Submitted to the 26^{th} *Annual International Symposium on Computer Architecture.*

Vector Microprocessors for Desktop Computing

Mark G. Stoodley and Corinna G. Lee
Department of Electrical and Computer Engineering
University of Toronto

23 October 1998

Abstract

Rapid advances in IC technology combined with recent trends in design complexity and application usage indicate that vector microprocessors can provide a scalable, cost-effective solution for desktop computers in the near future. In this paper, we review these trends and explain how they benefit vector processors. A decentralized design coupled with a simpler in-order issue mechanism will allow vector microprocessors to achieve the higher clock frequencies enabled by future IC technology. Moreover, the simpler vector hardware offers a favorable alternative to the increasing design sophistication of OOO superscalar processors. From the application side, it is expected that, by the year 2000, 90% of desktop cycles will be dedicated to multimedia workloads. For these applications vector microprocessors can supply up to twice the performance of today's out-of-order superscalar processors which are already aided by short-vector multimedia instructions.

One potential drawback to a vector-based desktop computer is the general belief that they can improve the performance of only vectorizable programs. In this paper, we use two approaches to address the problem of non-vectorizable performance.

First, we quantify the performance loss that might result and determine that the loss is not significant. This is because non-vectorizable applications tend to have only modest amounts of ILP. A 2-way in-order vector microprocessor should require no more than twice as many clock periods as a wider 4-way out-of-order processor to execute such applications. Some if not most of this cycle count difference should be made up by the greater clock rate possible by the vector microprocessor's decentralized design or by identifying vectorizable loops.

Second, we investigate the use of outer-loop parallelism to expand the number of loops that can be vectorized effectively. We show that directly vectorizing outer loops offers the greatest promise among compilation techniques that exploit outer-loop parallelism. The vector microprocessor can use outer-loop vectorization to execute select benchmarks 4 to 9 times faster than the OOO superscalar using loop interchange.

1 Introduction

Historically, vector architectures have been successfully implemented as supercomputers targeting scientific and engineering applications. In contrast, they have achieved limited success in less expensive technologies. Recent advances in IC technology, however, indicate that implementing a highly parallel vector microprocessor is no more expensive than current state-of-the-art microprocessor designs. In previous work, we assessed the implementation cost of various vector configurations and showed that high performance vector microprocessors require no more die area, and possibly significantly less area, than a 4-way, out-of-order superscalar processor [15][16]. Moreover, we also showed that vector implementations scale more cost-effectively than OOO superscalar designs when supporting more parallelism.

Having established the cost-effectiveness of vector microprocessors, we are broadening the context of our research and are now investigating the use of vector microprocessors in desktop computers. In addition to the rapid advances in IC technology, trends in design complexity and application usage suggest that vector microprocessors can provide a scalable, cost-effective solution for desktop computers in the near future. These trends have been reported elsewhere to explain the need to consider other alternatives for cost-effective desktop computing. Nonetheless, they are also equally applicable to vector microprocessors. In this paper, we review these trends and explain how they favor using vector microprocessors in desktop computers.

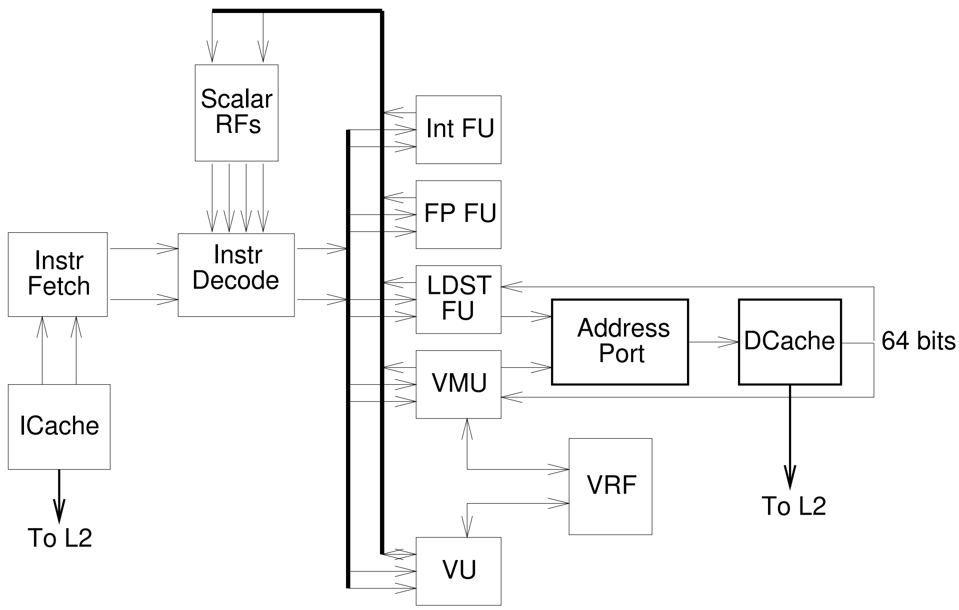
Although vector microprocessors may be appropriate candidates for desktops, it is generally believed that they can improve the performance of only vectorizable programs. Desktop computers, however, must also execute non-vectorizable applications efficiently. In our previous work, we did not address this perceived narrowness in applicability, having established the cost-effectiveness of vector microprocessors by conducting performance studies using easily vectorizable programs only.

In this paper, we turn our attention to the problem of non-vectorizable performance. We use a two-pronged approach to address this problem. First, we quantify the performance loss that a vector-based desktop might incur on non-vectorizable code. Second, we investigate a new technique that can expand the number of loops in multimedia applications that can be vectorized effectively and thus benefit from greater vector performance.

The rest of the paper is organized as follows. We begin in Section 2 with a description of a vector microprocessor that would be suitable for the desktop. Then in Section 3, we show how current trends in IC technology, design complexity, and application usage favor vector microprocessors. The remaining three sections address the problem of non-vectorizable performance. Section 4 briefly describes the benchmarks and processor configurations we use in our performance studies. In Section 5, we present evidence that the performance of nonvectorizable code may not be as bad as one might think when compared to current alternatives. In Section 6, we analyze the structure of multimedia applications and find that parallelism in outer loops can be extremely important to the performance of these applications. We compare the effectiveness of three compilation techniques that exploit outer-loop parallelism and find that Outer-Loop Vectorization shows greatest potential.

2 A Vector Microprocessor Design

We propose using a two-way issue, in-order superscalar processor tightly coupled with a vector processing unit as a scalable, cost-effective solution for desktop computing. We showed in a previous study that such a design achieves high performance on applications with abundant parallelism [16]. We will show later in this paper that this design also performs well enough on applications that are not highly parallel. A block



Issue Logic	
issue width	2
issue mechanism	in-order
Scalar Datapath	
#int FUs	1
#FP FUs	1
#LDST FUs	1

Vector Register File	
#vector registers	32
vector length	64
Vector Datapath	
VU width	8
#VUs	1
#VMUs	1

Memory Interface	
L1 data bus width	64 bits
L1-L2 data bus width	128 bits
#address ports	1

Figure 1: Vector Microprocessor Design

diagram of the design and a table listing its features are in Figure 1.

The vector unit is tightly coupled to the scalar processor in the sense that there is low latency communication provided between the vector unit and the scalar register file, allowing the vector unit to quickly access scalar values. Pairs of instructions (any combination of scalar and vector) are issued in program order and executed in either the scalar functional units or the vector units (VUs). The operands for scalar instructions are maintained in the scalar register file while vector operands are maintained in the vector register file (VRF).

The cache memory hierarchy is based on the MIPS R10000 [28]. The split L1 non-blocking caches are each 64KB, 2-way set-associative with 32B lines and 4 banks. The unified L2 blocking cache is 512KB, 4-way set-associative with 64B lines and 8 banks. The L1 hit time is 1 cycle, L1 miss penalty is 4 cycles, and the L2 miss penalty is 40 cycles.

Details of the vector hardware are shown in Figure 2, which illustrates the organization of the VU, the VMU, and how they are connected to the vector register file. For the sake of presentation, this figure shows

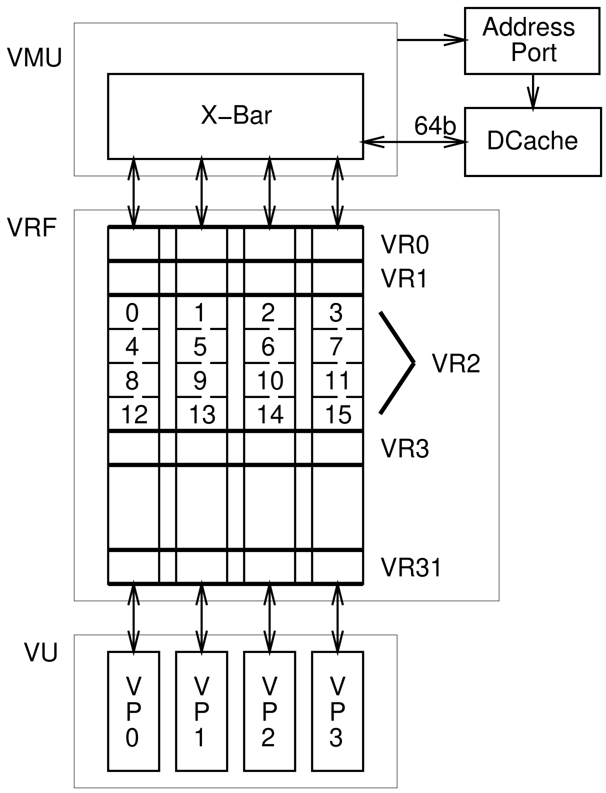


Figure 2: Vector Hardware Details

a vector unit with four parallel pipelines (VP0 - VP3) and vector registers with only 16 elements. However, the vector microprocessor we are proposing has 8 parallel pipelines and vector registers with 64 elements. The layout of register elements is shown for vector register 2 (VR2 in Figure 2).

Figure 2 illustrates two important vector parameters: vector width and vector length. The vector width determines the number of parallel pipelines in the vector unit. With a vector width of 4 as in Figure 2, a vector instruction is executed in groups of 4 parallel operations. The vector registers in the VRF are striped across the parallel pipelines so that the VRF can inexpensively supply a pair of operand values for each parallel operation. The vector length parameter determines how many total operations should be performed by a vector instruction. In previous work, we showed that longer vectors provide greater performance both by reducing the dynamic instruction count and by exploiting more parallelism to decrease the average cycles per operation [16].

The vector memory unit (VMU) provides a mechanism for moving vectors of data between the vector register file and the memory via a 64-bit data bus. The VMU shares the address port with the scalar load/store unit. Only one of these units is allowed to generate addresses in any cycle and, moreover, the VMU locks the address port for the duration of an entire vector instruction. Thus, the scalar unit cannot interrupt the VMU to access memory. This VMU design is similar to that of the Torrent T0 vector microprocessor designed at Berkeley [1].

The time to execute a vector memory instruction can vary tremendously and depends on the type of access (unit-stride, constant non-unit stride, or indexed), the data type of the memory instruction (byte, word, etc.), and the vector length. The reason for the large variation in execution time is due to the fixed width of the 64-bit data bus and the single address port.

Unit-stride accesses can easily utilize the full bandwidth of the 64-bit bus by using a single address. Thus, a 64-bit access from a single address can return several data values; the number of values returned is determined by the type of data being accessed. For example, a `vector load byte` fetches 8 elements per 64-bit access. With a vector length of 64, such an instruction would take 8 clock periods to execute, assuming all accesses hit in the L1 cache. A `vector load word` fetches only 2 elements per 64-bit access and thus takes 32 cycles to execute when its vector length is 64.

Non-unit strided and indexed vector memory instructions cannot utilize the full bus bandwidth with a single address. As a result, only one element at most can be accessed per clock period. Thus, regardless of the data type, strided and indexed memory instructions take VL cycles to execute, where VL is the vector length of the instruction.

3 Why Desktop Vector Microprocessors?

Rapid advances in IC technology combined with current trends in design complexity and application usage provide compelling reasons for using vector microprocessors in desktop computers. In this section, we review these trends and explain how they favor vector processors.

Both positive and negative trends in IC technology indicate that implementing a vector architecture in VLSI will result in a high performance, cost-effective design. One positive trend is the continuing enormous rate at which IC fabrication technology is improving: in the near future, computer architects are expected to have at their disposal one billion transistors on a single die [2]. Another positive trend is that clock frequencies have been growing at a tremendous rate. However, there is concern that this rate is in danger of slowing down due to the fact that wire delay is not scaling as quickly as gate delays [19]. As feature size shrinks to 0.1 micron and below, wire delays will begin to limit the on-chip distance a signal can travel in a single clock cycle.

One possible solution is to pipeline long wires to allow a shorter clock cycle. However, this may not always solve the problem. Additional pipeline stages to divide long wires may, for example, increase the amount of bypassing circuitry needed in the pipeline. If operand bypassing circuitry limits the performance of the processor, as has recently been found for out-of-order processors [24], then pipelining long wires may not result in a shorter clock cycle. Another solution is to use a decentralized architecture with only local communication and thus shorter wires. This is in contrast to architectures that are reliant upon large, centralized structures requiring more global communication.

Vector architectures tackle the wire-delay problem by virtue of being decentralized. A vector processor can be naturally partitioned into "lanes" as shown in Figure 2. Each lane consists of one pipeline from the vector unit and the bank of the vector register file that supplies that pipeline with operands. By replicating the vector unit's local control and the vector register file's decoder in each lane, these can be made completely independent of one another.

One complication is the crossbar in the vector memory unit. The crossbar must span the entire width of the vector register file that it connects to. Fortunately, this crossbar can be pipelined with only a single cycle of additional latency for vector memory instructions because all vector operand forwarding occurs through the vector register file and not through an explicit bypassing network as in the scalar pipelines.

In addition to trends in IC technology, another growing concern in the design of microprocessors is increasing complexity in the design of out-of-order superscalar processors. Dynamically managing greater amounts of instruction-level parallelism requires more complicated designs and a longer design cycle time. Moreover, the percentage of a processor's die area dedicated to control structures that manage parallelism increases at least quadratically as issue width increases [22]. This control circuitry is much more difficult to design than datapath structures; control circuits take longer to design, require more stringent testing as errors are more difficult to detect, and are difficult to design to meet high frequency clock targets. All of these effects add up to a longer design cycle and often result in implementations missing ship dates [8][7][9].

A vector microprocessor like the one described in Section 2 can avoid such design complexity by utilizing an in-order issue mechanism without sacrificing performance. An in-order processor is simpler to design, test, and make fast. Another important benefit is that it is easier to characterize the performance of in-order versus out-of-order designs, thus allowing performance results to be evaluated and understood more quickly [10].

Vector architectures also reduce design complexity by virtue of consisting primarily of datapath structures with simple control circuitry [13][16]. These structures are easily replicated to produce the vector hardware. This also allows designers to reuse designs across different processor generations and thus provide scalable multi-generational implementations with minimal effort.

Although the reduced complexity of a vector microprocessor permits a faster design cycle, the resulting simple design must still provide performance for important desktop computing applications. Trends in applications for the desktop together with IC technology trends suggest that a simple vector microprocessor can provide excellent scalable performance.

In particular, an application area that is growing in importance is multimedia [5]. By the year 2000, 90 percent of desktop computing cycles are expected to be spent on so-called “media” workloads [4]. Three key characteristics of multimedia applications favor the use of vector microprocessors for this application area.

First, many multimedia applications contain an abundance of parallelism. Previous work has demonstrated that highly parallel vector processors can exploit this abundant parallelism more effectively than traditional superscalar processors while requiring the same or less die area [15][16].

Secondly, the computer rather than a human will be the performance bottleneck even as computers become faster. This is because it is expected that the workload of multimedia applications, particularly those in image processing and graphics, will increase as computer performance improves. For example, the SIF (352x240) and CCIR 601 (720x486) image formats are the most common sizes currently supported by MPEG codecs [6]. While it may be possible in the near future for desktop computers to provide real-time video for SIF images using a software-only solution, there will be a demand for higher resolution formats such as EDTV (960x486) and HDTV (1920x1080).

To respond to such growing workloads, a computer design must scale in performance in a cost-effective and seamless manner. This is possible for vector architectures because of the third key characteristic of multimedia applications: greater amounts of parallelism are expected to accompany these increases in workload. Consequently, more highly parallel vector implementations can provide greater performance while less parallel implementations will be less expensive.

New instructions have been introduced into general-purpose microprocessors over the past five years to support multimedia applications [18] [23][17] [12][3][20]. However, because these instructions support a fixed amount of parallelism, these designs have several drawbacks especially with regards to scalability. For example, a `packed add` instruction operating on 16-bit values performs four adds in parallel with a 64-bit datapath. For a user to exploit, say, eight parallel adds in a future processor implementation, four things would have to occur: 1) a new instruction would need to be added to the instruction set which uses up precious opcode space; 2) the application would need to be rewritten to use this instruction because the compiler infrastructure to automatically exploit these instructions is not yet available; 3) the application would need to be recompiled; and 4) the application would need to be distributed to the users.

In contrast, a simple long vector microprocessor does not suffer from the four problems identified above. The ISA requires no additional changes to support greater parallelism in hardware. Furthermore, an application needs to be compiled only once to exploit the vector hardware but thereafter can take advantage of higher performance implementations without rewriting, recompiling or redistribution. Exploiting wider vector units or more vector units to improve performance via greater parallelism is handled by the vector unit control hardware and is invisible to the application.

Future implementations can also provide longer vectors (by increasing the size of the vector register file) and this, too, can be handled without recompiling code. The vector length is coded into a special register in the processor and because the application program can access this register, the vector compiler can easily generate code that dynamically stripmines loops according to the implementation's vector length; the striplength is determined when the program runs rather than when it is compiled [14].

Of course, there are other, non-multimedia applications that are important to desktop computing including the SPEC integer benchmark suite and productivity applications such as word processors and presentation software. These types of programs are generally believed to be non-vectorizable. We address the performance of non-vectorizable programs executing on a vector-based desktop computer in Section 5.

Finally, a vital factor in desktop performance is compiler technology. Major code restructuring, such as interchanging two loops in a loop nest, may be necessary to present an application's parallelism to the processor in a form that it can exploit, even for dynamically scheduling processors. While, in principle, a processor with a reorder buffer can perform this restructuring dynamically, in practice the reorder buffer is not always large enough; the window of available instructions does not span enough loop iterations to "see" the parallelism. Without compiler transformations such as loop interchange, it is possible for the application's parallelism to be completely inaccessible to a cost-effective dynamically scheduling processor [15].

On the other hand, extensive code transformation is also required for vector processors as well. Vector compiler technology is well established in the scientific domain and is very effective. Applying this mature 25-year old compiler technology to multimedia applications should be a small evolutionary step. In contrast, compiler technology for other architectures such as OOO superscalar, VLIW, and single-chip multiprocessors are more complex and may require additional research [11].

In summary, trends in IC technology, design complexity, and application usage favor vector microprocessors for desktop computing. Vector architectures provide a natural solution to mitigate the effects of longer wire delays and should thus be able to benefit from higher clock frequencies offered by future IC processes. The simpler control structure and dominant datapath structures of a vector microprocessor can reverse the undesirable trend of lengthening design times due to design complexity. Trends in application areas also favor vector architectures. Cost-effective scalable vector implementations are a natural match to the burgeoning application area of multimedia where highly parallel workloads are expected to increase with

SIMPLE Benchmark	Data Width	Input	Description
`colorspace`	8 bit	color image(s)†	Converts an image in RGB to YUV values.
`composite`	8 bit		Blends two images together by a blend factor *alpha*.
`convolve`	8,16 bit		Convolves an image with a 3x3 16-bit kernel.
`detect`	8,16 bit		Detects edges in an image based on a threshold.
`mxm`	32 bit	100x100 matrices	Multiplies two square matrices together.

COMPLEX Benchmark	Data Width	Input	Description
`chroma`	8 bit	color image(s)†	Merges two images on the basis of a "whiteness" threshold.
`compress`	8 bit 32 bit fp		Compresses an image using DCT.
`decrypt`	16 bit	16000 byte message	IDEA decryption.

† 320x240, 24-bit color

Figure 3: Benchmark Programs

improving computer performance. Finally, compiler technology is the most mature for vector architectures compared to others.

Although these trends indicate that vector microprocessors may be appropriate candidates for desktop computing, it is generally believed that they can improve the performance of only vectorizable programs. Desktop computers, however, must also execute non-vectorizable applications efficiently. In the remainder of this paper, we present results from our studies that investigated the problem of non-vectorizable performance.

4 Experimental Framework

Before presenting the results of our studies, we first describe the benchmarks and processors used in the studies.

Figure 3 shows the eight programs in our benchmark suite. Most of these contain memory accesses that have a constant, non-unit stride access pattern. All are highly vectorizable in that more than 90% of *scalar* execution time is spent in vectorizable functions. However, not all can be vectorized in a straightforward fashion due to control flow in a vectorizable loop or the bulk of the parallelism being in an outer loop. In the case of outer-loop parallelism, complex loop transformations are often required to enable vectorization. Thus we group the benchmarks into two categories. The SIMPLE category contains those that are easily vectorizable, while the COMPLEX category contains those that are more difficult to vectorize.

Figure 4 lists the salient features of the three processor configurations used in our studies.

Config	Issue Width	Issue Order	Vector Length		Vector Width	# VUs	VRegs renamed?	Speculative vector issue?
SS4	4-way	out	Not Applicable			Not Applicable		
SVX_FDW	4-way	out	8 4 2	(8b data) (16b data) (32b data)	8 4 2	1	Y	Y
LV	2-way	in	64	(all data)	8	1	N	N

Figure 4: Processor configurations to study outer-loop parallelism

The baseline configuration, SS4, is a 4-way issue, out-of-order superscalar processor based on the MIPS R10000 design but with a larger reorder buffer of 56 entries. We include this configuration to demonstrate how little today's superscalar processors are able to exploit outer-loop parallelism even when the code is restructured (via loop interchange) to present the parallelism to the processor in a form it can easily exploit.

The second processor, SVX_FDW, is similar to the baseline configuration except that it has a dedicated vector unit with a maximum vector length of 8 and maximum vector width of 8. The FDW suffix stands for "Fixed Data Width" indicating that vector length and width depend on whether byte, half-word, or word data are being operated on. Since vector instructions are short and execute in one cycle, this configuration models an out-of-order superscalar processor with multimedia instructions. Note that this comparison is optimistic since current multimedia implementations are not supported by as versatile a vector memory unit as we simulate in this study. In particular, strided memory accesses are *not* supported directly in hardware in existing implementations.

The third processor, LV, is the vector microprocessor described in Section 2. The vector execution for this model differs from the SVX_FDW processor in that vector instructions are issued in order, vector length is 64, vector width is always 8, vector registers are not renamed, and vector instructions cannot issue speculatively.

All three processors use the same 2-level cache-based memory hierarchy described in Section 2.

5 Non-Vectorizable Code Performance

The idea of a vector microprocessor being used for desktop computing may raise the concern that the performance of non-vectorizable programs, such as the SPEC integer benchmarks and productivity applications, may be inferior to the performance provided by the out-of-order superscalar processors used in today's desktop computer. In this section, we discuss two key characteristics of such applications that benefit vector processors by mitigating the reduced performance of non-vectorizable code. As part of this discussion, we argue that the performance of the vector microprocessor described in Section 2 should not be far behind that of an out-of-order superscalar processor, if at all.

The first key characteristic is that applications such as the SPEC integer benchmarks have only a moderate amount of extractable ILP. Although limit studies indicate that high levels of ILP (10 IPC) are present in such programs [26], architectural and compiler innovations over the past decade have not been highly successful at extracting this parallelism to improve performance.

For example, the PowerPC 620 was the first microprocessor to use out-of-order issue, branch prediction, multi-entry reservation stations, and register renaming all on a single chip [25]. Despite its aggressive design with 4-way issue and six functional units, this processor graduates no more than 1.5 instructions per cycle, achieving IPCs of 0.99 to 1.44 (average 1.23) on `eqntott`, `li`, `go`, and `compress` [25]. This is less than 37% of the 620's peak issue rate.

A simple scalar processor, which typically averages about 0.8 IPC [10], could achieve comparable performance on these programs if it uses a clock frequency that is approximately $1.23 \div 0.8 = 1.53\times$ faster than the PowerPC 620's clock rate. The 2-way, in-order vector microprocessor described in Section 2 should achieve a higher IPC than 0.8 and therefore require a clock frequency less than $1.5\times$ that of the out-of-order, 4-way 620 processor to provide comparable performance. As discussed in Section 3, a $1.5\times$ faster clock should be easily achievable with the simple decentralized design of the LV vector microprocessor.

In another study, Olukotun et al. compared the performances of out-of-order 2- and 6-way issue processors based upon the MIPS R10000 design[22]. Even though the 6-way issue processor had a reorder buffer eight times larger (128 entries versus 8) and three times the number of functional units, the difference in average graduated IPC was only a factor of 1.4. Again, this small difference in IPC could be counterbalanced by increasing the clock rate of the 2-way issue processor by more than 1.4 times.

Thus it is likely that the difference in cycle count for the LV processor relative to the SS4 processor will not be large when executing non-vectorizable programs. This cycle difference could be compensated for by the vector microprocessor's faster clock. Due to the increasing wire-delay trend described in Section 3, it is possible that a more aggressive OOO implementation would not achieve as high a clock rate.

To obtain an approximate upper bound on the difference in cycle count that we expect the 2-way, in-order LV design to incur on non-vectorizable code, we conducted a simulation study using non-vectorized versions of the multimedia benchmarks described in Section 4. We compiled these benchmarks using -O2 with the SGI C compiler and simulated their performance on LV and SS4. Since the SGI compiler does not generate any vector code, the difference in cycle counts between these two processors will provide an indication of the performance degradation for scalar code incurred by LV *when assuming the same clock frequency for both processors.*

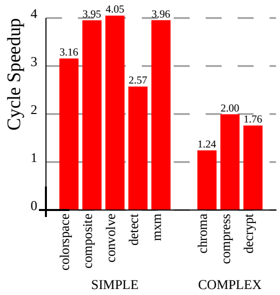


Figure 5: Cycle Speedup of SS4 over Vector Microprocessor on Non-Vectorized Code

Figure 5 shows the cycle speedup for the out-of order superscalar processor over the in-order vector microprocessor. The benchmarks are shown grouped into the SIMPLE and COMPLEX categories based on the ease of vectorization. We expect the non-vectorized execution of the COMPLEX programs to be similar to the execution of non-vectorizable programs such as the SPEC benchmarks.

The SIMPLE benchmarks have abundant parallelism that SS4 can effectively exploit as the large speedups in Figure 5 show. SS4 achieves a cycle speedup of more than 2.5 for all the benchmarks in this category. These benchmarks serve to demonstrate that wide out-of-order execution can effectively use the parallelism in easily vectorizable code. However, as we will show in the next section and have shown elsewhere [16], a simple long vector processor provides even greater performance on this type of code.

The parallelism in the COMPLEX benchmarks, on the other hand, is not as readily accessible to the out-of-order processor. SS4 achieves a cycle speedup of two or less over 2-way, in-order execution. These speedups are similar to those reported for the SPEC benchmarks, which are non-vectorizable.

These results suggest that a 2-way in-order vector microprocessor should require no more than twice as many clock periods as a wider 4-way out-of-order processor to execute non-vectorizable applications. Some if not most of this cycle count difference should be made up by the greater clock rate possible by the vector microprocessor's decentralized design.

Additionally, it may be possible to improve performance by vectorizing some portions of these programs even though they are widely believed to be non-vectorizable. In his doctoral thesis, Asanović investigated the potential for vectorizing benchmarks in the SPEC95 integer suite [13]. He demonstrated that substantial speedups can be achieved by rewriting 4 of the 8 benchmarks to vectorize several key loops with very few localized code modifications. As few as 0.3% of the source code lines in one program had to be modified, and at most 7.5% in another. The vectorized programs execute 16% (`compress`) to 350% (`ijpeg`) faster on the Torrent T0 vector microprocessor [13]. These individual speedups result in an average speedup of 1.32 over T0 scalar execution for the *entire* SPEC95 integer suite.

In brief, non-vectorizable applications contain modest amounts of extractable ILP. While a simple long vector microprocessor may not effectively use this ILP, any performance inefficiencies could be overcome by improvements in clock frequency and possibly identifying vectorizable loops in such applications.

The second key characteristic of non-vectorizable codes that suggests that any performance loss may not have significant impact is that the human user will likely be the performance bottleneck rather than the computer, especially for productivity tools such as word processors which involve heavy interaction with the user. At some point in the near future if not already, computers will be fast enough to make almost all of the wait time be on the user side. Moreover, it is unlikely that the workload of such tools will increase at the same rate that computer performance is improving. This is in contrast to the situation for multimedia applications where, as pointed out in Section 3, it is expected that the computer will be the bottleneck for some time.

6 Vectorizing Loops in Multimedia Applications

The previous section investigated the performance of non-vectorizable code on a vector microprocessor. In this section, we investigate whether compiler techniques can be used to increase the number of loops that can be vectorized. Vector compilers primarily target parallelism in inner loops. While an application's inner loop may not contain much parallelism, there could still be abundant parallelism in outer loops. For codes with these characteristics, it would be beneficial to extend the available vectorization techniques to enable the vector hardware to be exploited. One compilation technique that appears promising is *outer-loop vectorization* (OLV). To determine if the effort to implement OLV in our vectorizing compiler would be warranted, we conducted a feasibility study. We present the results of that study in this section.

6.1 Outer-Loop Parallelism in Multimedia Applications

Due to the growing importance of multimedia workloads to desktop computing [4], we first investigated how prevalent outer-loop parallelism (OLP) is in these applications. Unlike traditional desktop workloads like the SPEC95 integer suite, the computations performed in multimedia applications share many common features with scientific codes where loop parallelism is prevalent. Many multimedia computations require outer loop nests that iterate over a large number of objects and process each object independently of the other objects.

As an example, consider motion estimation, the dominant operation in MPEG encoding. The outer loops iterate over the 16x16 pixel *macroblocks* in each frame, computing the sum of absolute differences (SAD) between each macroblock and neighbouring macroblocks in the previous or following frames. There are many macroblocks to process since frames are large (352x240 for SIF ranging to 1920x1080 for HDTV). Furthermore, even as processor performance improves, users demand larger frame sizes for improved visual quality; the number of macroblocks to process will increase to saturate processor performance.

Although the outer loops in multimedia applications have a large trip count, each individual object does not typically require much processing. Processing one object may not require any loops, in which case the loops iterating over the objects can be easily vectorized. If loops are required to process each object, however, these loops often have short trip counts (less than 20) and may contain recurrences that prevent vectorization. Even if these loops are parallel, the short trip counts mean that vectorizing will not result in much speedup.

Returning to the motion estimation example, the SAD calculations are performed on 16x16 pixel macroblocks. The inner loops are small (16 in this case) and perform a reduction offering little parallelism to exploit. Moreover, execution speed does not dictate the size of each macroblock: they are chosen to provide good image and compression quality and hence their size is unlikely to change. Despite the poor scalability of the parallelism in the sum-of-absolute-differences calculation, almost all modern multimedia instruction sets provide an SAD instruction that accelerates computing the difference between two macroblocks despite the abundance of parallelism between macroblocks of the same frame.

In multimedia applications, then, exploiting the abundance of parallelism at the outer-loop level can be critically important to performance because parallelism in inner loops is not always present due to reductions and may be limited by short trip counts.

6.2 Exploiting Outer-loop Parallelism

Having established the abundance of OLP in multimedia applications, we now consider techniques to vectorize outer-loop parallelism. Compilers for scientific applications use a variety of techniques to exploit parallelism in outer loops. Most of these techniques transform the loop nests so that the outer loop parallelism is brought to the inner where traditional vectorizing techniques can be applied [27]. These techniques can also be applied to multimedia programs but some cannot be applied in all situations and some are not useful until the code has been extensively restructured first. The three techniques we describe here are loop unrolling, loop interchange, and outer-loop vectorization. We do not discuss other techniques such as loop coalescing and loop collapsing which are only useful when the inner loops are already parallel and hence vectorizable.

We will use a matrix multiply example to demonstrate how these transformations expose outer-loop parallelism. Figure 6a gives the original matrix multiply code to multiply matrices A and B and store the product in matrix C.

A simple technique to vectorize an outer loop is to completely unroll any nested inner loops; less than complete unrolling leaves an inner loop which prevents vectorizing the outer loop. Complete unrolling

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
        sum=0;
        for (k=0; k<N; k++)
            sum+=A[i][k]*B[k][j];
        C[i][j]=sum;
    }
```

(a) Original matrix multiply

```
N = 4;
for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
        sum=0;
        sum+=A[i][0]*B[0][j];
        sum+=A[i][1]*B[1][j];
        sum+=A[i][2]*B[2][j];
        sum+=A[i][3]*B[3][j];
        C[i][j]=sum;
    }
```

(b) Unrolled with N = 4

```
for (i=0; i<N; i++) {
    for (j1=0; j1<N; j1++)
        sum[j1]=0;
    for (k=0; k<N; k++)
        for (j2=0; j2<N; j2++)
            sum[j2]+=A[i][k]*B[k][j2];
    for (j3=0; j3<N; j3++)
        C[i][j3]=sum[j3];
}
```

(c) Interchanged

```
for (i=0; i<N; i++)
    for (j=0; j<N; j+=VL) {
        sum[0:VL]=0;
        for (k=0; k<N; k++)
            sum[0:VL]+=A[i][k]*B[k][j:j+VL];
        C[i][j:j+VL]=sum[0:VL];
    }
```

(d) Outer-loop vectorized

Figure 6: Matrix multiply code

requires that the compiler know the inner loops' trip counts and is only feasible if the inner loops have short trip counts to prevent huge code expansion. Since the inner loops in multimedia applications often have short and known trip counts, unrolling can effectively expose outer-loop parallelism in these applications. Even so, if there is more than one inner loop to unroll, the code expansion can be prohibitive. Figure 6b shows matrix multiply unrolled assuming N is 4. By completely unrolling the k loop, the j loop can now be vectorized. Moreover, the loop overhead of the inner loops is completely eliminated.

A second approach to exposing outer-loop parallelism is to transform the loop nest so that the outer (parallel) loop is moved to the innermost position in the nest. This is the approach typically used in vectorizing compilers but it has several drawbacks. Because this technique can only be applied to perfect loop nests[1], loop distribution and usually privatization are often applied before interchanging. The matrix multiply loop nest is not perfect because of the assignments to `sum` and the C matrix element in the j loop. The interchanged loop is shown in Figure 6c after privatizing the sum variable and distributing the j loop.

The distributed j loops ($j1$, $j2$ and $j3$) can now be vectorized since they are parallel inner loops. Because of the distribution, we would expect to see an increase in the number of scalar instructions: the loop overhead of the original j loop will be incurred three times. Privatizing the `sum` variable has two detrimental effects: more memory is required to hold the `sum` array, and the $j2$ loop now executes 4 memory operations per iteration because the sums can no longer be maintained in a register. This is in contrast to the original

[1] Perfect loop nests are those where all statements are contained in the innermost loop and not in any outer loop.

code where the `sum` variable could be maintained in a register and therefore the inner loop only executes 2 memory operations per iteration.

Loop interchange has several drawbacks. Loop distribution and privatization can degrade performance by replicating loop overhead and introducing additional memory instructions. In addition, because loop interchange relies on loop distribution and privatization to create perfect loop nests, it requires significant compiler technology to be effective. This technology is common in parallelizing vector compilers but may represent a significant implementation hurdle for scalar compilers. A final disadvantage is that specific types of dependences in the loop nest can disallow loop interchange [27], although we do not know how common these types of dependences are in multimedia applications.

Whereas loop unrolling and loop interchange extensively restructure a loop nest to expose outer-loop parallelism, outer-loop vectorization simply vectorizes the outer loop in place. The steps used to vectorize an inner loop can, with a few simple modifications, be applied to an outer loop [21]. OLV has been implemented in compilers for scientific applications (by Cray), but its performance has not been extensively studied or compared against alternative techniques. More relevant to our research is the fact that OLV has not been studied in the context of multimedia applications.

Figure 6d shows the matrix multiply code with outer-loop vectorization applied. In this code, N is assumed to be a multiple of VL, the processor's vector length[2]. In reading Figure 6d, it is important to know that the notation $I{:}J$ in Figure 6d refers to indices I,$I+1$,$I+2$,...,$J-1$. Thus, the reference `sum[0:VL]` refers to a vector of VL elements.

OLV has many features that make it attractive for exploiting outer-loop parallelism in multimedia applications. It does not rely on the inner loop's trip count being small or known and so is applicable in more loop nests than unrolling. Moreover, there is no code expansion because the inner loop does not have to be unrolled. Whereas interchanging the loops forces the `sum` array to be stored in memory and increases scalar overhead by distributing loops, OLV allocates the sums to a vector register and maintains the same loop structure as the original code. A further benefit of vectorizing the outer loop in place is that the scalar overheads of both the outer and inner loops are reduced, although not as much as in the unrolled case. The privatization analysis required for outer-loop vectorization, however, may require more complexity than required to interchange loops or to vectorize inner loops [21].

[2] This is to simplify the code in this example. It is not a restriction of the algorithm.

6.3 Exploiting OLP in Multimedia Applications

Outer-loop vectorization appears to have many advantages over other techniques. But before embarking on a time-consuming implementation in our compiler, we first determined if these advantages would translate into significant performance gains. To this end, we analysed our multimedia benchmarks suite for the presence of outer-loop parallelism that could be exploited by the techniques of the previous section. Five of eight benchmarks contained significant outer-loop parallelism. To evaluate the benefit of outer-loop vectorization, we applied loop unrolling, loop interchange, and OLV to four of the benchmarks (`composite`, `decrypt`, `detect`, and `mxm`) and compared the performance of the different versions. The fifth benchmark (`compress`) was unfortunately too complicated to transform by hand in a reasonable amount of time.

We applied loop unrolling at the source code level to `detect` and `decrypt`. These two benchmarks each have inner loops with trip counts small enough that unrolling is feasible. `Detect` contains a 3x3 filter convolution that unrolls to 9 loop iterations. `Decrypt` contains a single loop with a trip count of 8. We could not apply loop unrolling to `composite` or `mxm`; the inner loop trip count in `composite` is not known at compile time, and `mxm`'s inner loop trip count was too large (100) to reasonably unroll.

Loop interchange (and any necessary privatization and loop distribution) was applied at the source level to all four benchmarks to bring the innermost *parallel* loop to the innermost loop position.

For the SVX_FDW and LV processors, the unrolled and interchanged versions of each benchmark were generated by our VSUIF vectorizing compiler. For the SS4 processor, the SGI C compiler was used with -O2 to provide a fairer comparison since our vector compiler does not generate efficient scalar code.

Outer-loop vectorization was applied to all four benchmarks by writing assembly code, mimicking the actions of a vectorizing compiler. In all cases this was a straight-forward transformation of the code with privatization applied. The code created was similar in quality to that generated by the vector compiler for the unrolled and interchanged versions.

The performance of the four benchmarks is summarized in Figure 7. For each processor, there are four bars, one for each version of a benchmark: original (ORIG), interchanged (IC), unrolled (UR), and outer-loop vectorized (OLV). Where NONE appears, that version of the code does not exist for that processor[3]. For instance, there is no OLV benchmark for the SS4 processor because it has no vector hardware.

The first row of graphs shows the speedup over the performance of the original code executed on the SS4 configuration. The second and third rows of graphs show the cycles per operation (CPO) and number of dynamic operations (N), respectively. These are included to show why the speedup values differ across the

[3]NOTE TO REVIEWERS: Two failed simulations show NONE in the graphs: `decrypt`(IC) and `detect`(ORIG) both on SVX_FDW. These results will be ready for the final version of the paper. Other closely related simulations convince us that there will be no surprises when these results are ready.

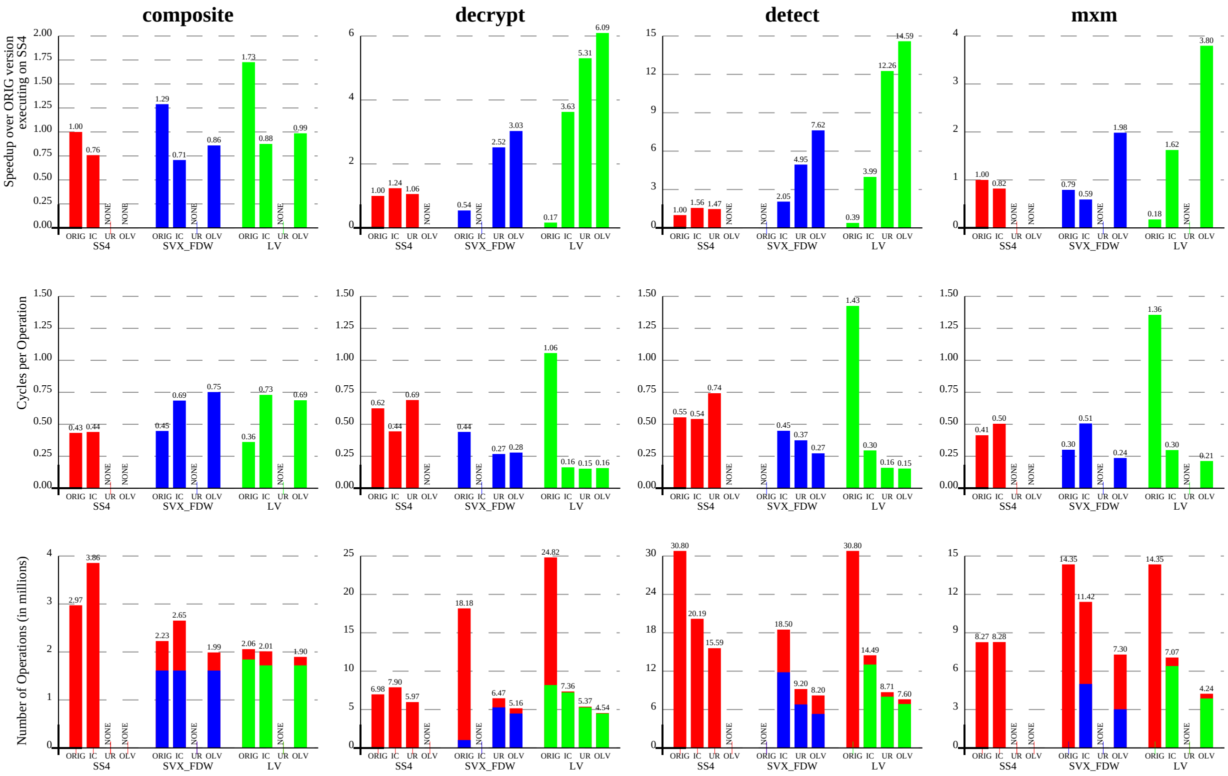


Figure 7: Performance graphs for exploiting outer-loop parallelism

different code versions. We use CPO rather than CPI because a vector instruction causes a maximum of 64 operations to execute. Because of this, CPI is a less interesting measure for comparing vector configurations to the superscalar configuration where there is a one-to-one correspondence between instructions and operations. The dynamic operations graph bars are divided for the vector configurations: the lower gradation represents vector operations, the upper gradation represents scalar operations. The scales of the vertical axes on the graphs in Figure 7 require special mention. The scales on the first and third rows of graphs are different for each benchmark, while the scales in the second row are the same for all benchmarks. The scales were determined by the size of the largest bar.

We discuss `decrypt`, `detect`, and `mxm` together because they have similar performance results. The original code for these three benchmarks is not vectorizable because of recurrences in the innermost loop. This means that for both the LV and SVX_FDW processors, the vector performance of `decrypt`, `detect`, and `mxm` depends greatly on exploiting OLP. For example, the OLV version of `detect` runs over 38 times faster than the original version executing on the LV processor. Even loop interchange, the least effective of the OLP techniques, achieves speedups of more than 9 over the original codes executing on the LV processor.

OLV is the most successful technique at exploiting outer-loop parallelism although unrolling, where it is applicable and feasible, comes close. We believe that the two techniques are equally effective and the difference is mainly because the OLV versions are handcoded while the unrolled versions are compiled.

The substantially poorer performance of loop interchange versus unrolling and OLV can be attributed to three factors: 1) loop distribution adds to the loop overhead of the outer loop, increasing the dynamic scalar operation count, 2) privatization adds vector memory instructions to the inner loop as occurred for the matrix multiply example in Figure 6c, and 3) the additional vector memory instructions sometimes miss in the cache, increasing the CPO. Lower quality scalar code generated by our vector compiler may be responsible for the first factor since it did not affect the performance of SS4, but the other two factors arise from the loop interchange technique itself.

The SVX_FDW processor, limited by shorter vector length and despite its out-of-order issue, could not match LV's performance. The difference ranged from about 15% for `decrypt` to 85% for `detect`. In part this difference can be attributed to executing more operations, but increased CPO is the main reason. Out-of-order issue simply cannot keep the vector unit as busy as longer vector instructions can.

For `decrypt` and `mxm`, an additional factor contributes to LV's superior performance. The wider data types used by these benchmarks narrow the vector width used by SVX_FDW because of its fixed width datapath. LV executes vector operations eight per cycle whereas SVX_FDW executes four and two vector operations per cycle for `decrypt` and `mxm`, respectively. To measure the impact this had, we conducted an experiment where we increased the vector width and vector length of SVX_FDW for `decrypt` and `mxm` to 8 (the same as the LV processor's width). In both cases, the speedup approximately doubled. While the reduced vector width is partially responsible for the performance difference between LV and SVX_FDW, the reduction in CPO caused by using longer vectors is still significant.

On the superscalar side, performance is also sensitive to coding style but the difference in performance among the different versions is not nearly as great as the differences observed for the vector configurations. The best coding style is loop interchange because this transformation exposes the outer loop's parallelism to the superscalar processor in a form that it can more easily exploit. Compared to the unrolled version, the interchanged version of each benchmark achieves a lower CPO but a higher dynamic operation count. The lower CPO results from the exposed parallelism and more than compensates for the increased dynamic operation count.

Although outer-loop parallelism is essential to the performance of `decrypt`, `detect`, and `mxm`, the `composite` benchmark benefits much less. Both the inner and outer loops in `composite` are parallel but only the inner loop accesses data with unit stride. The effectiveness of exploiting outer-loop parallelism in non-unit stride outer loops can be measured with this benchmark. Since our vector memory unit can only return one data value per access for non-unit stride accesses, vectorizing non-unit stride loops may not provide a great deal of benefit, especially if the loop is memory-bound, which is the case for `composite`. All of the OLP techniques perform worse than the original code for all the processors. This is due to the higher CPO which is caused by the slow rate at which data values are brought in from the memory system. Without better support for strided vector memory accesses, memory-bound applications with outer-loops that access strided data will not benefit from techniques that exploit parallelism at the outer-loop level.

The poor performance exhibited by the `composite` benchmark indicates that there is a class of applications for which the desktop microprocessor design in Section 2 may not be well suited: applications with non-parallel inner loops but whose parallel outer loops access memory with nonunit stride. More investigation is warranted to determine whether many applications contain only outer-loop parallelism and require support for strided accesses in order to benefit from outer-loop techniques.

7 Summary

In this paper, we have argued that current trends in IC technology, design complexity, and application usage make vector microprocessors an attractive choice for desktop computing. A decentralized design coupled with a simpler in-order issue mechanism will allow vector microprocessors to achieve the higher clock frequencies enabled by future IC technology. Moreover, because vector hardware is primarily datapath circuitry, it is easier to design, verify, and to reuse components, a favorable alternative to the increasing design sophistication of OOO superscalar processors. With regards to applications, one expert has predicted that, by the year 2000, 90% of desktop cycles will be dedicated to multimedia workloads. For these applications vector microprocessors can supply up to twice the performance of today's out-of-order superscalar processors which are already aided by short-vector multimedia instructions.

We have also investigated the performance of the applications that would execute in the remaining 10% of desktop cycles. First we quantified the potential performance loss and determined that the loss is not significant. These applications are either characterized by modest amounts of parallelism which even current superscalar processors have had only limited success in exploiting (SPEC applications) or the human user is or will soon be the bottleneck to increased performance (word processors, presentation software). A 2-way in-order vector microprocessor should require no more than twice as many clock periods as a wider 4-way out-of-order processor to execute such applications. Some if not most of this cycle count difference should be made up by the greater clock rate possible by the vector microprocessor's decentralized design or by identifying vectorizable loops.

The second way we tackled the problem of non-vectorizable performance was to investigate the use of outer-loop parallelism to expand the number of loops that can be vectorized effectively. After demonstrating the importance of exploiting outer-loop parallelism for multimedia applications, we compared the performance of three compilation techniques that enable the vectorization of this type of parallelism: loop interchange, loop unrolling, and outer-loop vectorization. We showed that directly vectorizing outer loops provides the best performance for three of the four benchmarks we analyzed. For these three benchmarks, the vector microprocessor can use OLV to execute 4 to 9 times faster than the OOO superscalar executing the fastest version.

For the fourth benchmark, exploiting outer-loop parallelism results in strided memory accesses which execute much slower than unit-stride accesses for the vector memory unit we simulated. Further investigation is required to determine whether these types of loops are common in desktop workloads; if so, vector microprocessors may require hardware support for fast non-unit strided memory accesses.

Future research includes examining memory system performance particularly in the face of increasing memory latencies and bandwidth. The vector microprocessor we propose, with its in-order design, does not have the ability to find instructions beyond a cache miss to hide miss latency. A concern is that as memory latencies increase, the performance benefits of vector microprocessors may be lost due to its potentially poor latency tolerance. A preliminary set of simulations has shown that this is not the case for our benchmarks, but further investigation is needed to understand the reasons why. We have also found that the existing level of memory bandwidth does not limit performance in our vector microprocessor design but again more simulations are required to understand why this is so.

References

[1] Krste Asanović and David Johnson. Torrent Architecture Manual, Revision 2.11. Technical report, International Computer Science Institute, 1994.

[2] Doug Burger and James R. Goodman (Guest Editors). Billion-Transistor Architectures. *IEEE Computer*, 30(9):46–48, September 1997. Special issue on The Future of Microprocessors.

[3] Digital Equipment Corperation. Advanced Technology for Visual Computing: Alpha Architecture with MVI. *http://www.digital.com/-semiconductor/-mvi-backgrounder.htm*, March 1997.

[4] W. Dally. Tomorrow's Computing Engines. *Keynote speech at Fourth International Symposium on High Performance Computer Architecture*, February 1998.

[5] Keith Diefendorff and Pradeep K. Dubey. How Multimedia Workloads Will Change Processor Design. *IEEE Computer*, 30(9):43–45, September 1997. Special issue on The Future of Microprocessors.

[6] Didier Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[7] Linley Gwennap. Class of '94 Has Mixed Success. *Microprocessor Report*, 11(14), Oct 27 1997.

[8] Linley Gwennap. Is It Soup Yet? *Microprocessor Report*, 11(2), Feb 19 1997.

[9] Linley Gwennap. RISC Disappointments Mount. *Microprocessor Report*, 11(17), Dec 29 1997.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan Kaufmann Publishers, San Fransisco, CA, 1996.

[11] Ken Kennedy. *High Performance Compilers*. Morgan Kaufmann, 1999 (expected).

[12] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The Visual Instruction Set (VIS) in UltraSPARC. In *Proceedings of the Compcon-95*, pages 462–469, March 1995.

[13] Krste Asanović. *Vector Microprocessors.* University of California at Berkeley, 1998. Ph.D. Thesis.

[14] Krste Asanović. *Vector Microprocessors.* University of California at Berkeley, 1998. Ph.D. Thesis, Section 5.1.4.

[15] Corinna G. Lee and Derek J. DeVries. Initial Results on the Performance and Cost of Vector Microprocessors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 171–182, December 1997.

[16] Corinna G. Lee and Mark G. Stoodley. Simple Vector Microprocessors for Multimedia Applications. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998. To be published.

[17] Ruby B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):52–59, August 1996.

[18] Ruby B. Lee and Michael D. Smith (Guest Editors). Media Processing: A New Design Target. *IEEE Micro*, 16(4):6–9, August 1996. Special topics issue on media processing.

[19] Doug Matzke. Will Physical Scalability Sabotage Performance Gains. *IEEE Computer*, 30(9):37–39, September 1997. Special issue on The Future of Microprocessors.

[20] Inc. MIPS Technologies. MIPS Extension for Digital Media with 3D. *http://www.mips.com/-arch/-ISA5/-#MDMX_indx*, March 1997.

[21] Viet Ngo. *Parallel Loop Transformation Techniques for Vector-Based Multiprocessor Systems.* University of Minnesota, 1994. Ph.D. Thesis.

[22] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of ASPLOS VII*, pages 2–11, Cambridge, MA, October 1996.

[23] Alex Peleg and Uri Wieser. MMX Technology Extensions to the Intel Archetecture. *IEEE Micro*, 16(4):10–20, August 1996.

[24] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[25] Trung A. Diep, Christopher Nelson, and John Paul Shen. Performance Evaluation of the PowerPC 620 Microarchitecture. In *Proceedings of 22nd Annual International Symposium on Computer Architecture*, pages 248–259, 1995.

[26] D.W. Wall. Limits of Instruction-Level Parallelism. *Proceedings of ASPLOS-VI*, pages 248–259, April 1991.

[27] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing, 1996.

[28] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.