Exploiting Vector Parallelism in Software Pipelined Loops

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

Abstract

An emerging trend in processor design is the incorporation of short vector instructions into the ISA. In fact, vector extensions have appeared in most general-purpose microprocessors. To utilize these instructions, traditional vectorization technology can be used to identify and exploit data parallelism. In contrast, efficient use of a processor's scalar resources is typically achieved through ILP techniques such as software pipelining. In order to attain the best performance, it is necessary to utilize both sets of resources. This paper presents a novel approach for exploiting vector parallelism in a software pipelined loop. At its core is a method for judiciously partitioning operations between vector and scalar resources. The proposed algorithm (*i*) lowers the burden on the scalar resources by offloading computation to the vector functional units, and (*ii*) partially (or fully) inhibits the optimizations when full vectorization will decrease performance. This results in better resource usage and allows for software pipelining with shorter initiation intervals. Although our techniques complement statically scheduled machines most naturally, we believe they are applicable to any architecture that tightly integrates support for ILP and data parallelism.

An important aspect of the proposed methodology is its ability to manage explicit communication of operands between vector and scalar instructions. Our methodology also allows for a natural handling of misaligned vector memory operations. For architectures that provide hardware support for misaligned references, software pipelining effectively hides the latency of these potentially expensive instructions. When explicit alignment is required in software, our algorithm accounts for these extra costs and vectorizes only when it is profitable. Finally, our heuristic can take advantage of alignment information where it is available.

We evaluate our methodology using several DSP and SPEC FP benchmarks. Compared to software pipelining, our approach is able to achieve an average speedup of 1.30× and 1.18× for the two benchmark sets, respectively.

Keywords: *Modulo Scheduling, Vectorization, ILP*

1 Introduction

Increasingly, modern general-purpose and embedded processors provide short vector instructions that operate on elements of packed data [9, 12, 21, 22, 24, 29]. Vector instructions are desirable because the vector functional units can perform the same operation on multiple operands in parallel. Thus, a vector instruction increases the amount of concurrent execution while maintaining a compact instruction encoding. In addition, their performance advantages are 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 in nature from fully parallel to fully sequential.

In the context of applications rich with data parallelism, compiler technology previously pioneered for vector supercomputers is not ideal for today's ILP-based processors. When loops contain a mix of vectorizable and non-vectorizable operations, a traditional approach may diminish ILP as it leads to separate loops for the vector and scalar operations. In the *vectorized* loops, scalar resources are not used well, 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. However, even dynamically scheduled processors can not parallelize instructions across distinct loops.

As an alternative to vectorizing technology, a loop-intensive program exhibiting data parallelism can be software pipelined, essentially converting available parallelism to ILP. Software pipelining overlaps instructions from different loop iterations and derives a schedule that attempts to maximize resource utilization. Unfortunately, without explicit instruction selection that vectorizes operations, the architecture's vector resources are not used and software pipelining is limited.

Today, the burden of programming short vector units falls largely on the application developer who manually orchestrates the mapping of operations to resources (i.e., which scalar operations to vectorize). Invariably, this style of hand-programming is tedious, error prone, and results in non-portable solutions. In this paper, we show that concurrently using 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. The intuition underlying this work is that better utilization of both scalar and vector resources will lead to greater overlap among iterations, and hence improve performance. Our algorithm effectively maps loops with data parallelism to processors with short vector instructions. It selectively chooses to vectorize the most profitable data parallel computations, while remaining cognizant of the resource requirements of the loop. As a result, the algorithm (*i*) lowers the burden on the scalar resources by offloading computation to the vector functional units, and (*ii*) partially (or fully) inhibits the optimizations when full vectorization will decrease performance.

A noteworthy aspect of our approach is that it 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 must explicitly reorganize the data using additional instructions [23]. Other processors (e.g., Itanium) support unaligned memory operations, but incur a performance penalty if the data span multiple cache lines [15]. Many of the techniques for satisfying alignment constraints [6, 11, 19] can be directly applied in our algorithm. Instruction selection considers any misalignment penalties and instruction overheads when choosing to vectorize an operation. Furthermore, because our work leverages the concept of software pipelining, we can easily hide the added latency of misaligned vector operations.

We implemented the proposed optimization in Trimaran [2], 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. We found that 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 [25], our optimization yields a 1.30× and 1.18× speedup on a set of DSP and SPEC FP benchmarks, respectively.

This paper is organized as follows: Section 2 describes the intended architectural model for this work. Section 3 motivates our approach with a simple example. Section 4 presents our algorithm for automatic instruction selection and selective vectorization. Section 5 contains our experimental evaluation. Section 6 describes related work and Section 7 concludes the paper.

2 Target Architecture

We believe our techniques are applicable to any machines that provide a combination of ILP and short vector hardware. In this paper, however, we focus our attention on VLIW architectures since they are the primary benefactors of static scheduling techniques. Several innovations have been developed to support ILP scheduling in general, and software pipelining in particular. Primarily, these consist of rotating registers and predication [8]. Predication is useful for eliminating code expansion by reusing the kernel code for the epilogue and prologue. It also enables the scheduling of loops with control flow [20]. Rotating registers eliminate scalar anti and output dependences by queuing register writes until they are used. In our evaluation, we assume these concepts are extended to vector registers as well so that vector operations can be software pipelined using the same technique. If rotating registers are not available, a similar effect can be achieved with *modulo variable expansion* [18, 26]. This method unrolls the loop and performs scalar renaming to remove false dependences.

Multimedia architectures typically require memory operations for communicating data between scalar and vector registers. In our evaluation we also assume this interface, even though our approach does not rely on it. Since vector instructions are never involved in a dependence cycle, software pipelining can hide the extra latency of explicit communication. However, these operations still compete for machine resources and can limit performance when these resources are in high demand. Since our system tightly integrates scalar and vector instructions, it could benefit from a faster communication mechanism. An ideal design would allow scalar operations the ability to read and write vector elements directly. This would eliminate the need for explicit communication instructions. Such a design could be achieved by superimposing the vector registers on the scalar registers.

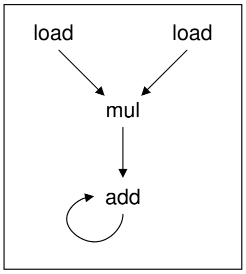
3 Motivating Example

We use the dot product shown in Figure 3(a) to motivate our approach. The data dependence graph is shown in part (b). For simplicity, we omit address calculations. Suppose we target a machine whose visible machine resources consist of three issue slots. Assuming single-cycle latency for all operations, a modulo schedule for the loop is shown in part (c). Here, we achieve an *initiation interval* (II) of two. In a modulo schedule, the II measures the throughput of the kernel. It is limited by the available resources and any cycles in the dependence graph [25]. The latter restrict throughput because a new iteration can not begin until its dependences from the previous iteration are satisfied. In the schedule of part (c), we require two cycles to execute the four loop operations.

Now suppose the target architecture supports execution of one vector instruction each cycle (including memory operations). Assume the machine implements a vector length of two for the operations in the loop. A dependence cycle prevents vectorization of the `add` operation. As a result, the traditional approach distributes the loop into one that performs the vector computation and another that performs the scalar computation. *Scalar expansion* is used to communicate intermediate values through memory. This is shown in part (d). Since the machine can issue only

```
DO I=1, N
  S = S + X(I) * Y(I)
ENDO
```

(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)

```
DO I=1, N, 2
  S$(I:I+1) = X(I:I+1) * Y(I:I+1)
ENDO

DO I=1, N
  S = S + S$(I)
ENDO
```

(d)

Figure 1: (a) A dot product kernel and (b) its data dependence graph. (c) Modulo scheduling the kernel results in an II of two. Subscripts specify the original iterations executed by each operation; the kernel is highlighted in gray. (d) Loop distribution of the kernel into vector and scalar loops.

one vector operation each cycle, software pipelining does not improve performance of the vector loop. Four cycles are required to execute the four vector operations. For a vector length of two, this amounts to an initiation interval $\mathrm{II}_v = 2$. The operations in the scalar loop can be overlapped, resulting in an initiation interval $\mathrm{II}_s = 1$. Combined, this achieves an overall initiation interval of $\mathrm{II}_v + \mathrm{II}_s = 3$, resulting in worse performance than modulo scheduling alone.

A better approach is to leave the loop intact so the vector and scalar operations can execute concurrently. This strategy is illustrated in Figure 2(a). Here we achieve an II of 1.5, since two iterations are initiated every three cycles. For simplicity, we have assumed explicit operations are not required for communicating between scalar and vector operations (namely, the vector multiply and the scalar add).

Finally, Figure 2(b) illustrates what is possible with selective vectorization. By inhibiting vectorization of one of the load operations, we achieve better overall resource utilization. This allows a schedule with an initiation interval of one, resulting in the best performance.

Cycle	Slot 1	Slot 2	Slot 3
1	$vload_{1\text{-}2}$		
2	$vload_{1\text{-}2}$		
3	$vmul_{1\text{-}2}$		
4	$vload_{3\text{-}4}$	add_1	
5	$vload_{3\text{-}4}$	add_2	
6	$vmul_{3\text{-}4}$		
…	…	…	…

(a)

Cycle	Slot 1	Slot 2	Slot 3
1	$vload_{1\text{-}2}$	$load_1$	
2		$load_2$	
3	$vload_{3\text{-}4}$	$load_3$	
4	$vmul_{1\text{-}2}$	$load_4$	
5	$vload_{5\text{-}6}$	$load_5$	add_1
6	$vmul_{3\text{-}4}$	$load_6$	add_2
…	…	…	…

(b)

Figure 2: (a) Modulo schedule obtained with full vectorization when the loop is left intact. This results in an II of 1.5. (b) Using selective vectorization, we can achieve an II of 1.

4 Instruction Selection and Vectorization

Before we describe our instruction selection and vectorization algorithm, we emphasize that software pipelining in the form of modulo scheduling [25] follows our optimization. This fact has strong implications for our algorithm design. 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 without affecting the throughput of the kernel. 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 loop iteration count is relatively high. Vectorizable operations are never involved in a recurrence. Otherwise, they could not be parallelized in the first place. This allows us to focus on resource usage alone during vectorization.

At a high level, the goal of instruction selection and vectorization is to divide the vectorizable operations between scalar and vector resources. This partitioning is done in a way that maximizes loop performance. In this phase we are concerned only with the decision of whether or not to vectorize each operation. Software pipelining and register allocation are performed later.

The conventional strategy vectorizes all data parallel operations. In loops with a large number of vector operations, this can leave scalar resources idle. Moving some operations to scalar units can provide a more compact schedule. In other situations, full vectorization may be more appropriate. This can occur when machine resources are overwhelmed by excessive transfer between vector and scalar register files. For the same reason, it may be advantageous to omit vectorization altogether in loops with little data parallelism. The best choice depends on the underlying architectural resources, the number and type of operations in the loop, and the dependences among them.

The problem is further complicated when the compiler is responsible for satisfying complex scheduling requirements. Most architectures provide heterogeneous functional units, each supporting a subset of the machine's instruction set. A particular operation may have many scheduling alternatives and could reserve multiple resources during its execution. Furthermore, scalar and vector operations may compete for the same resource. This is almost certainly the case for memory operations since the same functional units typically execute vector and scalar memory operations.

Below we provide our compilation methodology for exploiting data parallelism in software pipelinable loops. We describe our approach to identifying vectorization opportunities, followed by a description of our heuristic for selective vectorization. We conclude the section by addressing the communication and alignment constraints imposed by existing architectures.

4.1 Identifying Vectorizable Operations

Before we can proceed with operation partitioning, the first step is to identify all vectorizable operations in the loop body. To accomplish this, we use the data dependence theory first developed for vector supercomputers. This requires loop dependence analysis to identify cycles in the dependence graph. Operations in a dependence cycle must execute sequentially; the rest can be vectorized. The major difficulty here is to accurately identify dependences among memory references. A simple approach that assumes dependence between any store and load will prevent vectorization. For array-based code, an extensive literature exists for computing dependences among array accesses (see [5] for a review). In our toolchain, we use the implementation provided by SUIF [30]. After building the dependence graph, cycles are identified using Tarjan's algorithm for strongly connected components [28].

Traditionally, parallel operations are identified at the source level. Since we are partitioning actual machine instructions, data parallelism must be identified on a low-level intermediate format.

```
Partition-Ops ()
    for i ← 1 to numOps
        currPartition[i] ← SCALAR
    bestPartition ← currPartition
    bestCost ← BIN-PACK(currPartition)
    lastCost ← ∞
    while lastCost ≠ bestCost
        lastCost ← bestCost
        locked ← ∅
        for i ← 1 to numVectorizable
            cost ← SWITCH-ONE-OP (currPartition, locked)
            if cost < bestCost
                bestCost ← cost
                bestPartition ← currPartition
        currPartition ← bestPartition
    return bestPartition
```

```
Switch-One-Op (currPartition, locked)
    bestCost ← ∞
    for i ← 1 to numOps
        if VECTORIZABLE (i) ∧ i ∉ locked
            cost ← TEST-REPARTITION (i, currPartition)
            if cost < bestCost
                bestCost ← cost
                best ← i
    locked ← locked ∪ best
    currPartition[best] ← ¬currPartition[best]
    return BIN-PACK (currPartition)
```

Figure 3: Partitioner pseudo code.

Therefore, memory dependence information is computed in the front-end and passed as annotations to the backend. A low-level representation introduces dependences not present in the original source. In three-operand format, temporary registers are introduced to hold the results of subexpressions. The flow dependence involving these registers also creates an anti-dependence with the next iteration. This results in a dependence cycle that prevents vectorization. However, virtual registers whose uses are only reached by definitions in the same iteration can be *privatized* to remove the false dependence. Thus, while vectorization is traditionally performed using a source-level program representation, we show that identifying data parallelism in low-level codes presents no real problems.

4.2 Partitioning Heuristic

Be base our algorithm on the two-cluster partitioning heuristic due to Kernighan and Lin [16]. The pseudo code is shown in Figure 3. The algorithms iteratres over the vectorizable operations in the loop, moving them between a scalar and vector partition until the lowest cost partition is found. In the end, operations in the vector partition are vectorized and the rest remain scalar. Initially, all operations are assigned to the scalar partition, although a completely vector starting condition may be equally viable. At each step, an operation is chosen to move from one partition to the other. Once an operation is moved, it is *locked* and removed from consideration until the next iteration of the partitioner. This causes every operation to move exactly once per iteration. After each move, the algorithm computes the cost of the resulting configuration, noting the minimum cost encountered so far. When every operation has moved, the partition with the lowest overall cost is used as the new starting configuration for another iteration. The process terminates when we see no improvement over the starting configuration.

4.3 Operation Selection and Cost Calculation

Operations are selected for repartitioning based on a cost function. At each step, we choose the operation whose movement results in the configuration with the lowest overall cost. We define the cost of a given configuration to be the *weight* of the most heavily used resource, where the weight is the number of cycles the resource is needed for executing the operations in the configuration. In modulo scheduling terminology, this corresponds to the resource-constrained minimum initiation

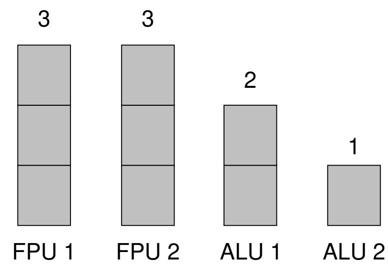


Figure 4: Bin-packing example.

interval (ResMII). In the absence of dependence cycles, this represents a lower bound on the II of the modulo schedule.

We calculate the cost of a configuration using a *bin-packing* approach akin to the original formulation in modulo scheduling. 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. Every placement of an operation in a bin increments the weight of that bin. If an operation reserves a resource for more than one cycle, the bin's weight is adjusted accordingly. In addition, each scalar operation is binned k times to match the work output of a single k-wide vector operation.

Since communication of operands between scalar and vector operations usually requires explicit instructions, the partitioning algorithm must account for these instructions as a result of its decisions. Specifically, when an operation is placed in a different partition than operations on which is it flow-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.

In the original bin-packing formulation, operations are selected for placement in order of their scheduling alternatives such that those with little freedom are binned first. This heuristic produces better results than an unordered placement. 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 each bin weight. This strategy tends to balance operations across bins even when these operations do not affect the cost of the overall configuration. This is illustrated in Figure 4. During bin-packing, assume we have reached a state with three operations placed in each FPU, two operations in ALU 1, and one in ALU 2. If the next operation can be placed on either ALU, a simple implementation may choose ALU 1 since the operation's addition to that bin does not increase the total cost. The sum squares of such a configuration is $3^2 + 3^2 + 3^2 + 1^2 = 28$. The alternative configuration has a lower value of $3^2 + 3^2 + 2^2 + 2^2 = 24$ and provides a balance among the ALUs.

This optimization allows the partitioner to quickly compare alternatives during operation selection. Ideally, we would like to perform a complete bin-packing for each possibility before selecting an operation for repartitioning. Unfortunately, this 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, deallocates the resources for the operation under consideration, and reserves the set of resources used in the other partition. For example, if we move an operation to the vector partition, we simply remove that operation's scalar resources from the bins and reserve its vector resources. If necessary, we also account for any transfer costs implied by the repartitioning. We then record the cost of the new configuration, restore the bins to their original state, and repeat the process for another operation. Once an operation is finally selected and repartitioned, we perform a fresh bin-packing to rebalance the bins.

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 an operation's producers and consumers remain in the other partition, necessitating explicit transfer instructions (whose resource requirements must be considered). However, the algorithm will continue to move one instruction at each step. If there are no alternatives that reduce the overall cost, the algorithm chooses the configuration that yields the lowest increase in cost. It is this strategy that allows the algorithm to climb out of a local minimum. As communicating operations are repartitioned, the communication cost decreases, ideally toward a new minimum.

The algorithm continues to iterate until no additional improvements are possible. In the worst case, this requires n total iterations for a loop containing n vectorizable operations. Every iteration repartitions each operation once and bin-packs each new configuration. Since bin-packing itself requires n steps, this results in an $O(n^3)$ algorithm. However, 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 in the instruction selection and vectorization phase is far less than the time spent modulo scheduling. Nonetheless, if a faster execution of the algorithm is desirable, we can artificially limit the number of iterations it carries out.

On a final note, we also compared the results from our algorithm to those generated from an exhaustive search approach that is practical for loops consisting of 15-20 instructions. For several randomly generated kernels, the heuristic produced identical results in all cases.

4.4 Loop Vectorization

After partitioning decisions are made, the next step is to construct a new loop with the vectorized instructions. In order to satisfy dependences, operations are emitted in a specific order. We start with the original dependence graph and combine each strongly connected component into a single node, called a *piblock* [5]. Data parallel operations comprise their own piblock. Dependences between operations in different piblocks are reconnected to the encompassing piblocks. This creates a new graph without dependence cycles. A topological sort of the graph then determines the order of the piblocks with respect to each other. This is analogous to the loop distribution phase in traditional vectorization.

For each operation selected for vectorization, we emit the vector equivalent opcode. Each scalar operation is emitted k times (where k is the vector length). If a scalar operation is contained in a dependence cycle, the operations in the cycle are emitted in their original control flow sequence in order to satisfy dependences. This effectively unrolls the piblock k times.

When explicit communication is required, the appropriate transfer operations are inserted after the operands are computed. The compiler also adjusts the loop increment and upper bound according to the vector length k. Lastly, if the loop trip count is not known statically, or if it is not a multiple of the vector length, a *cleanup* loop is inserted to execute the remaining iterations.

4.5 Alignment

An important consideration for contemporary short-vector extensions is the alignment constraint placed on vector memory operations. Specifically, a k-way load or store must operate on data that fall on a k-byte boundary. Some processors support unaligned memory operations, but incur a performance penalty if the data cross a cache line. Others permit no misalignment and rely on software to merge data from consecutive misaligned regions.

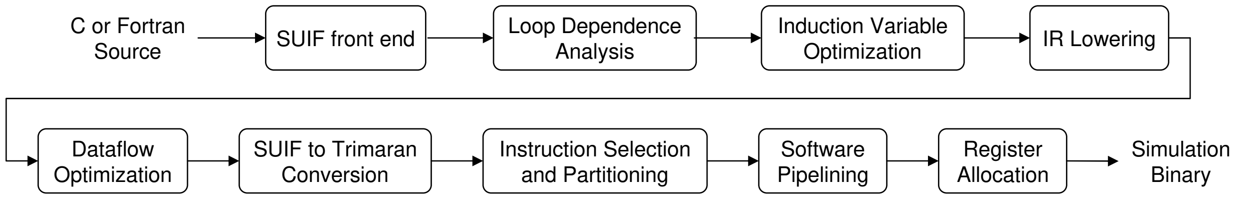


Figure 5: Compiler flow.

When dynamic alignment is supported in hardware, the extra latency incurred by misaligned loads is not a major concern in our approach. Vector operations never lie on dependence cycles. Therefore, the latency of a misaligned vector load can be tolerated by the software pipeliner. In the absence of alignment information, we assume a misaligned latency for vector loads during software pipelining.

If explicit instructions are required for misaligned memory operations, our algorithm handles the extra cost naturally. When considering a vector versus scalar allocation, the necessary instructions are bin-packed during cost evaluation. This way, we accurately gauge the trade-offs of vectorizing unaligned memory operations.

Solutions to the alignment problem have been proposed by us and other researchers [6, 11, 19, 32]. Current techniques attempt to both extract alignment information from the program, and transform loops to exhibit more aligned references. The transformation usually involves peeling the first few iterations from the loop until the memory references within the loop reach a certain aligned configuration. At this point, a vectorized version of the loop can proceed assuming the alignments reached in the pre-loop. These methods can be incorporated directly into our proposed optimization. Whenever specific alignment information is available, we model the cost of each vector memory operation appropriately.

5 Evaluation

Figure 5 shows the high-level compiler flow of our system. We use SUIF [30] as our compiler front-end. As mentioned, identification of vector parallelism requires accurate loop dependence information. We leverage SUIF's dependence analysis package to accomplish this. Our SUIF front-end also allows us to compile Fortran and provides us with a suite of existing optimizations. Our backend compiler and simulation system is provided by Trimaran [2]. In addition to writing a SUIF to Trimaran converter, several changes were required in the Trimaran infrastructure. Most notably, we have added support for vector instructions and vector registers throughout the system.

Extracting precise dependence information from C is notoriously difficult since the language allows arbitrary aliasing through pointers. This problem is usually solved by applying a whole-program alias analysis or by extending C with directives that allow the user to convey dependence information. For the C benchmarks we study, we assume pointer arguments passed to a function do not overlap. This is analogous to using the `restrict` keyword in function declarations and allows us to detect data parallelism when arrays are passed as arguments.

For all loops, we apply a suite of standard optimizations before scheduling. These include register promotion, common subexpression elimination, copy propagation, constant propagation, dead code elimination, induction variable optimization, and loop-invariant code motion. We did not employ any transformations specifically targeted to enhance data parallelism. Of these, loop interchange [5] would be particularly useful since it can move unit-stride memory references into

Issue width	6
Integer units	6
Load/store units	2
Floating-point units	2
Vector units	1
Vector length (32-bit elements)	4
Vector length (64-bit elements)	2

Table 1: Processor configuration.

the inner loop. Transformations that increase opportunities for vectorization will only improve performance since our system is not obligated to vectorize. In any case, the focus of this paper is not the identification of data parallelism. We are interested in how it can be exploited to create highly efficient loop schedules.

We apply our transformation 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 [4] and hyperblock formation [20] have made it possible to target a broader class of loops, but they are not used in this study. In the future, we plan to incorporate some of these techniques to broaden the classes of loops we vectorize.

We evaluate performance on the target machine shown in Table 1. We assume a total vector length of 128 bits. Therefore, the ISA supports vectors of four 32-bit elements or two 64-bit elements. This is consistent with contemporary machines [9, 24]. Multimedia architectures also support longer vectors of subword operations, but none of the benchmarks we study here use subword data types. We assume architectural support for misaligned memory references. However, the simulator charges double latency for any load operations that cross a cache line boundary. Essentially, we assume two sequential loads are required to retrieve these data.

In the results that follow, we show speedup of our technique over modulo scheduling. For this baseline, we unrolled each loop k times (for vector length k) prior to scheduling. This eliminates the advantage gained from reduced address arithmetic when loops are vectorized. When vectorization is enabled, an extra performance gain is possible because a single address calculation is needed for each vector memory operation. This same optimization can be realized without vectorization by taking advantage of the *base + offset* addressing mode available in most ISAs, including our target machine.

In Figure 6 we show the speedup for five DSP kernels: `median filter`, `Daub4` wavelet and inverse wavelet transforms, `FFT` (Fast Fourier Transform), and the `Haar` wavelet transform. The graph shows speedup for four different machine configurations. The baseline architecture is summarized in Table 1. In Figure 6 we also vary the number of FPUs and memory units between two and four. We focus on these resources since the bulk of computation in these kernels is memory operations and floating point calculations.

Each bar shows the speedup achieved from *full vectorization* and the additional performance gained with *selective vectorization* as guided by our algorithm. In both cases, the loop is left intact in order to overlap scalar instructions with vector instructions. When full vectorization is enabled, we simply vectorize all data parallel operations that are detected in the loop. Selective vectorization uses the partitioning algorithm described in Section 4. Notice that in many cases full vectorization performs worse than modulo scheduling. This occurs when the number of scalar resources surpasses the number of parallel operations that can be computed with the vector unit. In the case of `Haar`, it also occurs when the costs of transmitting the results of vectorized operations

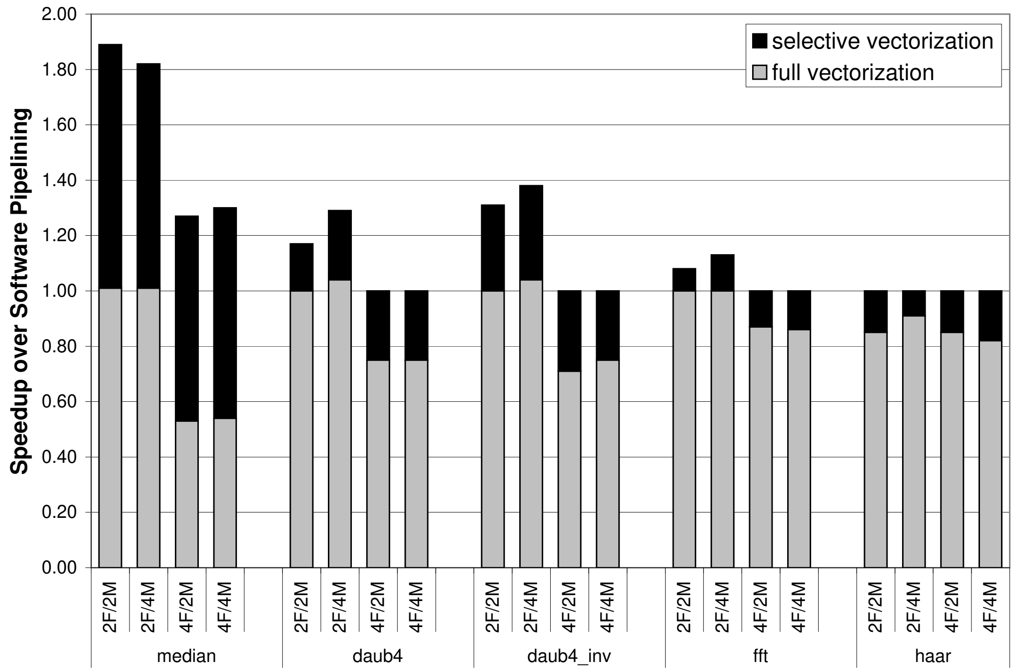


Figure 6: Speedup over modulo scheduling for a set of DSP kernels. For each kernel, four different machine configurations are shown with the number of FPUs (F) and memory units (M) varying between 2 and 4.

becomes the performance bottleneck.

In all cases, our heuristic is able to produce a schedule that meets or exceeds those provided by full vectorization or modulo scheduling alone. For many combinations, offloading vector computation to the scalar units results in a significant speedup. Our algorithm also automatically detects situations when baseline modulo scheduling provides the best performance. In this case, it throttles back vectorization or completely inhibits it.

In Table 2, we also show the speedup for several SPEC FP benchmarks. For these, we use the machine configuration of Table 1. These benchmarks were chosen because they exhibit a high degree of data parallelism. This characteristic also makes these codes highly amenable to software pipelining. However, as the table shows, a careful and balanced mapping of operations to the scalar and vector resources can provide superior performance.

Benchmark	Full Vectorization	Selective Vectorization
`alvinn`	1.08	1.18
`su2cor`	1.01	1.11
`swim`	1.04	1.18
`tomcatv`	1.07	1.26

Table 2: Speedup over modulo scheduling for a set of SPEC FP benchmarks.

6 Related Work

With the advent of short vector extensions, there is a need to supply access to these instructions to the high-level programmer. Most of this support has come in the form of inline assembly, macro calls, or specialized library routines. However, there has been success in providing automatic parallelization using traditional vectorization [6, 27]. Commercial products that make use of multimedia extensions include the Intel compiler [6], The VAST/AltiVec Compiler [3], and VectorC [1].

One of the major difficulties facing automatic vectorization for short vector extensions is the alignment restriction placed on vector memory operations. In [19], we proposed a static analysis for extracting alignment information. A nearly identical method was developed concurrently by Bik [6]. We also designed a system based on runtime profiling that transforms loops in an attempt to maximize the number of aligned references. Eichenberger and Wu [11, 32] have developed an effective method for reducing the cost of memory references when explicit instructions are required to handle misalignment.

Traditional vector compilation is largely based on constructing an accurate dependence graph of a loop [13, 17]. As mentioned, this is used as the basis for identifying data parallelism. Additionally, it is used as a framework for determining the validity of transformations that increase opportunities for vectorization. Classic examples include loop distribution, scalar expansion, and loop interchange. In this research we have made extensive use of the text by Allen and Kennedy [5]. Another excellent reference is the text by Wolfe [31].

Modulo Scheduling has been studied extensively in the literature. As a result, we can not list all the contributions made in the field. We have based our work primarily on Rau's original description of Iterative Modulo Scheduling [25]. Many of these techniques were developed at Cydrome for one of the first commercial VLIW architectures [8]. Similar techniques were developed concurrently by Lam [18]. Extensions to the core modulo scheduling algorithm include techniques to reduce register pressure [10, 14] and the ability to schedule loops with control flow [20]. Recently, there has been interest in modulo scheduling for clustered architectures [7, 33]. 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. In our work, partitioning is complicated by its connection to instruction selection. That is, choosing a vector versus scalar allocation also influences the specific opcode and number of operations that must be generated.

7 Conclusion

Short vector extensions have been integrated into the ISA of many general-purpose and embedded microprocessors. This has added a data parallel component to traditional ILP hardware. To exploit this design, we propose a new method for exploiting vector parallelism in software pipelined loops. Our approach selectively vectorizes operations in important program loops to improve resource utilization between the scalar and vector functional units. This results in lower minimum resource constraints and allows for software pipelining with shorter initiation intervals. Although our techniques complement statically scheduled machines most naturally, we believe they are applicable to any architectures that tightly integrate support for ILP and data parallelism.

In this paper we show that our strategy can lead to performance gains that exceed techniques targeting ILP or data parallelism alone. Furthermore, the methods presented here are applicable to machines with arbitrary resource constraints. An important aspect of this is the ability to manage explicit communication between vector and scalar instructions. Our methodology also allows for a natural handling of misaligned vector memory operations. For architectures that provide hardware support for misaligned references, software pipelining effectively hides the latency of these potentially expensive instructions. When explicit alignment is required in software, our algorithm accounts for these extra costs and vectorizes only when it is profitable. Finally, whenever alignment information is available, the information is easily incorporated into our heuristics.

Our compiler infrastructure is still developing. Currently, we are unable to accommodate loops containing control flow or early exits. In spite of this, we are able to achieve an average speedup of 1.30× and 1.18× on a set of DSP and SPEC FP benchmarks, respectively. When loops are fully vectorized before software pipelining, our optimization yields a speedup of 1.12× and 1.32×.

References

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

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

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

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

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

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

[7] 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.

[8] J. C. Dehnert, P. Y. Hsu, and J. 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] 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.

[10] 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.

[11] 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.

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

[13] G. Goff, K. Kennedy, and C.-W. Tseng. Practical Dependence Checking. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Ontario, June 1991.

[14] 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.

[15] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 1: Application Architecture*, October 2002.

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

[17] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, Williamsburg, VA, January 1981.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] Motorola. *AltiVec Technology Programming Environments Manual*, November 1998.

[24] 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.

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

[26] 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.

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

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

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

[30] 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.

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

[32] 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.

[33] 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.