of one vector instruction each cycle, including vector memory operations. Assume vector instructions operate on vectors of length two. In the face of loop carried dependences, a traditional vectorizing compiler distributes a loop into vector and scalar portions, as shown in part (d). Scalar expansion is used to communicate intermediate values through memory.

Since the processor can issue only one vector operation each cycle, modulo scheduling cannot discover additional parallelism in the vector loop. Four cycles are needed to execute four vector operations (two loads, one multiply, and one store). This amounts to an initiation interval of 2.0, since one iteration of the vector loop actually completes two iterations of the original loop. The operations in the scalar loop can be overlapped so that an iteration completes each cycle. Overall, this results in an initiation interval of $2 + 1 = 3$, which is inferior to the performance gained from modulo scheduling alone. Even if the overhead of scalar expansion is overlooked, vectorization cannot recover from the degradation of ILP due to loop distribution.

A more effective approach is to leave the loop intact so vector and scalar operations can execute concurrently. This strategy is illustrated in Figure 1(e). Here, an II of 1.5 is achieved since the kernel completes two iterations every three cycles. Note that two scalar additions are needed to match the work output of the vector operations. In this example, we assume explicit operations are not required to communicate between scalar and vector instructions (namely, between the vector multiply and the scalar add). In reality, the cost of communicating operands must be considered. For many machines, communication incurs a large overhead as data must be transmitted through memory using a series of loads and stores.

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

(a)



(b)

Cycle	Slot 1	Slot 2	Slot 3
1	LOAD (1)	LOAD (1)	
2	MUL (1)		
3	LOAD (2)	LOAD (2)	ADD (1)
4	MUL (2)		
...	...	...	...

(c)

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

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

(d)

Cycle	Slot 1	Slot 2	Slot 3
1	VLOAD (1-2)		
2	VLOAD (1-2)		
3	VMUL (1-2)		
4	VLOAD (3-4)	ADD (1)	
5	VLOAD (3-4)	ADD (2)	
6	VMUL (3-4)		
...	...	...	...

(e)

Cycle	Slot 1	Slot 2	Slot 3
1	VLOAD (1-2)	LOAD (1)	
2		LOAD (2)	
3	VLOAD (3-4)	LOAD (3)	
4	VMUL (1-2)	LOAD (4)	
5	VLOAD (5-6)	LOAD (5)	ADD (1)
6	VMUL (3-4)	LOAD (6)	ADD (2)
...	...	...	...

(f)

Figure 1. (a) Dot product and (b) its data dependence graph. (c) Modulo schedule with an II of 2.0. Numbers in parentheses specify the original iterations to which each operation belongs. The kernel is highlighted. (d) Loop distribution into vector and scalar loops. (e) Modulo schedule for full vectorization without distribution gives an II of 1.5. (f) Modulo schedule with selective vectorization gives an II of 1.0.

Figure 1(f) illustrates that it is possible to outperform the other techniques if vectorization decisions are carried out judiciously. In the example, the selective vectorization of only one load operation leads to better resource utilization. In part (f), all issue slots are filled in the kernel, and in each cycle the maximum of one vector operation is issued. This results in an initiation interval of 1.0.

3. Selective Vectorization

The goal of selective vectorization is to divide vectorizable operations between scalar and vector resources in order to maximize loop performance. At this stage, we are concerned only with the decision of whether or not to vectorize each operation. Software pipelining, in the form of modulo scheduling [31], and register allocation are performed in subsequent phases of the compilation. In Section 4, we describe our compilation flow in more detail.

The fact that modulo scheduling follows our optimization has strong implications for our algorithm design. Specifically, the algorithm is solely concerned with balancing resource utilization, and ignores the latency of all operations. The underlying assumption is that vector operations rarely lie on dependence cycles. An exception is when the dependence distance is greater than or equal to the vector length. For example, a loop statement $a[i+4] = a[i]$ has a dependence cycle but can be vectorized for vector lengths of four or less. In practice, these situations are uncommon and dependence cycles typically prevent vectorization altogether. When an operation does not lie on a dependence cycle, its latency is of minor concern since dependent operations can be separated by pipeline stages. Although long-latency operations tend to increase the length of the pipeline's prologue and epilogue, this has little impact on performance as long as the iteration count is relatively high.

3.1. Algorithm Overview

We base our selective vectorization algorithm on Kernighan and Lin's two-cluster partitioning heuristic [16]. The algorithm is outlined in Figure 2. It divides instructions between a vector and a scalar partition. Operations are moved one at a time between the partitions, searching for a division that minimizes a cost function (described below). The algorithm is iterative. In Figure 2, lines 7-19 represent a complete iteration. All operations are originally placed in the scalar partition. Every iteration repositions each vectorizable operation exactly once (lines 10-15). With each move, the algorithm computes the cost of the resulting configuration, noting the minimum cost encountered (lines 16-18). Once each operation has been repartitioned, the configuration with the lowest cost is used as the starting point for the next iteration (line 19). The process terminates when an iteration fails to improve on its starting configuration. Ultimately, operations remaining in the vector partition are vectorized.

```
PARTITION-OPS ()
01 foreach op ∈ OPS
02    currPartition[op] ← SCALAR
03 bins ← BIN-PACK (currPartition)
04 bestPartition ← currPartition
05 bestCost ← HIGH-WATER-MARK (bins)
06 lastCost ← ∞
07 while lastCost ≠ bestCost
08    lastCost ← bestCost
09    locked ← ∅
10    foreach vectorizable op
11       bestOp ← FIND-OP-TO-SWITCH (bins, currPartition, locked)
12       currPartition ← SWITCH-OP (bestOp, currPartition)
13       locked ← locked ∪ bestOp
14       bins ← BIN-PACK (currPartition)
15       cost ← HIGH-WATER-MARK (bins)
16       if cost < bestCost
17          bestCost ← cost
18          bestPartition ← currPartition
19    currPartition ← bestPartition
20 return bestPartition

FIND-OP-TO-SWITCH (bins, currPartition, locked)
21 bestCost ← ∞
22 foreach op ∈ OPS
23    if op is vectorizable ∧ op ∉ locked
24       cost ← TEST-REPARTITION (op, bins, currPartition)
25       if cost < bestCost
26          bestCost ← cost
27          bestOp ← op
28 return bestOp

TEST-REPARTITION (op, bins, currPartition)
29 currPartition ← SWITCH-OP (op, currPartition)
   // release resources including communication overhead as necessary
30 bins ← RELEASE-RESOURCES (op, bins, currPartition)
31 bins ← RESERVE-RESOURCES (op, bins, currPartition)
32 return HIGH-WATER-MARK (bins)

BIN-PACK (currPartition)
33 foreach r ∈ RESOURCES
34    bins[r] ← 0
35 foreach op ∈ OPS
36    bins ← RESERVE-RESOURCES (op, bins, currPartition)
37 return bins

RESERVE-RESOURCES (op, bins, currPartition)
38 if currPartition[op] = SCALAR
39    opcode ← SCALAR-OPCODE (op)
40    num ← VECTOR-LENGTH
41 else
42    opcode ← VECTOR-OPCODE (op)
43    num ← 1
44 for i ← 1 to num
45    bins ← RESERVE-LEAST-USED (opcode, bins)
   // reserve resources for transfering operands, as needed
46 if COMMUNICATION-NEEDED (op, currPartition)
47    foreach c ∈ COMMUNICATION-OPS (op, currPartition)
48       bins ← RESERVE-LEAST-USED (c, bins)
49 return bins

RESERVE-LEAST-USED (opcode, bins)
50 foreach r ∈ RESOURCES-REQUIRED (opcode)
51    minHigh ← ∞
52    minCost ← ∞
53    foreach a ∈ ALTERNATIVES (r)
54       temp ← bins
         // reserve resource for appropriate number of cycles
55       temp[a] ← temp[a] + CYCLES (opcode, a)
56       high ← 0
57       cost ← 0
58       foreach r ∈ RESOURCES
59          high ← max(high, temp[r])
60          cost ← cost + temp[r]^2
61       if (high < minHigh) ∨ (high = minHigh ∧ cost < minCost)
62          minHigh ← high
63          minCost ← cost
64          best ← a
65    bins[best] ← bins[best] + CYCLES (opcode, best)
66 return bins

HIGH-WATER-MARK (bins)
67 high ← 0
68 foreach r ∈ RESOURCES
69    high ← max(high, bins[r])
70 return high

SWITCH-OP (op, currPartition)
71 if currPartition[op] = SCALAR
72    currPartition[op] ← VECTOR
73 else
74    currPartition[op] ← SCALAR
75 return currPartition
```

Figure 2. Partitioner pseudo code.

3.2. Cost Calculation

Operations are selected for repartitioning based on a cost function. We define the cost of a configuration to be the *weight* of the most heavily used resource (lines 67-70), where the weight is the number of machine cycles the resource is reserved. In modulo scheduling terminology, this corresponds to the resource-constrained minimum initiation interval (ResMII). In the absence of dependence cycles, the ResMII represents a lower bound on the II of the modulo schedule.

We compute the cost of a configuration using a bin-packing approach (lines 33-37) akin to the original formulation in modulo scheduling [31]. Namely, a bin (with zero initial weight) is associated with each compiler-visible resource. Operations are selected one at a time and assigned to the bin that minimizes the weight of the most heavily-used resource. Each placement of an operation in a bin increases the weight of that bin. If an operation reserves a resource for more than one cycle, the bin's weight is adjusted accordingly (line 55). In addition, each scalar operation is binned k times to match the work output of a single k-wide vector operation (lines 38-40).

Since communication of operands between scalar and vector operations usually requires explicit instructions, the partitioning algorithm accounts for these operations as a result of its decisions (lines 46-48). Specifically, when an operation is placed in a partition different from operations on which it is dataflow-dependent, the appropriate communication instructions are also binned. Note that a particular operand is transferred at most once since all consumers can reuse the transmitted data.

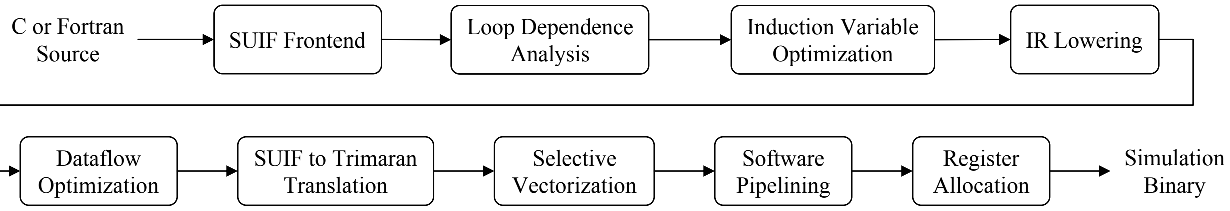


Figure 3. Compiler flow.

In the original bin-packing formulation [31], operations are selected for placement in order of their scheduling alternatives such that those with little freedom are placed first. This heuristic produces better results than unordered insertion. We make one further optimization: when two scheduling alternatives do not increase the weight of the most heavily-used resource, we select the option that minimizes the sum of the square of the bin weights (lines 53-65). This strategy tends to balance operations across bins even when these operations do not immediately affect the total cost. It also allows the partitioner to quickly compare alternatives during operation selection.

When choosing the best operation for repositioning, the algorithm does not perform a complete bin-packing for every possible move. Such an approach is prohibitively expensive since it requires n bin-packing passes before an operation is selected. Instead, the algorithm checkpoints the current state of the bins, releases the resources for the operation under consideration, and reserves the set of resources needed in the other partition (lines 29-32). For example, if an operation is moved to the vector partition, its scalar resources are released and its vector resources reserved. If necessary, we also account for any transfer operations added or eliminated in the new configuration. The cost is then recorded, the bins are restored to their original state, and the process repeats until all operations are considered. Only after an operation is finally selected for repositioning are the bins rebalanced with a fresh bin-packing (line 14). In order for this scheme to work well, it is important that all bins are balanced as much as possible during bin-packing. Otherwise, the removal of an operation tends to lead to an inaccurate cost estimate. The optimization described above provides this functionality.

Note that when an operation is moved from one partition to another, the cost of the new configuration may increase. This occurs, for example, when the producers and consumers of an operation remain in the other partition, necessitating explicit transfer instructions. Thus, while each iteration of the algorithm improves the overall cost, some steps (lines 11-12) within an iteration may increase the cost.

In the worst case, the algorithm requires n total iterations for a loop containing n vectorizable operations. Every iteration repartitions each operation once and bin-packs each new partition. Since our greedy bin-packing requires n steps, this results in an O(n^3) algorithm. In practice we observe that a solution is found after only a few iterations, making the algorithm very practical. In fact, for our benchmarks, the time spent during selective vectorization is far less than the time spent modulo scheduling. If a faster execution of the algorithm is desirable, we can artificially limit the number of iterations carried out.

3.3. Loop Transformation

If partitioning selects operations for vectorization, we next construct the transformed loop. Operations in the vector partition are replaced with their corresponding vector opcodes. Scalar operations are emitted k times, where k is the vector length. When explicit communication is required, the appropriate transfer operations are also generated. To satisfy dependences, it is important to emit operations in a specific order. Strongly connected components are sorted topologically, with operations in each component emitted in original program order. This is analogous to the loop distribution performed in traditional vectorization [6, 39]. Finally, the loop increment and upper bound are adjusted according to the vector length, and a cleanup loop is constructed for cases where the upper bound is unknown or is not a multiple of the vector length.

4. Evaluation

Figure 3 illustrates our compiler flow. We use SUIF [38] as the compiler frontend. SUIF provides accurate memory dependence information that is crucial for identifying data parallelism. Additionally, it provides a suite of existing dataflow optimizations. Our compiler backend and simulation environment are provided by Trimaran [3]. Trimaran provides a modulo scheduler and all necessary backend functionality, such as register allocation. In order to connect these two infrastructures, we implemented a SUIF-to-Trimaran IR translator. In addition, we added support for vector operations throughout the compiler.

For all benchmarks, we applied a suite of standard optimizations before vectorization. These include register promotion, common subexpression elimination, copy propaga-

Processor Parameter	Value
Issue width	6
Integer units	4
Floating-point units	2
Load/store units	2
Branch units	1
Vector units (shared int/fp)	1
Vector merge units	1
Vector length (64-bit elements)	2
Scalar integer registers	128
Scalar floating point registers	128
Vector integer registers	64
Vector floating point registers	64
Predicate registers	64

Op Type	Latency	Op Type	Latency
Int ALU	1	Fp ALU	4
Int Multiply	3	Fp Multiply	4
Int Divide	36	Fp Divide	32
Load	3	Branch	1

Table 1. Processor configuration.

tion, constant propagation, dead code elimination, induction variable optimization, and loop-invariant code motion. Aside from scalar privatization [6], we did not use any optimizations specifically designed to expose data parallelism. Advanced loop transformations would likely improve the performance of our system. However, the focus of this paper is not the identification of data parallelism, but how it can be exploited to create highly efficient schedules.

We identify vectorizable operations using the approach first developed for vector supercomputers [6, 39]. This method requires loop dependence analysis to identify dependence cycles. Operations in a dependence cycle must execute sequentially; the rest can be vectorized. The major difficulty is to accurately identify dependences between memory references. A simple approach that assumes dependence between any load/store pair usually precludes any vectorization. For array-based code, an extensive literature exists for computing dependences (see [6] for a review). After building the dependence graph, cycles are identified using Tarjan's algorithm for strongly connected components [36].

Dependence analysis is simplified by that fact that our benchmarks are coded in Fortran. Our techniques are equally applicable to languages such as C. However, further methods are typically required to cope with pointer aliasing. The simplest solution is to extend the language with directives that allow the user to convey dependence information (e.g. the `restrict` keyword). Another approach is to use a whole-program alias analysis. Perhaps the most practical solution is to insert runtime dependence checks. This is the strategy taken by the Intel compiler [7].

Selective vectorization and modulo scheduling are applied to `do` loops without control flow or function calls. In general, these are the loops to which both software pipelining and vectorization are most applicable. Innovations such as if-conversion [6] and hyperblock formation [22] have made it possible to target a broader class of loops, but they were not used in this study.

Details of the simulated architecture are shown in Table 1. Since Trimaran is an infrastructure for VLIW architectures, the evaluation below focuses on statically scheduled processors. However, note that our technique is applicable to any architecture that implements a combination of instruction level and short vector parallelism.

Our simulated processor provides one vector unit for both integer and floating point computation. Vector operations are assigned the same latency as their scalar counterparts. A separate vector unit is available for merging data from two vector registers. This is used to support misaligned vector memory operations. In our evaluation, we assume the target architecture does not allow unaligned memory operations. As a result, it is the responsibility of the compiler to merge data from adjacent regions.

Support for merging is typically provided through specialized instructions[1]. An unaligned vector load is implemented with two aligned loads followed by a merge to extract the desired elements from two registers. An unaligned store is even more expensive as data must first be loaded from memory, merged with data in registers, and then written back to memory. Fortunately, much of the overhead of misaligned memory operations can be eliminated in vectorized loops by reusing data from the previous iteration [13, 40]. This eliminates the extra memory operations otherwise required by misaligned references.

The Trimaran modulo scheduler relies on rotating registers and predicated execution. Therefore, we have extended the infrastructure with rotating vector registers and predicated vector operations. If rotating registers are not available, a similar effect is achievable with modulo variable expansion [19, 32].

Our benchmarks consist of a number of scientific applications gathered from the SPEC 92, 95, and 2000 floating point suites. These benchmarks represent the subset for which our compiler detects some degree of data parallelism. In keeping with contemporary processors, we assume vector operands comprise a total of 128 bits. Since our benchmarks operate on 64-bit data, this leads to a vector length of two. The Pentium 4 is an example of an existing architecture that supports vectors of two double-precision elements.

[1] On AltiVec, for example, merging is performed with the `vperm` instruction.

In general, our technique is most applicable where short vectors are used. As vector length increases, the processing power of the vector units begins to overwhelm that of the scalar units, and full vectorization becomes increasingly advantageous. The processor in Table 1 has a vector length of two, and therefore the throughput of the scalar units equals or exceeds that of the vector units. Selective vectorization is most applicable in such situations.

We fully simulated our benchmarks using the training input workloads to keep simulation times practical. Many of benchmarks have large memory footprints that can heavily burden the memory system. However, we have not implemented any optimizations that specifically target the memory system. As a result, our evaluation does not take memory system performance into account.

4.1. Speedup

We now compare the selective vectorization algorithm described in Section 3 to several different parallelization techniques. We have implemented a *traditional* vectorizer in SUIF based on the scheme described by Allen and Kennedy [6]. When loops contain a mix of vectorizable and non-vectorizable operations, they are distributed into separate loops. Scalar expansion is used to communicate data between loops. In a straightforward implementation, vectorization tends to create a large number of distributed loops. In order to mitigate this effect as much as possible, we perform loop fusion in the vectorizer [9].

To study the effects of loop distribution, we also implemented a second vectorizer in the backend. As in the traditional vectorizer, all data-parallel operations are vectorized. However, in this scheme the loop is left intact (i.e., not distributed) in order to expose ILP among vector and scalar operations. Scalar operations are unrolled by the vector length in order to match the work output of the vector operations. In the results that follow, we refer to this technique as *full* vectorization. In both vectorizers, all loops are modulo scheduled.

Speedups are shown relative to modulo scheduling. For this baseline, we unroll all loops k times (for vector length k) in order to lower the loop overhead and reduce computation due to address arithmetic. In the absence of vectorized operations, unrolling serves to match the benefit provided by vectorized memory operations which use one address to access multiple locations. With unrolling, the same effect is realized using *base* + *offset* addressing.

The architecture in Table 1 does not provide specialized support for communicating operands between scalar and vector functional units. Communication is accomplished through memory using a series of load and store operations. Since the resulting cost is so high, we made one improvement to the traditional and full vectorizers: an operation is not vectorized unless it has at least one vectorizable predecessor or successor. Doing otherwise is clearly unfavorable. With selective vectorization, such scenarios are detected automatically.

Benchmark	Traditional	Full	Selective
093.nasa7	0.18	0.76	1.04
101.tomcatv	0.71	0.99	1.38
103.su2cor	0.63	0.94	1.15
104.hydro2d	0.94	1.00	1.03
125.turb3d	0.38	0.93	0.95
146.wave5	0.76	0.96	1.03
171.swim	1.01	1.00	1.17
172.mgrid	0.53	0.99	1.26
301.apsi	0.51	0.97	1.02

Table 2. Speedup compared to modulo scheduling.

Our target processor only supports aligned vector memory operations. Therefore, the compiler inserts the proper instructions to access misaligned data. We do not employ any techniques that provide alignment information, meaning that all vector memory operations are assumed to be misaligned. However, most of the overhead is eliminated using data from the previous iteration [13, 40].

Table 2 shows the speedup of each vectorization technique over modulo scheduling. Comparing traditional and full vectorization, we see that performance degradation from loop distribution is considerable. Much of this is due to the fact that our simulated architecture does not support an efficient scatter/gather mechanism. When memory operations are not vectorizable, they are first aggregated into contiguous memory so they can be used directly in vectorized computation.

In most cases, full vectorization (with modulo scheduling) matches the performance of modulo scheduling alone. An exception is `nasa7` where vectorization significantly underperforms modulo scheduling. In all cases but one, selective vectorization yields the best performance. For some benchmarks, the improvement is substantial. In the case of `tomcatv`, selective vectorization achieves a 1.38x speedup over modulo scheduling alone. For some benchmarks performance improvements are small, and in the case of `turb3d`, selective vectorization perfoms worse than baseline modulo scheduling. In fact, our algorithm does create more compact schedules for the critical loops in this benchmark. However, tighter schedules tend to increase the number of stages in a software pipeline, leading to longer prologues and epilogues. Since the critical loops in `turb3d` have low iteration counts, the prologue and epilogue contribute significantly to the execution, thereby negating the advantage of vectorization.

Benchmark	Number of Loops	ResMII			II		
		Better	Equal	Worse	Better	Equal	Worse
093.nasa7	30	9 (30.0%)	21 (70.0%)	0 (0.0%)	8 (26.7%)	21 (70.0%)	1 (3.3%)
101.tomcatv	6	5 (83.3%)	1 (16.7%)	0 (0.0%)	5 (83.3%)	1 (16.7%)	0 (0.0%)
103.su2cor	38	27 (71.1%)	11 (28.9%)	0 (0.0%)	27 (71.1%)	11 (28.9%)	0 (0.0%)
104.hydro2d	67	23 (34.3%)	44 (65.7%)	0 (0.0%)	23 (34.3%)	44 (65.7%)	0 (0.0%)
125.turb3d	12	4 (33.3%)	8 (66.7%)	0 (0.0%)	4 (33.3%)	7 (58.3%)	1 (8.3%)
146.wave5	133	57 (42.9%)	76 (57.1%)	0 (0.0%)	51 (38.3%)	73 (54.9%)	9 (6.8%)
171.swim	14	5 (35.7%)	9 (64.3%)	0 (0.0%)	5 (35.7%)	9 (64.3%)	0 (0.0%)
172.mgrid	16	9 (56.2%)	7 (43.8%)	0 (0.0%)	9 (56.2%)	7 (43.8%)	0 (0.0%)
301.apsi	61	18 (29.5%)	42 (68.9%)	1 (1.6%)	17 (27.9%)	39 (63.9%)	5 (8.2%)

Table 3. Number of loops for which selective vectorization finds an II better, equal to, or worse than competing techniques.

4.2. Opportunities for Selective Vectorization

Table 3 examines the degree to which selective vectorization is performed. For each benchmark, we show 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). Since no technique can improve the II when it is constrained by recurrences, we only report loops that are resource-limited. Table 3 separates results into the resource-constrained II (ResMII) as computed by the modulo scheduler and the final II. As shown, there are a significant number of loops for which selective vectorization provides an advantage. Furthermore, there is only one loop across all benchmarks for which selective vectorization produces a loop with higher resource requirements. In the last column, we see that selective vectorization leads to a small number of loops with an inferior II. This is due to the fact that iterative modulo scheduling is a heuristic. Even when we lower resource requirements, the algorithm is not guaranteed to achieve a lower II in all cases.

Benchmark	Considered	Ignored
093.nasa7	1.04	0.78
101.tomcatv	1.38	1.22
103.su2cor	1.15	1.02
104.hydro2d	1.03	0.98
125.turb3d	0.95	0.81
146.wave5	1.03	0.99
171.swim	1.17	1.08
172.mgrid	1.26	1.14
301.apsi	1.02	0.97

Table 4. Speedup of selective vectorization compared to modulo scheduling when communication overhead is considered vs. ignored.

4.3. Communication

In Table 4 we demonstrate the importance of communication considerations during selective vectorization. The second column shows the speedup compared to modulo scheduling when explicit transfer operations are taken into account during cost analysis (replicated from Table 2). Recall that our simulated architecture requires a series of load and store instructions to communicate operands between vector and scalar resources. In the third column we show the performance when communication overhead is ignored during partitioning. Note that for correct operation, these instructions are still inserted prior to modulo scheduling. When communication is neglected, most benchmarks experience a severe performance degradation. It is clear that a viable solution must track communication costs carefully if selective vectorization is to be successful.

4.4. Alignment

Finally, Table 5 shows what is possible when memory alignment information is available. In the second column, we show the speedup of selective vectorization compared to modulo scheduling. These numbers reproduce those shown in Table 2, where the vector memory operations are assumed to be misaligned. The third column shows speedup when the overhead of merging misaligned regions is disregarded. Alignment overhead can be eliminated when operations are known at compile-time to be aligned. The results in the last column of Table 5 represent a best-case scenario since every reference is assumed to be aligned.

Static alignment information can be gathered using a number of techniques. Our goal is not to advocate a particular approach. Rather, we point out that when static alignment information is available, it is readily incorporated into our system. Simply, explicit realignment operations are considered during cost analysis. For many benchmarks, improvements are modest as software pipelining is able to hide

Benchmark	Misaligned	Aligned
093.nasa7	1.04	1.07
101.tomcatv	1.38	1.48
103.su2cor	1.15	1.16
104.hydro2d	1.03	1.05
125.turb3d	0.95	0.95
146.wave5	1.03	1.04
171.swim	1.17	1.21
172.mgrid	1.26	1.26
301.apsi	1.02	1.02

Table 5. Speedup of selective vectorization compared to modulo scheduling when vector memory operations are assumed to be misaligned vs. aligned.

the extra latency associated with explicit realignment. However, there are a few benchmarks (most notably `tomcatv`) that benefit from the reduced resource contention.

5. Related Work

With the advent of short vector extensions, there is a need to supply access to these instructions to the high-level programmer. Early support came in the form of inline assembly, macro calls, and specialized library routines. Some proposals for automatic parallelization advocate a basic block approach [20, 34, 33, 18]. Most approaches follow classic techniques [6, 39] developed for vector supercomputers [7, 10, 13, 25, 26, 35, 41]. Commercial products targeting multimedia extensions include the Intel compiler [7], the IBM XL compiler [2], the VAST/AltiVec compiler [4], and VectorC [1].

Whether parallelism is identified within a basic block or across loop iterations, existing techniques focus on extracting as much DLP as possible. They do not attempt the partial vectorization we advocate in this paper. We expect that commercial vectorizers perform sophisticated cost analyses to determine when vectorization is profitable and when it should be excluded. We argue that cost analysis is more accurate in the compiler backend where vectorization decisions can be evaluated in terms of actual architectural resources.

One of the primary difficulties facing automatic vectorization for short vector extensions is the alignment restriction placed on vector memory operations. Our approach readily copes with the alignment restrictions imposed by many architectures. Some processors (e.g., AltiVec) require that vector memory operations address locations that are aligned on natural boundaries. As a result, the compiler explicitly reorganizes the data using additional instructions. Other processors (e.g., Pentium) support unaligned memory operations, but incur a performance penalty if the data span multiple cache lines. Many of the techniques for satisfying alignment constraints [7, 13, 17, 21, 28, 40, 43] can be directly applied in our algorithm.

The software pipeliner in Trimaran is an implementation of Rau's iterative modulo scheduling [30, 31]. Similar methods were developed by Lam [19], who pioneered modulo variable expansion. Important extensions to the core modulo scheduling algorithm include techniques to reduce register pressure [12, 15] and the ability to schedule loops with control flow [22]. To our knowledge, we are the first to advocate vectorization as a method to improve resource utilization in modulo scheduled loops.

Recently there has been interest in modulo scheduling for clustered architectures [5, 8, 27, 42]. Explicit communication among clusters is similar to communication between vector and scalar operations. In both cases, partitioning can introduce transfer operations not present in the original loop. Our approach to vectorization differs in two ways. First, operands transfered between vector and scalar units are typically done through memory, requiring load and store operations that compete for resources with existing memory operations. In contrast, a clustered architecture usually employs an operand network with dedicated resources devoted to communication between clusters. Second, our algorithm performs instruction selection while also tracking changes in resource requirements. A vector opcode may have completely different resource requirements than its corresponding scalar opcode. These requirements, and their competition with other instructions, are monitored closely in order to accurately gauge the trade-offs of vectorization.

Our selective vectorization methodology implements the partitioning heuristic developed by Kernighan and Lin [16]. We believe the algorithm provides an intuitive match for our problem. It is possible that other partitioning heuristics are also suitable. However, since our problem involves exactly two partitions, more general partitioning heuristics may be less applicable. Regardless of the algorithm used, we argue that resource utilization must be tracked carefully when considering partitioning alternatives. Software pipelining facilitates this by allowing us to neglect operation latency during vectorization.

6. Future Work

The algorithm presented in this paper conceptually unrolls loops by a factor equal to the vector length. However, it may be feasible to consider larger scheduling windows. For example, assuming a vector length of two, we could vectorize across iterations $3i$ and $3i+1$ and leave the operations in iterations $3i+2$ in scalar form. Here, we assign whole iterations to one set of resources. In the absence of loop-carried dependences, this approach requires no communication between scalar and vector operations. The drawback is that alignment is adversely affected since aligned memory operations are only possible when the unroll factor is a multiple of the vector length.

Another potential improvement relates to register pressure. Most contemporary multimedia designs separate the vector and scalar register files. For such architectures, selective vectorization can reduce spilling by using both sets of registers. More effective partitioning might be possible if the algorithm can determine which decisions lead to fewer spills.

Currently, our infrastructure only targets well-behaved `do` loops. Further performance improvements are possible if the scope is extended to handle `while` loops and loops with early exits. Techniques for modulo scheduling such loops already exist [32]. Much of the complexity lies in ensuring the software pipeline is drained properly upon exit. The task is more difficult in the presence of vector operations since some elements of a vector operation should not execute. Draining a software pipeline with vector instructions could be accomplished if the architecture supports vector masks. In the absence of special hardware support, it might be possible to execute vector instructions normally, as long as we ensure that vector stores only write intended memory locations.

Finally, this work would readily benefit from any loop transformations that expose data parallelism, in particular loop interchange and reduction recognition [6]. The former can create unit-stride memory references in the inner loop, and the latter allows for the vectorization of reductions.

7. Conclusion

Short vector extensions have been integrated into the ISA of many general-purpose and embedded microprocessors, adding a data parallel component to ILP designs. In this paper, we propose a new methodology for exploiting vector parallelism in software pipelined loops. Our approach judiciously vectorizes operations in important program loops to improve overall resource utilization, allowing for software pipelines with shorter initiation intervals. To our knowledge, this is the first paper to show that a union of ILP and DLP can lead to improved performance. Selective vectorization is applied in the backend, where the impact of vectorization is measured with respect to actual machine resources. This allows the algorithm to accurately account for any communication of operands between scalar and vector operations. Finally, our approach provides a natural mechanism for managing the alignment restrictions enforced by modern multimedia architectures.

We evaluate our methodology using nine SPEC FP benchmarks. Compared to software pipelining, our selective vectorization approach achieves a maximum speedup of 1.38×, with an average of 1.11×.

Acknowledgements

We gratefully acknowledge Charles Leiserson for bringing the work of Kernighan and Lin to our attention. We acknowledge Krste Asanović, Alexandre Eichenberger, William Thies, and Peng Wu for many helpful discussions. We also thank the anonymous referees for their valuable suggestions on an earlier version of this paper. This research was supported in part by NSF award EIA-0071841, and DARPA contracts PCA-F29601-03-2-0065 and HPCA/PERCS-W0133890.

References

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

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

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

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

[5] A. Aletà, J. M. Codina, J. Sánchez, and A. 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.

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

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

[8] J. M. Codina, J. Sánchez, and A. 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.

[9] A. 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.

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

[11] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, March 2000.

[12] A. E. Eichenberger and E. 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.

[13] A. E. Eichenberger, P. Wu, and K. 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.

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

[15] R. 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.

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

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

[18] A. Kudriavtsev and P. 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.

[19] M. 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.

[20] S. Larsen and S. 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.

[21] S. Larsen, E. Witchel, and S. 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] D. M. Lavery and W. 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] R. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

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

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

[26] D. Naishlos, M. Biberstein, S. Ben-David, and A. 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.

[27] E. Nystrom and A. 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.

[28] I. Pryanishnikov, A. Krall, and N. 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.

[29] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, 20(4):47–57, July 2000.

[30] B. Rau, M. Lee, P. Tirumalai, and M. 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.

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

[32] B. R. Rau, M. S. Schlansker, and 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.

[33] J. Shin, , M. Hall, and J. 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.

[34] J. Shin, J. Chame, and M. 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.

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

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

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

[38] 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.

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

[40] P. Wu, A. E. Eichenberger, and A. 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.

[41] P. Wu, A. E. Eichenberger, A. Wang, and P. 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.

[42] J. Zalamea, J. Llosa, E. Ayguadé, and M. 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.

[43] Y. Zhao and K. Kennedy. Scalarization on Short Vector Machines. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 187–196, Austin, TX, March 2005.