Published in the *Proceedings of the 30^{th} Annual International Symposium on Microarchitecture*, pp 171–182, December 1997. 171

Initial Results on the Performance and Cost of Vector Microprocessors

Corinna G. Lee and Derek J. DeVries
{corinna,devrier}@eecg.toronto.edu
Department of Electrical and Computer Engineering
University of Toronto

Abstract

Increasingly wider superscalar processors are experiencing diminishing performance returns while requiring larger portions of die area dedicated to control rather than datapath. As an alternative to using these processors to exploit parallelism effectively, we are investigating the viability of using single-chip vector microprocessors. This paper presents some initial results of our investigation where we compare the performance and cost of vector microprocessors to that of aggressive, out-of-order superscalar microprocessors.

On the performance side, we show that vector processors are able to execute a highly parallel, integer-based application 1.5–7.3 times faster than superscalar processors can by exploiting parallelism more effectively. This ability stems from the use of vector instructions. Vector instructions exploit parallelism *across* loop iterations by implicitly re-scheduling operations and *temporally localizing* the parallelism. Vector instructions also reduce instruction bandwidth by more than an order of magnitude because they express an abundance of parallelism in a compact encoding.

On the cost side we show that, to achieve these performance gains, highly parallel, integer-based vector microprocessors are *no more* costly to implement than existing in-order and out-of-order superscalar microprocessors. One reason for this is that the organization of a vector register file provides tremendous bandwidth without incurring a large area penalty. A second reason is that the control logic for issuing vector instructions is relatively simple.

Both the performance gains and cost savings are possible because vector processors rely on a vectorizing compiler, rather than hardware, to detect parallelism and to express it in a compact form to the hardware. These initial results suggest that transferring this functionality to the compiler offers a tremendous performance/cost benefit.

1 Introduction

Growth rates in VLSI technology are expected to continue at their current pace such that, by the year 2007, there will be 20 billion transistors available on a single die [4]. Based on trends in microprocessor designs over the past seven years, this virtual cornucopia of single-die resources is likely to be used to improve the performance of ever wider superscalar processors capable of exploiting larger amounts of instruction-level parallelism (ILP) [5]. State-of-the-art commercial microprocessors today are among the largest VLSI chips produced and are capable of issuing between 4 and 6 instructions out of order [32, 16, 14].

Research on improving superscalar performance, however, suggests that superscalar processors wider than 4-issue may not be the most effective technique for exploiting ILP. Ongoing research also suggests that implementing wider superscalars may not be the most efficient use of VLSI resources. Even today, a significant portion of the die for most commercial superscalar processors is needed to determine which instructions to execute in parallel. In contrast, the die area used to actually execute operations is relatively small.

With the diminishing performance returns of increasingly wider superscalar processors and larger portions of die area dedicated to control rather than datapath, researchers are exploring alternatives to superscalars to use single-chip resources more effectively. Olukoton *et al.* at Stanford University and Mowry at the University of Toronto are investigating the use of a small number of processors combined with a multilevel cache system on a single chip [23, 26]. There has also been renewed interest in VLIW architectures [22, 7]. A third alternative, simultaneous multithreading, has been shown by Tullsen *et al.* to be effective at using the resources in a wide out-of-order superscalar [28, 27].

We propose exploring yet another alternative: extending existing microprocessor instruction sets to incorporate a vector architecture. Vector architectures have had successful commercial implementations across a broad range of computer sizes for more than 20 years. For various reasons, however, vector architectures have not been considered at the microprocessor level until very recently. One reason is that a vector architecture can be area-intensive due to its large vector register file and numerous functional units, undesirable features during the first 25 years of VLSI technology when area was at a premium. Another is that a vector architecture relies on sophisticated compiler technology so that a wider community of users can benefit from its performance-enhancing features, a disadvantage also of multiprocessors and VLIW architectures. A third reason is that a vector architecture can exploit only certain types of parallelism while other architectures can potentially exploit all types.

Despite these drawbacks, over the last three years, almost every major microprocessor company has introduced limited vector extensions to their instruction sets for the purposes of supporting multimedia applications [18]. These vector extensions use short vectors ranging in length from 2 to 8 and operate on narrow 8-bit or 16-bit data. The vector extensions were introduced to exploit the parallelism in multimedia applications more effectively than superscalar hardware does.

These vector extensions are a good first step towards incorporating a vector architecture at the microprocessor level. However, almost no research has been published on how to best use vector architectures for multimedia applications [21]. One possible reason for the lack of results could be that, in addition to the usual simulation tools, a significant amount of compiler infrastructure is also required to carry out a comprehensive investigation on vector architectures, traditional ones or otherwise.

To determine whether such an extensive, in-depth study is justified, we have conducted an initial study on the performance and cost of vector processors. This paper presents the results of this study and previews the benefits in both performance and cost that we expect to see. For our study, we compared the performance of vector microprocessors to that of aggressive, out-of-order superscalar processors. We are using superscalar processors as the baseline because both vector and superscalar processors are designed to improve performance by exploiting parallelism. Comparing these two architectural styles is more insightful than using a scalar-only processor as the baseline. We also examined the cost of achieving vector performance to further motivate why a more comprehensive study would be of interest.

The remainder of the paper is organized as follows. Section 2 describes the superscalar and vector processors that we used for this study. Section 3 presents the performance results for a highly parallel, integer application while Section 4 examines the implementation cost of providing this vector performance. Finally, Section 5 summarizes the main findings of the study and concludes with a description of future work.

2 Processor Configurations

Figure 1 lists the processor configurations that we used for our study. Both the superscalar and vector processors use the MIPS instruction-set architecture. All the processors are configured with integer and load/store units. This is because, for this initial study, we are interested in exploiting the parallelism in applications that use integer data only. Thus the availability of floating-point units would have no effect on the performance results.

The three superscalar configurations—SSsmall, SSbig, and SShuge—are based on the MIPS R10000 [32]. In addition to issuing instructions out of order, these superscalar processors allow loads to dynamically by-pass stores as long as no memory conflicts occur. Also, a store can issue as soon as its address is known even if the data to store is not ready yet, thus avoiding long dependency chains involving loads and stores [31].

SSsmall is essentially the R10000 with the following exceptions. The two floating-point units have been replaced with two more integer units and one more load/store unit has been included. To minimize the performance impact of fetching instructions, the fetch width has been increased from 4 instructions to 16 instructions. Finally, although the ALUs in the R10000 are functionally asymmetric, we use homogenous integer units in our simulations. These integer units are also comparable in functionality to those in the Alpha 21264 [9].

SSbig is a larger version of SSsmall in that the issue width has been widened from 4 to 6. To increase the chances that 6 instructions will issue per cycle, the other hardware structures have also been augmented. For example, the re-order buffer is enlarged from 48 entries to 128 entries and enough physical registers are provided so as to eliminate any non-true register dependences via register renaming. The number of integer units and load/store units has also been increased to 16 and 8, respectively. Although SSbig is extremely aggressive by today's standards, the introduction of the Alpha 21264 suggests that future superscalar implementations are moving towards such dimensions [9].

SShuge is an extremely large and impractical superscalar configuration. We use this processor in a simulation study to demonstrate that the vector processors listed in Figure 1 are able to exploit significantly more parallelism than superscalar processors can to improve performance.

The vector configurations—Vsmall, Vmed, and Vbig—are based on the Torrent-0 (T0) microprocessor which extends the MIPS instruction set to include vector instructions for fixed-point arithmetic [29, 2]. A Torrent vector instruction executes up to 32 operations, and T0's vector register file contains 16 vector registers, each of which consists of 32 32-bit registers. The functional units on T0 include a scalar arithmetic unit, two vector arithmetic units each of which contain 8 ALUs, and a vector memory unit which handles both vector and scalar memory instructions. Although there is a 1KB on-chip instruction cache, data accesses go off chip to 8MB of SRAM.

T0's vector hardware is organized as 8 parallel lanes, with each lane holding one pipeline for each of the three vector units. The 8 lanes are clearly visible as eight vertical strips in T0's die photo [29]. In addition, the 32 elements of each vector register is striped across all 8 lanes.

Up to 8 operations per vector computational instruction can be executed each cycle and at most 4 cycles are needed to execute such instructions. Because the memory unit connects to a 128-bit data bus, a maximum of 4 32-bit *unit*-stride accesses can occur per cycle. However, because there is only a 32-bit address bus, non-unit stride and indexed vector memory instructions proceed at one access per cycle. Thus vector memory instructions require up to 8 or 32 cycles to execute depending upon the access pattern.

Unlike superscalar implementations, T0 issues at most one instruction per cycle to either the scalar unit or one of the three vector units. With the appropriate mix of instructions, nonetheless, it is possible to saturate T0's functional units by issuing a vector instruction each cycle and having the scalar unit issue an instruction every fourth cycle. In all, T0 can execute up to 21 32-bit operations in one cycle.

To sustain this execution rate, the vector register file must be able to support either 32 reads and 20 writes per cycle if a vector load and two vector computational instructions are being executed or 36 reads and 16 writes per cycle for a vector store and two com-

Component	Superscalar Processors (based on MIPS R10000)			Vector Processors (based on Torrent-0)		
	SSSMALL	SSBIG	SSHUGE	VSMALL	VMED	VBIG
ISA	MIPS			MIPS with fixed-point vector extensions		
fetch width	16 instructions	16 instructions	128 instructions	1 instruction		
issue width	4 instructions	6 instructions	24 instructions	1 instruction		
re-order buffer size	48 instructions	128 instructions	768 instructions	——		
BTB size	512 2-bit entries	2048 2-bit entries	16K 2-bit entries	——		
# physical registers	64 integer 64 floating-point (128 total)	160 integer 160 floating-point (320 total)	800 integer 800 floating-point (1600 total)	32 scalar 16 32-element vector (544 total)		
# integer units	4	16	16	4	8	16
# load/store units	2	8	8	2	4	8
memory system	ideal			ideal		
C compiler	SGI –O2 (Version 5.3)			vector-enhanced SUIF (Version 1.1.0)		

Figure 1: Simulation Parameters for Superscalar and Vector Processors

putational instructions. As part of the discussion on the cost of implementing such a highly parallel vector processor, we explain in Section 4 how relatively little die area is needed to provide such tremendous register bandwidth.

Of the vector processors we simulated, VMED is the closest in structure to T0 with the following changes. Homogenous integer units are used in both vector arithmetic units and each integer unit is capable of executing a full 32-bit multiply using an implementation similar to that in the R10000. The total number of execution units is halved from 16 to 8 and the vector registers are re-organized to be striped across 4 lanes. Finally, the load/store units are similar to those in the superscalar configurations in that a separate address port is associated with each data access. Thus up to 4 non-unit strided accesses or 4 indexed vector accesses could be executed per cycle. As we will show in Section 4, given current VLSI trends, a vector processor with on-chip memory is viable now or will be in the near future, and thus the availability of multiple address ports is a reasonable assumption.

VSMALL is a smaller version of VMED in that each vector arithmetic unit contains 2 integer units rather than 4 and the vector registers are striped across 2 lanes. In addition, the data bus has been halved to 64 bits. VBIG is a larger version of VMED: each vector unit contains 8 integer units as in the T0 and twice as many data accesses can occur in parallel. The functional units for VSMALL and VBIG have been chosen to match those of their superscalar counterparts, SSSMALL and SSBIG, respectively.

The simulations for all processor configurations use an ideal memory system, and all processors use R10000 latencies and repeat rates [32]. Finally, a superscalar executable was generated using SGI's Version 5.3 C compiler invoked with the –O2 flag while a vector executable was generated using a vectorizing compiler that we developed [6]. The vectorizing compiler is based on the SUIF 1.1.0 compiler infrastructure [30] with the *machsuif* 1.1.0 extensions for specialized code generation and machine-dependent optimizations [25]. Similar to a superscalar executable, once a vector executable is created, it is binary compatible across a wide configuration of vector processors.

3 Performance of PGP Decryption

As an initial step to verify that our simulation environment was functioning correctly, we began our study with a small set of parallel loops. To demonstrate the broader applicability of vector processing, we chose loops that are not from scientific nor engineering applications and that use integer data rather than floating-point data. Because the initial set of loops are parallel, we expected that both vector and superscalar processors would perform equally well. As the results in this section show, however, we instead discovered that superscalar processors are *unable* to exploit the parallelism in a highly parallel application to the same extent that vector processors can. By analyzing and fully understanding these initial results, we expect to apply this knowledge to produce similar performance results on a broader range of larger programs.

For space reasons, we present the results for a single application: PGP decryption. Results for the full suite can be found in [6].

3.1 PGP Decryption Code

PGP (Pretty Good Privacy) is a public-key encryption software package that was published as freeware in 1991 by Philip Zimmerman [19]. In addition to satisfying the general criteria of being non-scientific, non-floating-point but highly parallel, PGP was included in our initial study because interest in encryption algorithms for security reasons is increasing with the explosive growth in Internet use and the PGP algorithm is one of the more popular ones.

For our study, we examined the performance of decrypting an 8,000 block message, where one block is 16 bits. The bulk of time for PGP decryption is spent in a double-nested loop where an 8-block key is applied to 8-block chunks of the message at a time [17]. The inner loop is executed 8 times as it iterates over each block in a chunk. The decryption of a block is highly dependent on the decryption of the previous block. These dependences form several long dependence cycles within the inner loop. The outer loop iterates over the chunks in a message. Decrypting a chunk does not depend on the decryption of any other chunk. Because iterations of the outer loop are completely independent of each other while those of the inner loop are not, we expect that most of the parallelism to exploit will be among operations of different iterations of the outer loop.

Although the superscalar processors can, in theory, exploit the parallelism in both the inner and outer loops, the vector processor is unable to exploit any of the parallelism in the original version of the loop. So that the vector hardware could be used, we rewrote the loop in two different ways.

The `unrolled` version completely unrolls the inner loop so that the remaining loop can be vectorized. The `interchanged` version interchanges the two loops so that the iterations of the transformed inner loop are independent of each other and the long dependence cycles exist between iterations of the transformed outer loop. The `interchanged` version essentially moves the parallelism into its inner loop. To make the interchanged loop functionally equivalent to the original, four scalar variables are expanded into temporary arrays whose length is equal to the length of the message being decrypted.

It is possible to perform these transformations automatically in the compiler. However, we have not yet incorporated such features into our research compiler and so have made these transformations at the source-code level. By doing so, we are also able to conduct a fairer performance comparison. The source codes for both versions are given to the compilers of the vector and superscalar processors, and hence the benefits of both transformations are made available to all the processor configurations that we simulated.

We looked at superscalar performance on the original, rolled loop and found it to be comparable to that of the `unrolled` version. This is not surprising since an out-of-order superscalar processor essentially unrolls a loop dynamically.

We also simulated a `handcoded` version for the vector processor to see how much additional performance we could obtain with a better vector compiler. The `handcoded` version is similar to the `interchanged` version except that the introduced temporary arrays are allocated to vector registers rather than memory when the inner loop is stripmined.

3.2 Performance Results

Figure 2 shows the simulation environment used to generate our performance data. The same C source code is input to the two compilers.

Superscalar simulation proceeds by executing the compiled binary using `pixie` and feeding the resulting execution trace into `cello`, a reconfigurable timing simulator developed by Todd Mowry while he was at the University of Toronto. `Cello` supports the simulation of a broad set of superscalar processors ranging from simple in-order scalar processors to wide out-of-order superscalars with aggressive scheduling and branch prediction.

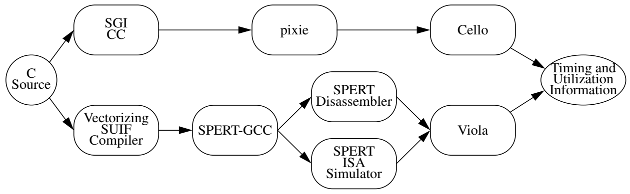


Figure 2: Simulation Environment

Vector simulation proceeds by executing the compiled binary on an ISA simulator that is part of the SPERT development package provided by Torrent's designers. The ISA simulator generates an execution trace which is then fed into `viola`, a reconfigurable timing simulator developed for this performance study. Although hardware parameters, such as the maximum vector length and the number of vector registers, are fixed, the number of computational and memory vector units as well as the number of lanes per vector unit are fully configurable, as is the latencies of each operation. `Viola` can also optionally model hardware containing full 32-bit integer multipliers and a memory system that supports strided accesses at the same rate as unit-stride accesses.

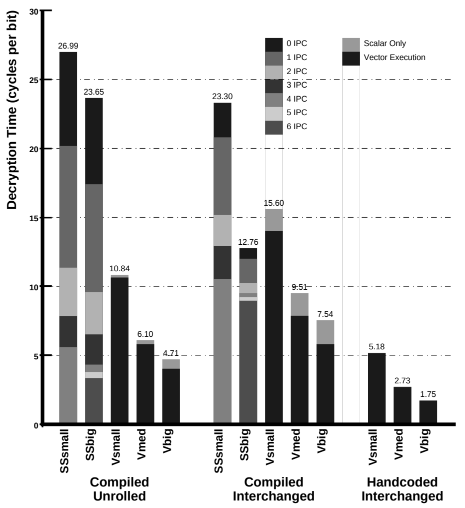
More details of the simulation environment can be found in [6].

3.2.1 Vector Processors Exploit Significantly More Parallelism

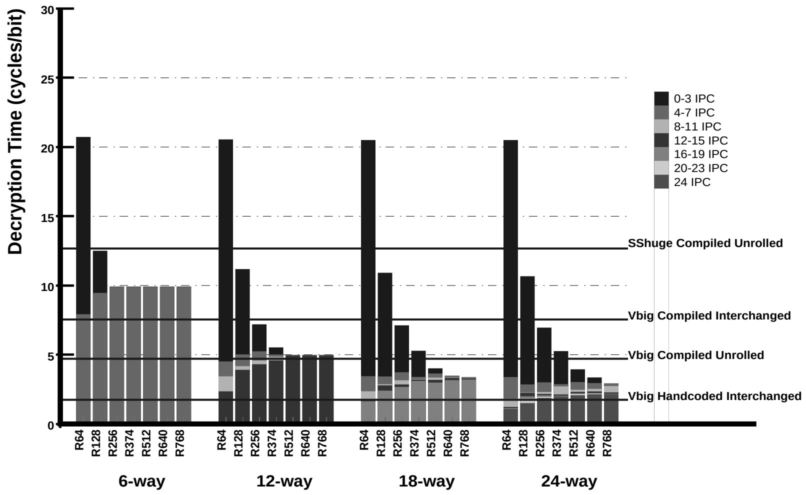
Figure 3(a) shows how quickly the various processors can decrypt an 8,000-block PGP-encrypted message when using the different loop versions. Performance is given as the number of cycles per bit in the message. A bar showing superscalar performance is shaded to show the percentage of time during which 0 to 6 instructions were graduated per cycle. The uppermost shaded part corresponds to 0 instructions being graduated, while the lowermost portion corresponds to the maximum number that a particular configuration can graduate. A bar showing vector performance is divided into two parts. The uppermost gray-shaded part indicates the percentage of time spent executing only scalar instructions. The lower dark-shaded portion represents the time spent executing one or more vector instructions.

The relative performances of the three loop versions for a vector processor indicates that greater performance is possible if we were to add more optimizations to our vectorizing compiler. This is not surprising since our vectorizing compiler is a research prototype. We have put little work in code optimization and, instead, have concentrated on vector code generation itself. In a related study, we have examined the code emitted from our compiler to determine where improvements can be made [6].

Despite the unoptimized code being generated by



(a) Superscalar & Vector Performance



(b) Unrealistic Superscalar Performance

Figure 3: Performance of PGP Decryption

our vectorizing compiler, the vector processors always decrypt faster than any of the superscalar processors in the study regardless of which version is used. The one exception is VSMALL executing the `interchanged` version. Vector processors are faster even when executing with comparable configurations of functional units or even fewer units.

The reason why the superscalar processors are slower than the vector processors is because the vector processors are exploiting *significantly more* parallelism. To prove this, we simulated increasingly aggressive (to the point of being unrealistic) superscalar implementations to find superscalar configurations that execute as fast as the vector processors execute. We simulated 6-way, 12-way, 18-way, and 24-way processors with re-order buffers that are multiples of 128 entries as well as ones with a 64-entry re-order buffer. The functional unit configuration is the same as SSBIG's and VBIG's. SSHUGE is the largest configuration we simulated: a 24-way superscalar with a 768-entry re-order buffer.

Other implementation parameters were set to sizes that would preclude them from being impediments to parallel execution. For example, the instruction fetch width is set to 128 instructions. The BTB is increased to 16K 2-bit entries, and the number of renaming registers is increased proportionally with the re-order buffer size.

Figure 3(b) shows the performance results of our simulations. The vertical bars are the superscalar decryption times when using the `interchanged` loop version. The bars are shaded to show the proportion of time spent graduating 0 to 24 instructions per cycle. The lower three horizontal lines give VBIG's decryption times for the three versions of the algorithm. The topmost horizontal line is the performance of SSHUGE executing the `unrolled` version.

Only superscalar configurations substantially larger than SSBIG *and* executing the `interchanged` version can exploit the available parallelism as effectively as VBIG. A 12-way issue superscalar with a 256-entry re-order buffer is the *smallest* configuration that executes faster than VBIG. Much larger superscalars are needed to achieve performance comparable to that of VBIG executing the faster versions of the loop. An 18-way, 512-entry superscalar executing the `interchanged` version exploits enough parallelism to decrypt as fast as VBIG executing the `unrolled` version. The largest configuration we simulated, SSHUGE, is still 1.6 times *slower* than `handcoded` vector performance.

Although we used the two loop versions so that our compiler could generate vectorized code, Figure 3 shows that superscalar execution also benefits greatly, if not more, from these vectorizable versions of the loop. SSHUGE provides a nearly two-fold performance improvement over SSBIG on the `unrolled` version. However, SSHUGE executing the `unrolled` version is still no faster than SSBIG executing the `interchanged` version.

A perusal of the generated code for both versions revealed that the code was well-optimized and, moreover, contained no code for register spills. Instead the reason superscalar execution performs relatively poorly with the `unrolled` version is because the parallelism is too far apart. As explained earlier, there is little parallelism within the body of the unrolled loop due to the long dependence cycles in the original inner loop, while the abundance of parallelism exists in the original outer loop. The lack of abundant parallelism in the unrolled loop and the inability to exploit the parallelism across the loop iterations is indicated by the low IPC of 3.55 achieved by the 24-way SSHUGE processor. Because the unrolled loop contains 675 instructions, independent instructions are executed extremely far apart in time. A re-order buffer substantially larger than 768 entries would be needed for a superscalar processor to improve its `unrolled` performance by another factor of 5 or 6.

Superscalar processors execute the `interchanged` version faster because this version *temporally localizes* the parallelism by moving it from the original outer loop to the transformed inner loop. An iteration of the inner loop now contains about 120 instructions, and thus, independent instructions from successive inner-loop iterations can be detected in a re-order buffer with 768 entries or less. SSHUGE's high IPC of 20.27 is evidence that parallelism is being exploited. The reason why the superscalar processors are still slower than vector processors, even when executing the `interchanged` version, is explained in the next section.

Vector processors execute *both* loop versions fast because they use vector instructions to exploit parallelism *across* loop iterations rather than within a loop iteration. Although the loop-level parallelism in the `unrolled` version is temporally distant when executed sequentially, transforming the code into vector instructions re-schedules the operations so that operations from different iterations are now executed together in a vector instruction. Note that only a vectorizable version of a loop is needed to exploit parallelism because vectorization itself implicitly reschedules the parallelism so that it can be exploited effectively. Hence, performance can be improved by executing either the `unrolled` or `interchanged` version on a vector processor. Of course, other code optimizations could still be applied to improve performance further.

3.2.2 Performance Impact of Reducing Instruction Bandwidth

The interchanging transformation and the use of vector instructions are techniques that statically schedule parallelism so that substantially less hardware can be used to exploit the parallelism. In a similar fashion, additional compiler optimizations such as software pipelining could be applied to the superscalar code to reduce the size of the re-order buffer further. Such optimizations would essentially emulate the scheduling and register allocation that vector code translation does implicitly.

We did not conduct a study that used these more sophisticated compilation techniques for superscalar execution, in part, because we did not have the time nor resources to do so. Instead, we analyzed our existing simulation data to establish lower bounds on decryption time for statically scheduled superscalar execution or dynamically scheduled superscalars with unrealistically enormous re-order buffers. Our analysis of these lower bounds shows that the ability to detect parallelism is only partially responsible for the vector processors' performance. The other reason for the better performance is the substantially reduced instruction bandwidth due to using vector instructions.

A lower bound on decryption time is I/N where I is the number of instructions executed per bit and N is the issue width of the superscalar processor. No matter how well code is scheduled and regardless of the hardware configuration of an N-way superscalar processor, the fastest such a processor can issue or graduate instructions is N per cycle. Hence, if the lower bound for an N-way superscalar is slower than any vector times, it is unlikely that such a processor will be able to out-perform the vector processor.

Figure 4(a) shows the lower bounds for issue widths varying from 4 up to 12. For the `unrolled` version, the superscalar processors execute 44.98 instructions for each bit. Thus, an N-way superscalar would require at least $44.98/N$ cycles to decrypt one bit. Similarly, the fastest an N-way superscalar could decrypt using the `interchanged` version is $59.43/N$ cycles per bit. For the sake of comparison, the decryption times for the vector processors are also shown.

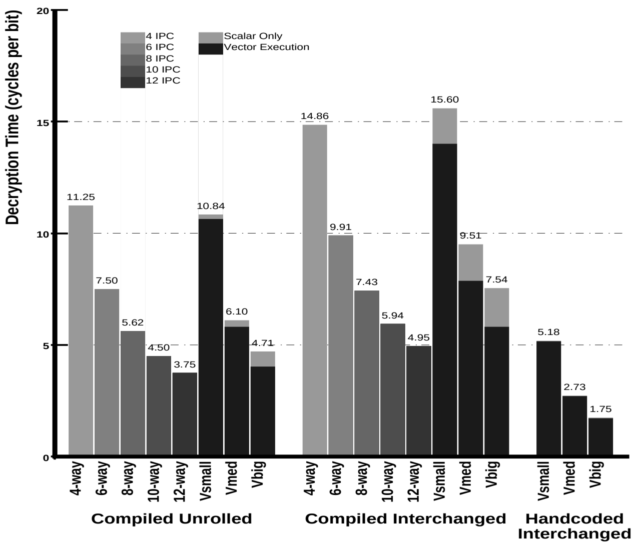
With more sophisticated scheduling, a 4- or 6-way superscalar processor could possibly decrypt almost as fast or as fast as its vector counterpart executing the *unoptimized* compiled vector code for the same source program. A superscalar as wide as 10-issue would be needed, however, to approach the performance of even the smallest vector processor executing the `handcoded` version. Superscalar processors significantly wider than 12-way are needed to achieve performance comparable to optimized VMED or VBIG execution. Even under the assumption that a 12-way superscalar can exploit as much paralellism as it can, such a processor would still be more than 2 times *slower* than VBIG executing the `handcoded` version.

Figure 4(b), which gives the average number of instructions executed to decrypt one bit, shows why superscalar processors are less effective. A superscalar processor must issue and execute over an *order of magnitude* more instructions than a vector processor when executing this highly parallel loop. Although there is an abundance of parallelism, superscalar processors are unable to exploit it *fully*. The fundamental problem in exploiting highly parallel code for superscalar processors is that they execute *one instruction for each operation*. In constrast, vector processors use vector instructions that execute many operations each. For the vector processors that we simulated, each vector instruction can execute up to 32 operations and thus reduce the dynamic instruction count by more than an order of magnitude.

4 Implementation Cost

In the previous section, we demonstrated that vector processors can exploit an abundance of parallelism more effectively than aggressive, state-of-the-art superscalar processors. To achieve this performance, the vector processors use a large register file with tremendous bandwidth. Moreover, the largest of the vector configurations, VBIG, contains 16 integer units and 8 load/store units. One would expect the costs of implementing such resources to be prohibitively high. In this section, we quantify these costs and, by comparing them to the implementation costs of several state-of-the-art commercial superscalar microprocessors, we show that the cost of a highly parallel vector microprocessor is no more than that of a superscalar one.

We use die area as a measure of cost. Although die area is not as scarce a resource as it once was, there are several benefits to achieving higher performance with the same die area or, alternatively, using smaller die without sacrificing performance. One benefit is that



(a) Lower Bounds on Superscalar Decryption Times

Loop Version	Processor Type	
	superscalar	vector
Compiled Unrolled	44.98	2.51
Compiled Interchanged	55.43	3.48
Handcoded Interchanged	—	0.96

(b) #Instructions Executed

Figure 4: Performance Impact of Reducing Instruction Bandwidth

die area greatly affects the actual cost of a die [13]. Another benefit is that a smaller die facilitates distributing the global clock signal and controlling clock skew since signals travel a shorter distance. This is particularly helpful when dealing with higher clock frequencies where clock distribution and clock skew become more problematic. Also, a smaller die uses less power, a desirable feature for high-performance microprocessors used in laptop computers and if they are to be considered for use in embedded applications.

A comparison of the die areas of different implementations, however, can be complex. Differences in area may be due to many factors that are orthogonal to how a processor exploits parallelism. These factors include fabrication technology, clock frequency, datapath width, and circuit and layout design. In the following, we attempt to adjust the areas to account for these differences so that remaining area differences will be due to parallel-specific features such as datapath configuration, register organization, and instruction-issue mechanism.

Figure 5 lists die areas as well as clock frequencies and process technologies for actual microprocessor implementations and for SSBIG. The processors are grouped into three categories based on the mechanism for issuing instructions: vector, in-order superscalar, and out-of-order (OOO) superscalar. The vector microprocessor, Torrent-0, is fully functional at 45 MHz and is fabricated with Hewlett-Packard's CMOS26G process using 1.0μm scalable CMOS design rules and two metal layers [29]. The clock frequencies of the superscalar microprocessors range from 200 to 600 MHz, and the implementations were fabricated with 0.5μm or 0.35μm CMOS technology and 4–6 metal layers. The die areas for the actual implementations were obtained by measuring components in annotated die photomicrographs and floorplans [3, 11, 12, 20, 2, 10, 9]. The data for SSBIG are based on Olukoton *et al.*'s study and the areas are extrapolations from the R10000 implementation scaled to 0.25μm technology [23].

Three types of die area are given: the area of the entire die in the processor's implementation technology, the die area used for just the processor components, and the processor die area when scaled to a 0.25μm process. Excluded from processor die area are cache and TLB structures, external interface logic, and the pad ring. Area for processor clocking and overhead is scaled relative to the processor portion of the entire die. Although we consider only the processor portion for this initial study, we intend to examine the cost and performance of appropriate memory structures for vector microprocessors in the future.

The processor die areas are scaled to a 0.25μm process to provide a common basis for comparing cost. A line width of 0.25μm was chosen because it should be possible to implement chips in such a process within two years [4]. Unless stated otherwise, die areas in the remainder of this paper are given for a 0.25μm process technology.

An immediate observation of Figure 5 is that the vector implementation requires the least amount of area when all implementations are scaled to the same process technology. T0's small size is even more impressive when one considers that T0 contains from 2 to 4 times as many execution units as the superscalar implementations and the same number as SSBIG. Moreover, T0's die may be even smaller if it were redesigned using 4 or more metal layers rather than 2.

Part of the difference in area can be attributed to T0's much slower clock frequency. T0's clock rate is 6 to 13 times slower than the rate of the superscalar

Microprocessor	Issue Mechanism	Clock Frequency	Line Size	Total Die Area	Processor Die Area	Processor Die Area Scaled to 0.25μm
Torrent-0	single-issue vector	45 MHz	1.0 μm	280.56 mm^2	240.25 mm^2	14.73 mm^2
Alpha 21164	4-way in-order	300 MHz	0.5 μm	298.65 mm^2	87.43 mm^2	21.86 mm^2
Alpha 21164	4-way in-order	433 MHz	0.35 μm	208.80 mm^2	53.93 mm^2	27.51 mm^2
UltraSPARC II	4-way in-order	330 MHz	0.35 μm	156.25 mm^2	74.04 mm^2	37.77 mm^2
MIPS R10000	4-way OOO	200 MHz	0.35 μm	297.14 mm^2	131.17 mm^2	66.92 mm^2
HP PA-8000	4-way OOO	250 MHz	0.5 μm	319.57 mm^2	271.07 mm^2	67.77 mm^2
Alpha 21264	6-way OOO	600 MHz	0.35 μm	313.96 mm^2	136.82 mm^2	69.81 mm^2
SSBIG	6-way OOO	500 MHz	0.25 μm	457.00 mm^2	250.00 mm^2	250.00 mm^2

Figure 5: Die Areas of Superscalar and Vector Processors

implementations even though its entire (unscaled) die area is comparable to the entire (unscaled) die areas of the superscalar processors. The main reason for the slower rate is T0's coarser process technology: 1.0μm with two metal layers. The 45MHz clock rate is actually competitive with full-custom, commercial processors implemented in similar processes.

Implementing T0 in a 0.25μm technology would reduce the maximum distance that a signal would travel by a factor of 4 and thus increase the clock rate to 180 MHz without having to substantially re-design any circuits. However, this clock rate is still 1.7 to 3.3 times slower than the clock rates of the superscalar implementations. Although critical-path circuits could be enlarged in order to implement T0 with a faster clock rate in a 0.25μm process, we assume for the purposes of this comparison that a re-design with 4 or more metal layers would result in circuits that can operate at a higher frequency without a substantial area penalty.

4.1 Breakdown of Die Areas

To account for other area differences and to separate differences due to features related to ILP and those that are not, we analyze how much of the die is used to implement individual components of each processor. Figure 6(a) shows the areas for datapath, registers, and control for each processor die. The datapath includes integer units, floating-point units, and load/store units. Control consists of instruction fetch, decode, renaming, and issue logic. "Other" is the area for clocking and overhead.

4.1.1 Datapath

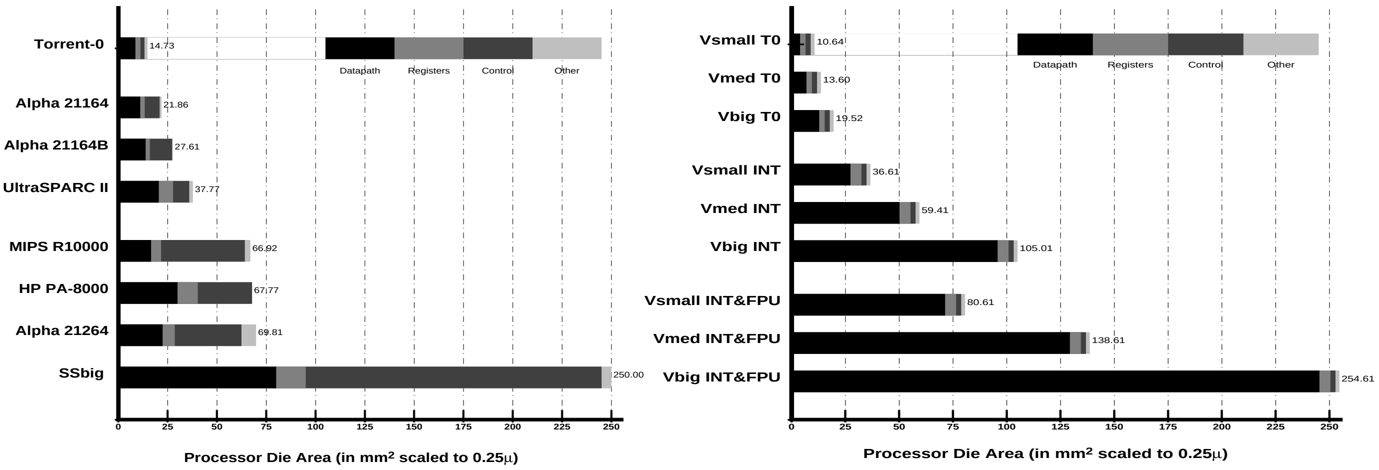
One observation from Figure 6(a) is that the area for T0's datapath is about 30% smaller than the smallest superscalar datapath. Part of the difference in datapath area is due to differences in the configurations of the execution units. T0 is a 32-bit processor with only integer units but has 16 of these. In contrast, the superscalar processors implement 64-bit datapaths that include floating-point units but typically have only 4 to 9 execution units in total.

Upon measuring the areas for individual ALUs among the various implementations, we find that a T0 ALU is approximately 10 times *smaller* than a superscalar ALU that executes comparable operations. The difference in data widths accounts for part of the difference in ALU area: shifter and multiplier size as well as input/output bussing can grow as the square of the data width, giving at most a factor of four difference in area due to bit width. The remaining factor of 2.5 difference in area can be attributed to the differences in clock frequency, as explained above, and to support for superscalar execution. The superscalar integer units include hardware for bussing and bypassing data across several functional units, whereas there is no such communication among the integer units in the T0 vector hardware.

4.1.2 Registers

Another observation from Figure 6(a) is that the area for Torrent's vector registers is either the same or several times smaller than the space needed for superscalar registers even though T0's vector register file contains 512 registers and must provide sufficient bandwidth to allow 20 operations to execute in parallel. In contrast, existing superscalar implementations provide 128–150 registers split across two or three register files that provide a combined bandwidth ranging from a minimum of 9 reads and 6 writes up to a maximum of 14 reads and 7 writes. Even after doubling the area to adjust for bit width, the vector register implementation would occupy approximately the same area as the register files in the larger superscalar implementations but provide more than 3 times the storage capacity and more than 3 times the bandwidth.

A further observation is that, even among the superscalar register files, there is a large variation in areas. Much of the variation is due to differences in register capacity and bandwidth. In a straightforward



(a) Areas for Actual Implementations & SSbig

(b) Estimated Areas for Vector Implementations

Figure 6: Breakdown of Processor Die Areas

design for a superscalar register file, both greater capacity and higher bandwidth result in larger areas. For example, the Alpha 21264 registers occupy close to 3 times more area than the 21164 registers in order to provide almost twice the capacity and twice the write bandwidth. Similar trends also exist among the other superscalar register files.

Differences in capacity, bandwidth of individual registers, and access speed mostly explain why the vector register file uses less space while providing more capacity. These, however, do not fully explain how T0's vector register file can provide 3 to 4 times more *overall* bandwidth in comparable area. T0's tremendous overall bandwidth is provided by the organization of its register file. Recall that the vector register file is not only partioned into 16 vector registers but also striped across 8 lanes. Vector registers are laid adjacent to each other in the Y-direction, while lanes of vector registers are adjacent in the X-direction. Thus, one select line can be used to place 8 values from the same vector register onto their respective data lines and 8 times more bandwidth is provided with a minimal increase in area.

In addition, each vector register has 5 read ports and 3 write ports, each of which is connected to a common data line that runs through a lane of the register file. Thus each lane can perform 5 reads and 3 writes simultaneously, for a total of 40 reads and 24 writes in one clock cycle for the entire register file. This is sufficient bandwidth to support the execution of 2 vector computational instructions and 1 vector memory instruction. Note that, unlike other vector processors such as the Cray Y-MP which use only dual-ported registers, T0's register design can sustain its peak bandwidth even if all three vector instructions are reading and/or writing out of the same vector register. Because vector registers are striped across the lanes, sufficient register bandwidth can be automatically provided without altering the total occupied area whenever the number of lanes is changed.

The technique of partitioning registers to achieve greater bandwidth without increasing area is already used to a lesser extent in the superscalar implementations. Integer register files have always been separate from floating-point register files, although increased bandwidth was not the only reason for this partitioning. A more direct example is the design of the integer register file in the Alpha 21264. The register file and datapath are partitioned into two clusters because a unified design would require additional operand buses and register read ports that would increase the area by 22% and make critical paths longer than the target cycle time [9]. The tradeoff is an extra cycle of latency for transfering data between clusters. The extra cycle could potentially be eliminated by clever instruction scheduling and register assignment algorithms.

4.1.3 Control

Returning to Figure 6(a), the final observation we note is that the area occupied by control logic is substantially larger in the superscalar implementations, both in terms of absolute area and as a proportion of the processor die. The reasons for this are the hardware structures needed to support issuing multiple instructions per cycle and, in the case of OOO implementations, dynamic scheduling.

A concrete example of how out-of-order issue adds to implementation cost is provided by the 56-entry re-order buffer of the HP PA-8000. The HP reorder buffer consists of 850,000 transistors and includes 6,720 comparators to determine when instructions can issue [8]. In comparison, the entire Torrent-0 chip contains 730,000 transistors and requires only 23 comparators to determine when an instruction can issue [1]. When scaled to the same technology, the HP re-order buffer occupies only slightly less area than the entire T0 chip including the pad ring.

Another example illustrating the cost of dynamic scheduling is the control logic for SSBIG, which is more than 3.5 times larger than the control for the other OOO superscalars. This is because SSBIG's 128-entry re-order buffer and renaming logic for 320 registers are implemented as single physical entities. In contrast, most superscalar implementations use several instruction queues and register files separated by functionality to mitigate quadratic increases in area when lengthening the instruction window.

Although in-order superscalar implementations exclude the hardware structures for dynamic scheduling, they still require significantly more area for control than the vector processor, both proportionally and absolutely. This is because in-order superscalars must still include hardware to check for independent instructions and to rename registers.

The reason why vector processors do not have these hardware structures is because much of the functionality for detecting parallelism has been transferred to the compiler. Moreover, the compiler is able to express the parallelism that it detects to the hardware in the compact form of vector instructions. These, in turn, allow the fetching, decoding, and issuing of a vector instruction to be only slightly more complex than that of a scalar instruction. As well, the encoding of a vector instruction is no more complex than the encoding of a scalar instruction, despite being more complex in functionality.

The savings in area due to this compiler reliance will be even greater as more parallelism is supported in the hardware. This is because, although the hardware structures for multi-way issue and dynamic scheduling will grow quadratically in area for a linear increase in issue width [23], the hardware structures for controlling instruction issue in a vector processor are expected to increase only minimally even as hardware is added in the form of additional lanes to support more parallelism. The control logic needed to determine whether a vector instruction can issue must track interlocks and manage chaining. These functions, in turn, depend only on the number of vector units and their pipeline latencies and not on the number of lanes. This is because there are no inter-lane dependences to consider during issue.

The only place in the datapath where there is interlane interaction is the memory crossbar. Thus the only portion of the control that would increase quadratically with the number of lanes is the logic that generates the control signals for the memory crossbar. However, any quadratic growth due to increasing beyond 8 lanes would not result in a substantial increase in overall area because the memory-crossbar logic occupies less than 5% of the control logic, which in turn is a small portion of the processor die.

4.2 Vector Die Areas

Based on our above analysis, we now estimate the processor die areas for VSMALL, VMED, and VBIG. We remove area differences due to features unrelated to vector or superscalar execution by assuming 64-bit datapaths and comparable execution units. For the sake of comparison, we use three types of execution units.

Figure 7 shows the areas of the components we used. The areas listed in the second column are based on the T0 implementation, while the areas in the third and fourth columns are a mixture of T0 components and 21264 execution units. For the 64-bit integer unit, we use the 21264 ALU in cluster CL1 which multiplies, while the 64-bit floating-point unit is based on the entire 21264 FPU which performs addition, multiplication, division, and square root. For the 64-bit datapaths, the area for a load/store unit is four times that of a T0 unit to account for both the wider data width and the increase in address ports. The area for the 64-bit vector register file is also double that of the 32-bit T0 vector registers. Area for global control, clocking, and overhead are assumed to remain unchanged.

Figure 6(b) shows the area breakdown and total area of the three vector configurations using the three types of execution units. In contrast to superscalar implementations, the vector implementations are dominated by datapath. VSMALL INT is about the same size as a 4-way issue, in-order superscalar processor, supports approximately the same amount of parallelism, but is able to exploit highly parallel code more effectively. VMED INT occupies about the same area as the out-of-order superscalars but supports about twice as much parallelism, and consequently achieves greater performance on highly parallel code. VBIG INT's datapath occupies slightly more area than SSBIG's but its control requires more than an order of magnitude *less* space. As a result, VBIG INT is about 2.5 times *smaller* than SSBIG while performing 2 to 5 times *faster* on highly parallel compiled code.

VSMALL INT&FPU, the least parallel vector processor that supports floating-point execution, would also be feasible to implement in today's technology. Based on the typical growth for die size over the past few years [4], the more parallel versions should be possible in the near future. Note that the areas of both VBIG INT&FPU and SSBIG are about the same but that more than 90% of the vector die is used for execution units while 60% of the superscalar die is occupied by control logic. Moreover, SSBIG contains only integer units whereas VBIG INT&FPU includes support for floating-point execution. Thus, a single-chip vector microprocessor makes more effective use of larger die and could conceivably provide supercomputing-level performance on scientific and engineering applications in the not-too-distant future.

5 Concluding Remarks

From the results of our initial study, we make several observations about the use of vector microprocessors for executing highly parallel code.

On the performance side, a vector microprocessor is able to exploit an abundance of vectorizable parallelism to improve performance more effectively than a superscalar microprocessor does when both utilize the same configuration of functional units and, in some cases, even when the vector processor implements fewer functional units. The vector processor's ability to exploit significantly more parallelism stems from the use of a vector instruction. A vector instruction exploits the parallelism *across* loop iterations rather than within a loop iteration. Generating vector

Component	Torrent-0 32b IUs	Alpha 21264 64b IUs	Alpha 21264 64b IUs & FPUs
scalar unit	0.96	4.54†	13.34†
vector execution unit (EU)	0.45	4.54†	13.34†
vector load/store unit (L/S)	0.58	2.32	2.32
vector register file	2.55	5.10	5.10
control	2.37	2.37	2.37
clock & overhead	1.80	1.80	1.80

Component areas marked with a † are based on execution units in the Alpha 21264. All other areas are based on Torrent-0 components.

Figure 7: Component Areas for Vector Processors

code implicitly re-schedules operations and *temporally localizes* parallelism so that the parallelism can be easily used by the hardware. A vector instruction also expresses an abundance of parallelism in a compact encoding that is no more complex than a scalar instruction. This, in turn, can reduce instruction bandwidth by more than an order of magnitude and allow up to 21 operations to execute in parallel without having to fetch, decode, and issue a comparable number of instructions.

Reducing instruction bandwidth is equally if not more important than being able to detect parallelism so that far more than 6 operations can execute in parallel. It is possible for a wide-issue superscalar processor to exploit as much parallelism as it can by either increasing the size of its hardware structures or relying on a compiler to find and schedule parallel instructions. However, a superscalar processor is still hindered by its inability to express parallelism in a compact form and consequently will leave a significant amount of parallelism unused once it begins issuing at its peak rate.

On the cost side, vector processors implement many functional units as well as an extremely large capacity, high-bandwidth register file. For example VBIG, the largest vector configuration we simulated, contains 16 integer units, 8 load/store units, and 512 registers that provide an overall bandwidth of 40 reads and 24 writes per cycle. Nonetheless, the implementations of integer-based vector processors are no more costly than the implementations of existing in-order and out-of-order superscalar processors. Based on current growth trends in VLSI technology, support for floating-point in a highly parallel vector microprocessor should be possible in the near future. The observation that vector processors are not costly to implement is also supported by the fact that most commercial microprocessors have included limited vector extensions for multimedia applications while incurring a less than 5% increase in VLSI area.

There are two reasons why vector processors are no more area-intensive than superscalar processors. One is that partitioning the register file into vector registers and lanes provides tremendous bandwidth without incurring a large area penalty. The second reason is that the control logic for issuing vector instructions is only slightly more complex than the control logic for scalar issue. In contrast, instruction control in superscalar processors is significantly more complex and thus requires substantially more area to implement. The difference in area for registers and control will grow even greater as additional hardware is implemented to support more parallelism. In short, the area for control and, to a lesser extent, registers is drastically reduced in a vector processor, leaving much of the die available to implement datapath functionality especially for implementations that support a high degree of parallelism.

The performance-boosting features as well as the area-saving features in vector processors are possible because much of the functionality of detecting and expressing parallelism is transferred to the compiler. The above observations suggest that the resultant tradeoff due to this reliance on a vectorizing compiler offers a tremendous cost/performance benefit.

These conclusions are based on the simulation results of one integer application assuming an ideal memory system. However, we believe that the gains in performance are sufficiently high, even for *unoptimized* vector code, and that the cost of implementation is sufficiently low that we are now conducting a more extensive performance study using large application programs to determine whether these performance results are typical and, if not, to investigate techniques to uncover any parallelism for the benefit of both vector and superscalar processors.

We also plan to examine vector performance in the context of a realistic memory interface and memory system. Although vector processors have a reputation for requiring high memory bandwidth, we expect vector performance to be no worse than superscalar performance. This is because vector execution has simple needs for instruction fetching in comparison to superscalar execution which must contend not only with fetching multiple instructions per cycle but also with aligning instructions especially across branches. In addition, vector memory instructions encode information about data access patterns that could be used to assist in providing high memory bandwidth. In a related project, researchers at the University of California at Berkeley are investigating the use of a vector processor to exploit the tremendous bandwidth potential in a merged processor-DRAM chip [24, 15]

Acknowledgements

We thank Krste Asanović for providing us with the SPERT distribution package and detailed information about Torrent T0. We also thank Todd Mowry for the use of his `cello` superscalar simulator. This work was supported in part by NSERC and ITRC, a Centre of Excellence funded by Technology Ontario.

References

[1] Krste Asanović. Private communication, 1996.

[2] Krste Asanović, James Beck, Bertrand Irissou, Brian E.D. Kingsbury, and John Wawrzynek. T0: A Single-Chip Vector Microprocessor with Reconfigurable Pipelines. In *Proceedings of the European Solid-State Circuits Conference*, volume 22, September 1996.

[3] William J. Bowhill et al. A 300MHz 64b Quad-Issue CMOS RISC Microprocessor. In *Digest of Technical Papers for the International Solid-State Circuits Conference*, volume 38, pages 182–183,362, San Fransisco, CA, February 1995.

[4] William F. Brinkman. The Transistor: 50 Glorious Years and Where We Are Going. In *Digest of Technical Papers for the International Solid-State Circuits Conference*, volume 40, pages 22–26, San Fransisco, CA, February 1997.

[5] 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.

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

[7] Kemal Ebcioğlu and Erik R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proceedings of the International Symposium on Computer Architecture*, volume 24, pages 26–37, Denver, CO, 1997.

[8] Neela Bhakta Gaddis, Joseph R. Bulter, Ashok Kumar, and William J. Queen. A 56-Entry Instruction Reorder Buffer. In *Digest of Technical Papers for the International Solid-State Circuits Conference*, volume 39, pages 212–213,447, San Fransisco, CA, February 1996.

[9] Bruce A. Gieseke et al. A 600MHz Superscalar RISC Microprocessor with Out-Of-Order Execution. In *Digest of Technical Papers for the International Solid-State Circuits Conference*, volume 40, pages 176–177,451, San Fransisco, CA, February 1997.

[10] David Greenhill et al. A 330MHz 4-Way Superscalar Microprocessor. In *Digest of Technical Papers for the International Solid-State Circuits Conference*, volume 40, pages 166–167,449, San Fransisco, CA, February 1997.

[11] Paul E. Gronowski et al. A 433MHz 64b Quad-Issue RISC Microprocessor. In *Digest of Technical Papers for the International Solid-State Circuits Conference*, volume 39, pages 222–223,449, San Fransisco, CA, February 1996.

[12] Silicon Graphics: MIPS Group. MIPS R10000 Microprocessor. *http://www.mips.com/products/r10k/T5_Die.html*, 1996.

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

[14] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. *http://www.digital.com/-semiconductor/a264up1/index.html*, Nov 1996. Slide presentation at Microprocessor Forum, Oct 22–23, 1996.

[15] Christoforos E. Kozyrakis et al. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

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

[17] Xuejia Lai. Detailed Description and a Software Implementation of the IPES Cipher. Signal and Information Processing Laboratory, November 1991.

[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] Jeff Licquia. Frequently Asked Questions for Pretty Good Privacy. *http://www.prairienet.org/pub/providers/pgp/-pgpfaq.txt*, May 1995.

[20] Jon Lotz, Gregg Lesartre, Samuel Naffziger, and Don Kipp. A Quad-Issue Out-of-Order RISC CPU. In *Digest of Technical Papers for the International Solid-State Circuits Conference*, volume 39, pages 210–211,446, San Fransisco, CA, February 1996.

[21] Trevor Mudge. Strategic Diretions in Computer Architecture. *ACM Computing Surveys*, 28(4):671–678, Dec 1996.

[22] Ravi Nair and Martin E. Hopkins. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups. In *Proceedings of the International Symposium on Computer Architecture*, volume 24, pages 13–25, Denver, CO, 1997.

[23] 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.

[24] David Patterson et al. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, March–April 1997.

[25] Michael D. Smith. Extending SUIF for Machine-dependent Optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford University, January 1996.

[26] J. G. Steffan and T. C. Mowry. The Potential for Thread-Level Data Speculation in Tightly-Coupled Multiprocessors. Technical Report CSRI-TR-350, Computer Systems Research Institute, University of Toronto, February 1997.

[27] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the International Symposium on Computer Architecture*, volume 23, pages 191–202, Philadelphia, PA, 1996.

[28] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the International Symposium on Computer Architecture*, volume 22, pages 392–403, Santa Margherita Ligure, Italy, 1995.

[29] John Wawrzynek, Krste Asanović, Brian Kingsbury, James Beck, David Johnson, and Nelson Morgan. SPERT-II: A Vector Microprocessor System. *IEEE Computer*, 29(3):79–86, March 1996.

[30] R. Wilson et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 29, pages 31–37, *http://suif.stanford.edu/suif/suif.html*, 1994.

[31] Kenneth C. Yeager. Private communication, 1996.

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