To be published in the *Proceedings of the* 31^{st} *Annual International Symposium on Microarchitecture, December 1998.* 1

Simple Vector Microprocessors for Multimedia Applications

Corinna G. Lee and Mark G. Stoodley
{corinna,stoodla}@eecg.toronto.edu
Department of Electrical and Computer Engineering
University of Toronto

Abstract

In anticipation of the emergence of multimedia applications as an important workload, microprocessor companies have augmented their instruction-set architectures with short vector extensions*, thus adding basic vector hardware to state-of-the-art superscalar processors. Although a vector architecture may be a good match for multimedia applications, there is growing evidence that the control logic for increasingly complex superscalar processors is difficult to implement.*

Rather than combining a complex superscalar core with short wide vector hardware, we propose using a much simpler processor design that is similar to traditional vector computers with long vectors and simple control logic for instruction issue. Such a design would use the bulk of its transistors and die area for datapath and registers, and thus lessen the time required to design, implement, and verify control.

In this paper, we present data that quantifies this trading of control transistors for datapath and register transistors. We demonstrate that a 2-way, in-order vector processor with a vector length of 64 and a vector width of 8 requires no more die area, and possibly significantly less area, than a 4-way, out-of-order superscalar processor with short vector extensions. Furthermore, we show that the simple long vector processor is, on average, 2.7 times faster executing multimedia applications than the superscalar processor, and 1.6 times faster than one with short vector extensions.

To explain the reasons for the higher performance, we analyze execution time in terms of dynamic operation *count and cycles per* operation *(CPO). A vector processor executes fewer operations by using vector instructions to stripmine a loop. Moreover, a long vector processor achieves a lower CPO by effectively using parallelism at both the operation and the instruction levels. Thus by reducing* both *terms of the CPO equation, the simple long vector processor achieves greater performance.*

1. Introduction

Advances in microprocessor design over the past decade have been primarily driven by two application domains: technical and scientific applications for uniprocessor desktops, and transaction processing and file-system workloads for multiprocessor servers. It is expected, however, that application domains will shift over the next two decades. Although it is difficult to predict what future applications will be, there is a growing consensus that multimedia applications will increase in importance as greater priority is given to more human-friendly interfaces and to personal mobile computing [7, 20].

In anticipation of this emerging applications area, microprocessor companies have augmented their instruction-set architectures with *short vector extensions*, thus adding basic vector hardware to state-of-the-art superscalar processors. Figure 1 lists the extensions that have been introduced or announced by all major microprocessor companies.

A common aspect of the vector extensions listed in Figure 1 is that all use a wide datapath that is partitioned to execute narrower data types in parallel. These narrower data types are more typical for multimedia applications which manipulate sound and image data. Almost all use a 64-bit datapath; the HP MAX-1 uses a 32-bit datapath while the PowerPC AltiVec uses a 128-bit datapath.

It is useful to characterize a vector implementation in terms of its *vector length* and *vector width*. Vector length is the maximum number of operations that a vector instruction can execute, while vector width refers to the number of operations that are executed in one clock cycle for a vector instruction.

Thus, each 64-bit vector extension can be viewed as an extremely short vector architecture with a vector length of 8 and a vector width of 8 for 8-bit data types. Wider data types are executed with even short vector lengths. Such vector configurations are quite different from the more typical vector lengths of 64 or 128 and vector widths of 1 or 2 which appear in vector supercomputers.

Although a vector architecture may be a good match for multimedia applications, there is mounting evidence that combining one with increasingly complex superscalar processors will be difficult to implement. Over the past 2 years, shippings of superscalar processors have been delayed repeatedly in order to meet target speeds [15, 14, 16]. Late shippings are often attributed to complex out-of-order designs. Promises of a huge transistor budget within a decade offer the possibility of implementing even more aggressive and complex de-

Processor	Short Vector Extension	Year	
Sun UltraSPARC	VIS: Visual Instruction Set [19]	1995	shipped
Hewlett-Packard PA-RISC	MAX-1: Multimedia Acceleration eXtensions [24]	1995	shipped
	MAX-2 [25]	1996	shipped
Silicon Graphics MIPS	MDMX: MIPS Digital Media eXtension [12]	1996	announced
Digital Alpha	MVI: Motion Video Instructions [5]	1996	announced
Intel Pentium	MMX: MultiMedia eXtensions [28]	1996	announced
		1997	shipped
Intel Katmai	SIMD floating-point extensions [18]	1998	beta
Motorola PowerPC	AltiVec [27]	1998	announced

Figure 1. Short Vector Extensions in General-Purpose Microprocessors

signs [4].

Rather than combine a complex superscalar core with short wide vector hardware, we propose using a much simpler processor design that is similar to traditional vector computers with long vectors and simple control logic for instruction issue. Such a design would use the bulk of its transistors and die area for datapath and registers, and thus lessen the time required to design, implement, and verify control.

In this paper, we present data that quantifies this trading of control transistors for datapath and register transistors. We demonstrate that a 2-way, in-order vector processor with a vector length of 64 and a vector width of 8 requires no more die area, and possibly significantly less area, than a 4-way, out-of-order superscalar processor with short vector extensions. Furthermore, we show that the simple long vector processor is, on average, 2.7 times faster executing multimedia applications than the superscalar processor, and 1.6 times faster than one with short vector extensions.

For this paper, we focus on the use of vector architectures for multimedia applications because, as mentioned earlier, multimedia applications are growing in importance and because the effectiveness of vector architectures in other application areas has been reported elsewhere. Their effectiveness on scientific and engineering applications has been demonstrated by their historically dominant use in the supercomputing arena, while other researchers are currently investigating the vectorizability of SPECint programs [1].

The remainder of the paper is organized as follows. In the next section, we describe the details of the processors that we study. In Section 3, we give area estimates for the simple long vector processor and compare its area to those of existing OOO superscalar processors. In Section 4, we present CPI and CPO analyses of simulation-based performance data to explain why greater performance is achieved by the long vector processor.

2. Processor Configurations

Figure 2 lists the features of the processors in our study. We include an out-of-order, 4-way superscalar processor for comparative purposes. The OOO superscalar processor is modeled after the MIPS R10000 with a PA8000-sized re-order buffer [30, 21].

The features that are most relevant to this study are highlighted in bold: issue order, issue width, vector length, and vector width. The last two features are determined by the configuration of the vector register file and the vector datapath, respectively. These features are varied in different combinations for the three processors while other features are not. For example, the ISA, cache-based memory system, and memory bandwidth are the same for all three processors. In this way, the impact on performance and die area of the common features should be approximately the same across all processors. Thus any performance or cost differences that we observe can be attributed to the four features of interest.

The vector processors are based on the Torrent-0 (T0) microprocessor [2, 29]. The T0 is a single-chip vector microprocessor that was implemented by researchers at the University of California at Berkeley. It is fabricated with Hewlett-Packard's CMOS26G process using 1.0μm scalable CMOS design rules and two metal layers, and was first fully functional in April 1995 at 45MHz.[1] Unlike vector supercomputers, the T0 implementation is inexpensive by virtue of being fabricated as a single VLSI chip [23]. In addition to being inexpensive, T0 is also a "nimble" vector implementation [1]. Much of T0's nimbleness can be attributed to the tight integration of the scalar processor and vector hardware on a single die, thus reducing the scalar overhead of vector execution significantly. T0's single-die implementation also allows back-to-back vector instructions to execute in the same vector

[1]The main reason for the relatively slow clock rate is the coarser process technology. The 45MHz clock rate is actually competitive with full-custom commercial processors implemented in similar processes [1].

FEATURE	PROCESSORS		
	OOO Superscalar	OOO Short Vector	Simple Long Vector
ISA	64b MIPS	64b MIPS with vector extensions	
issue order	**out of order**		**in order**
issue width	**4 instructions**		**2 instructions**
fetch width	4 instructions		2 instructions
re-order buffer size	56 instructions		—
#physical registers	64 integer 64 floating-point	64 integer 64 floating-point 32 **8-element** vector	32 integer 32 floating-point 32 **64-element** vector
datapath	2 integer units 1 load/store unit	2 integer units 1 load/store unit **1 VU with 8 IUs**	2 integer units 1 load/store unit **1 VU with 8 IUs**
memory system	64-bit data bus, 64-bit address bus 2-level cache memory based on the R10000 implementation		
C compiler	SGI V5.3 –O2	SGI V5.3 –O2 and VSUIF V1.1.0	

Figure 2. Superscalar and Vector Processors

unit without any intervening recovery cycles. A vector unit in a supercomputer with its board-level implementation must typically wait an additional 4 cycles when executing successive vector instructions. Additional advantages of implementing a vector architecture in microprocessor technology, rather than supercomputer technology, include an area-efficient implementation of a high bandwidth vector register file, more efficient energy consumption, and microprocessor-like operational latencies.

Although we base the details of our vector implementation on the T0, we use our own vector ISA [22]. We defined a separate vector ISA in order to support floating-point data for graphics applications as well as scientific and engineering applications. Moreover, our vector extension is different from the MIPS MDMX short vector extension [12] because we are also interested in studying the effects of varying vector lengths and vector widths.

Our vector MIPS ISA is defined in the coprocessor 2 opcode space and includes memory and computational instructions that execute in vector mode. The memory instructions include loads and stores of various data widths using unit stride, constant stride, or gather/scatter. The computational instructions include integer arithmetic, logical, shift, and floating-point instructions. The vector ISA is a load/store architecture which defines 32 vector registers, each of which consists of 64 elements. Similar to traditional vector architectures, and unlike the recent short vector extensions, our vector ISA includes a vector length register (VLR) that specifies the number of operations a vector instruction executes.

T0-like features in the vector processors we study include chaining of data-dependent vector instructions and zero-cycle recovery time for the vector unit when executing successive vector instructions. As in the T0, both vector memory and vector computational instructions can be chained. In addition to handling RAW data hazards, the vector processors also resolve RAR structural hazards. Finally, operational latencies are the same as in the R10000 [30] although the integer units in the vector datapath include a 16x64 multiplier array to allow vector multiply instructions for narrower data widths to execute at a fully pipelined rate. Vector multiplication of wider data types has a non-unit repeat latency.

The bus interface to the memory system consists of a 64-bit address bus and a 64-bit data bus. At most, only one address can be placed on the 64-bit address bus in one cycle. As a result, only unit-stride vector instructions can proceed at the maximum *memory* vector width. Non-unit stride vector memory instructions proceed with a vector width of one.

Because the memory data bus is 64 bits, the *memory* vector widths for unit-stride vector instructions are determined by the width of data being accessed. For example, 8 8-bit data or 4 16-bit data could be accessed in a single clock cycle. On the other hand, the vector width for *computational* vector instructions is fixed at 8 by the number of integer units in a vector unit. Thus it is possible to have a computational vector width of 8 and a memory vector width of 4 when processing halfword data. Chaining hardware automatically provides the necessary dependency checks to execute chained computational and memory instructions of different vector widths.

The memory system itself is a 2-level cache system based on the R10000. The split L1 caches are each 64KB, 2-way set-associative with 32B lines and 4 banks. The unified L2 cache is 512KB, 4-way set-associative with 64B and 8 banks. The L1 miss penalty is 6 cycles and the L2 miss penalty is 19 cycles.

We emulate the execution behavior of an OOO short vector extension using our more conventional vector ISA and set the vector length to 8. In addition rather than using a partitioned 64-bit datapath, we instead use a vector unit containing 8 integer units. How an 8-wide vector datapath is implemented primarily affects the cost of the implementation and has little impact on performance. Thus, when determining the area requirements of the OOO short vector processor we assume the partitioned datapath implementation but for our performance study we assume the vector-unit implementation to simplify our simulation infrastructure.

By emulating the OOO short vector extensions with a more conventional vector datapath, we will overestimate the performance of programs using data types that are wider than 8 bits. This is because in a 64-bit partitioned vector implementation, a program with 16-bit data would execute with a vector length of 4 and vector width of 4. On the other hand, the OOO short vector processor in our study executes such a program with a vector length of 8, vector memory width of 4, and vector computational width of 8.

3. Processor Die Areas

In this section, we estimate the area required to implement the simple long vector processor and compare its area to those of existing OOO superscalar processors. We focus only on the processor portion of the die since, in our performance study, we assume the same cache-based memory system for all the processors.

3.1. Simple Long Vector Processor

We determine the area required to implement the simple long vector processor by estimating the space requirements for each of the major processor components: vector datapath, vector register file, scalar datapath, scalar register file, and control logic for instruction issue.

Using the Torrent-0 as a realistic VLSI implementation of a vector microprocessor, we conducted a detailed analysis of T0's area requirements in a previous study [23]. Our analysis showed that, of the major processor components, the most area-intensive for a vector processor are the datapath and vector register file in contrast to aggressive superscalar processors where control for instruction issue dominates.

For this study, we improve upon our previous analysis by using more precise estimates for the component areas. These are listed in Figure 3. We give areas for two implementations of the long vector processor: an area-efficient implementation and a high-performance implementation. Differences in area between the two are highlighted in bold font. As in our previous study, these areas

PROCESSOR COMPONENT	AREA IN MM^2: Area Efficient	AREA IN MM^2: High Performance
64b Vector Datapath		
8 integer units	**24.0**	**36.0**
load/store unit	3.0	3.0
64b Vector Register File		
32 64-element vector registers	**9.5**	**19.0**
64b MIPS R5000		
scalar integer and FP datapath	10.3	10.3
scalar integer and FP register file	0.5	0.5
instruction issue	0.8	0.8
Clocking and Overhead	4.0	4.0
TOTAL	**52.3**	**73.8**

Figure 3. Component Areas for Two Implementations of the Long Vector Processor

are based on actual implementations. Because the implementations on which we base the estimates are implemented in varying process technologies, we scale all the areas to a common line size of 0.25μm.

The area-efficient 64b integer units are 3mm^2 each. This area has been extrapolated from T0's 32b MIPS scalar datapath and 16x16 multiplier array as follows: $2 \times$ area of 32b scalar datapath $+ 4 \times$ area for 16x16 multiplier array $= 2 \times 1\text{mm}^2 + 4 \times 0.25\text{mm}^2$. The 16x64 multiplier array allows vector multiplies of narrow data types ($\leq$ 16 bits) to be fully pipelined.

As independent evidence that 3mm^2 is a reasonable area estimate for a 64b integer unit, we measured the areas of integer units in several in-order and out-of-order superscalar implementations using annotated photomicrographs and floorplans [11, 3, 13, 10, 26, 9]. The areas of the integer "execution box" for in-order processors such as the MIPS R5000 and Alpha 21164 range from 3.25mm^2 to 3.72mm^2. These execution boxes include iterative multiplier/dividers, and two ALUs. The OOO counterparts, MIPS R10000 and Alpha 21264, have adder/multiplier units that measure 4.5–4.65mm^2. Additional bussing and reservation stations to support out-of-order execution are the reasons for the increase in area.

Because the simple long vector processor uses in-order issue, it would use integer units closer in design to the in-order units. Although the area we choose is slightly smaller than that of the in-order units, the T0 implementation suggests that this is a conservative estimate. The individual integer units of T0's vector datapath are much smaller than the MIPS scalar datapath because they do not require hardware, such as logic for PC handling, that is included in the scalar core. Nonetheless, we conservatively base our area estimate on the larger datapath.

We estimate the area for a high performance integer unit to be 4.5mm^2 which is the area for an OOO unit.

We extrapolate the area for the load/store unit from T0's vector memory datapath as follows: area for 128x256 crossbar $+ \; 2 \; \times$ area for shifting/aligning 32b data. The load/store unit interfaces with a 64-bit memory data bus on one side and 8 64-bit data busses on the other, thus requiring a 64x512 cross-bar to transfer data from the memory system into the vector register file. T0 has a 128x256 cross-bar to perform the same functionality and that occupies about 1mm^2. This crossbar can be restructured without any area growth to provide the necessary functionality for the long vector processor. Shifting and aligning 64b data between the memory data bus and register busses requires twice the area used in the 32b T0 vector memory unit.

Because there is only one address bus, only one address needs to be computed per clock cycle. Moreover, scalar and vector instructions use the same 64b data bus to the memory system. Thus, computing addresses and handling TLB functions is handled by hardware in the scalar portion of the processor, and additional area for address functionality is not needed in the vector load/store unit.

We compute the areas for the two implementations of our vector register file based on layout details given by Asanović in Chapter 5 of his doctorate dissertation [1]. In addition to giving the area for storing data, the details of the register file design also include the area for overhead circuitry such as read sense amplifiers, data latches for writes and reads, multiplexors, and drivers, The areas of all the storage and overhead components are based on cells used in the T0 implementation.

The area-efficient vector register file time-multiplexes the word and bit lines at twice the clock rate to allow both a read and a write access in the same clock period. The high-performance implementation adds additional bussing and ports to avoid time-multiplexing while still allowing simultaneous read and write accesses.

We use the MIPS R5000 [11] as the 64-bit scalar processor, which is responsible for instruction control, scalar execution, and data address processing.

3.2. Comparison of Die Areas

Figure 4 plots the areas of the simple long vector processor and several superscalar processors. Also shown is a breakdown of each processor's area into its major components. "Other" is the area for clocking and overhead. We obtained the areas of the superscalar processors by measuring annotated die photomicrographs and floorplans [10, 26, 9].

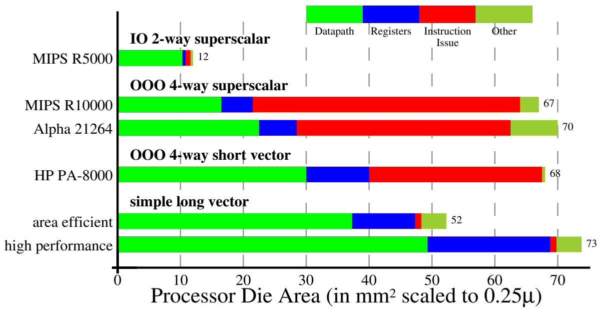

Figure 4. Breakdown of Processor Areas

A comparison of die areas of different implementations 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, non-processor components, datapath width, and circuit and layout design. We want to minimize differences due to these factors so that remaining differences will be due to parallel-specific features such as datapath organization, register file implementation, and instruction-issue mechanism.

To this end, all die areas are scaled to a 0.25μm process to eliminate areal differences due to differences in line size, The areas that we compare are for processor components only. Excluded are cache and TLB structures, external interface logic, and the pad ring. Lastly, all areas are based on actual VLSI implementations.

The MIPS R10000, Alpha 21264, and HP PA-8000 are existing implementations of OOO superscalar and OOO short vector processors. Of the three OOO superscalar processors, only the PA-8000 implements its short vector extensions, MAX-2. The implementation for the MAX-2 instructions occupies less than 0.2% of the area shown in Figure 4 [25]. For comparison purposes, areas for the MIPS R5000 are also included.

The area-efficient implementation of the long vector processor is about 70–75% the size of the OOO implementations, while the high-performance version is 4–7mm^2 larger. Note that this is only for the processor portion of the die. Typically there are on-chip caches as well as hardware for TLBs and external interface logic. For the MIPS R10000 and Alpha 21264, the processor portion occupies 45% of the die, while 85% of the HP PA-8000 die is dedicated to the processor. The reason a larger portion of the die is used for the PA-8000 processor is because its die does not include any on-chip caches. Thus, while 7mm^2 is a 10% increase over the *processor* area of an OOO superscalar implementation, it is at most a 5% growth in total die area after non-processor components such as on-chip caches are considered.

Consistent with the results in our previous study, the area breakdown shows that the simple long vector processor uses its transistors differently from the OOO superscalar processors. For the long vector processor, 70% of the die is dedicated to functional units, while only 1–2% is used for instruction issue. The remaining 20–25% of the die is used for the large vector register file. In contrast, the OOO superscalar processors use 40–65% of the area for hardware structures to support multi-way and out-of-order instruction issue. This is comparable or, in some cases, more than the area used for datapath.

The reason the long vector processor is able to trade instruction-issue for datapath and registers is because it uses vector instructions that compactly encode parallel operations. Thus much of the functionality for detecting parallelism is transferred to software. Figure 4 quantifies this tradeoff and shows that using area for datapath and registers rather than control is a reasonable alternative to existing designs. In the next section, we will show that this tradeoff also provides significant performance gains as well.

We have been conservative with our estimates of the long vector processor's die area. For example, we use full 64-bit integer units for the vector datapath and 64-bit elements for the vector register file. A more area-conservative design could use a partitioned datapath such as in the HP PA-8000 and implement a correspondingly narrower vector register file. In addition by re-structuring the vector register file, it is possible to reduce its area requirements by almost half without sacrificing bandwidth [1]. In short, it may be possible to implement a simple long vector multimedia processor in one-third to one-half of the area of existing OOO superscalar processors.

4. Performance Results

We now investigate the performance of vector processors executing multimedia applications.

Figure 5 describes the programs and inputs we use as the workload for our performance study. The widths of the dominant data types are also listed. All programs are written in C. Four are image processing applications while the other two programs are versions of the IDEA encryption/decryption algorithm for PGP. Details of the two versions and an analysis of their performance difference are given elsewhere [23]. All these programs contain highly vectorizable functions where much of the execution time is spent.

The same C source code is executed on all three processors. For superscalar-only execution, we compile and

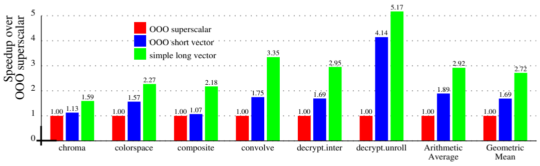

Figure 6. Processor Performance

link the programs with the SGI C compiler. For vector execution, we compile the vectorizable functions with VSUIF, a vectorizing compiler we developed [6]. Because little effort has been put into optimizing the code generated by VSUIF, non-vectorizable portions are compiled with the SGI compiler to benefit from its optimizing cabilities. The SGI linker is used to combine the separately compiled object files into an executable.

To generate performance statistics, both superscalar and vector executables are executed using `Vmable`, a MIPS ISA simulator that we modified to emulate the execution of vector instructions. Correctness, especially of the vector executable, is verified by comparing `Vmable`'s output against that of native execution. `Vmable` also generates instruction and data addresses that are input to `Vcello`, our vector-enhanced version of Todd Mowry's `cello` timing simulator. `Vcello` can be configured to emulate the timing of the processors that we study. `Vcello` also keeps track of cache misses, branch mispredictions, etc. and produces architectural performance statistics that characterize the execution of a program.

Figure 6 shows the performance of the two vector processors relative to the superscalar processor. The speedup graph shows that vector execution is faster than wide-issue, out-of-order scalar execution: the short vector processor is on average 1.7 times faster than the superscalar processor, and the long vector processor is 2.7 times faster.[2] This is not surprising since the vector processors support more parallelism. Recall, however, that the cost of this greater support, as measured by die area, is not significantly larger than that of the OOO-only implementation.

The OOO short vector and simple long vector processors have the same peak computational rate and peak memory bandwidth. Nonetheless, the simple longer vector processor is 1.25 to 2.0 times faster than the complex shorter vector processor. Across all the programs,

[2] When reporting averages in the text, we use the geometric mean (GM) because we are averaging ratios (CPI and CPO) and normalized data [8]. Consequently, we can use the GM's multiplicative properties when summarizing our analyses of CPI and CPO data. We also show the arithmetic average, which has a more intuitive interpretation, together with the geometric mean in the graphs. Because both averages are within a few percent of each other for the data we present, the geometric mean is a reasonable approximation of the arithmetic average.

BENCHMARK	DATA WIDTH	INPUT	DESCRIPTION
`chroma`	8 bit	320x240	Merges two images on the basis of a “whiteness” threshold.
`colorspace`	8 bit	24-bit	Converts an image in RGB to YUV values.
`composite`	8 bit	color	Blends two images together by a blend factor α.
`convolve`	8,16 bit	image(s)	Convolves an image with a 3x3 16-bit kernel.
`decrypt.unroll`	16 bit	16,000-byte	Unrolled version of IDEA decryption.
`decrypt.inter`	16 bit	message	Loop-interchanged version of IDEA decryption.

Figure 5. Benchmark Programs

the long vector processor executes on average 1.6 times faster than the short vector processor.

In the remainder of this section, we analyze the performance of the three processors to explain why the vector processors are faster than the superscalar-only processor and why the longer vector is faster than the shorter.

4.1. CPI $\times$ Dynamic Instruction Count

The classic method for determining why one computer executes faster than another is to deconstruct each computer's execution time into the three components of the CPU performance equation: cycle time, average cycles per instruction (CPI), and dynamic instruction count [17]. In the remainder of this paper, we refer to this equation as the “CPI equation” to distinguish it from the “CPO equation” that we discuss in the next subsection.

For the purposes of this study, we make the reasonable assumption that all three processors can be implemented with similar state-of-the-art clock frequencies. OOO short vector processors have already been implemented at the same clock frequencies as OOO superscalar processors, while a long vector processor with its simpler control logic could be implemented with a much faster clock frequency. Thus this assumption potentially underestimates the performance of the simple long vector processor.

The graphs in Figure 7 show the data for the other two components of the CPI equation. For the dynamic instruction count, we also show the number of scalar and vector instructions executed for each program.

As expected, the CPI for the superscalar processor is low: between 0.4 and 0.7, or equivalently between 1.4 and 2.5 instructions per cycle.

All instructions in the short vector processor including the vector ones still require a minimum of only one clock period to execute. Nonetheless, this processor has a CPI that is, on average, 2.5 times higher than the CPI of the superscalar processor. The higher CPI suggests that less instruction-level parallelism is being exploited. One reason for the reduced ILP is that by using vector instructions, there is less parallelism to be used at the *instruction*

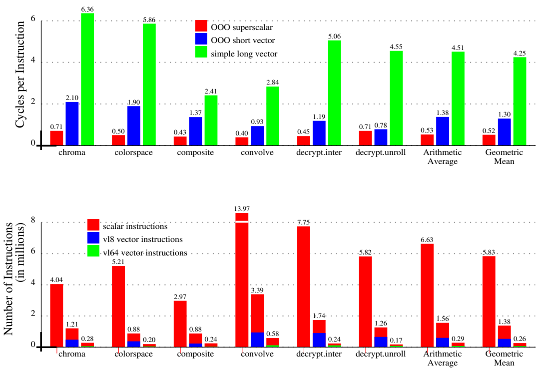

Figure 7. CPI & Dynamic Instruction Count

level. A vector instruction executes 8 operations that, in a scalar processor, would be 8 separate instructions that could be executed in parallel.

The CPI for the long vector processor is very high, ranging from 2.4 to 6.4 cycles per instructions. This is 6 to 12 times greater than the superscalar's CPI. The reason for the substantially higher CPI is that a vector instruction for the long vector processor takes 8 clock periods to execute.

Despite the substantially higher CPIs, the vector processors are faster than the superscalar processor. To understand why, it is important to consider the other half of the CPI equation: the dynamic instruction count. The bottom half of Figure 7 shows the number of instructions executed by each processor.

An immediately obvious trend among the processors is that the dynamic instruction count is inversely proportional to the CPI. The superscalar processor, which has the lowest CPI, executes far more instructions than the vector processors. In contrast, the long vector processor with the highest CPI executes 12–34 times fewer instructions than the superscalar processor and 3.7–7.4 times fewer instructions than the short vector processor.

The greatly reduced instruction count is the result of using stripmined vector code. A stripmined vector loop that uses vector lengths of 64 is comparable to *implic-*

itly unrolling the loop 64 times. The scalar instructions that handle the bookkeeping of the stripmined code correspond to the loop overhead code of the original loop. The vector instructions execute the operations of the loop body itself.

Stripmining a loop is better than explicitly unrolling it for two reasons. First, the stripmined vector loop reduces the *dynamic* count for both the loop overhead instructions and loop body instructions, while explicit unrolling reduces only the overhead count. Second, stripmining achieves this reduction without substantially increasing the *static* instruction count, whereas explicit unrolling requires replicating loop body instructions.

An example of a loop overhead instruction is one that updates the loop index variable (`add r1,r1,1`). In the rolled version of a loop, the same scalar instruction for carrying out this functionality is executed 64 times. The stripmined and unrolled versions would collapse these into a single instruction (`add r1,r1,64`) that is executed once for every 64 loop iterations. Similar transformations are applied to other scalar instructions that constitute the loop overhead. Examples include those that update address values. In this way the number of loop overhead instructions executed is reduced by a factor of 64.

An example of an instruction sequence that includes a loop body instruction is one that loads an element from an array and updates the associated address register: `load r3,0(r2); add r2,r2,4`. In this case, the `load` instruction is a loop body instruction while the `add` instruction is a loop overhead instruction. In the rolled version, these two instructions would be executed 64 times resulting in a dynamic instruction count of 128.

For the version that explicitly unrolls the loop 64 times, the `load` instruction would be repeated 64 times but with a different offset for each `load` instruction. The address update instruction, however, would *not* be repeated and instead would be transformed into `add r2,r2,256`. Thus the number of *static* instructions has grown from 2 to 65, while the *dynamic* instruction count has been reduced from 128 to 65.

The stripmined vector version reduces the dynamic count even further, while keeping the static count comparable to the rolled version. The 64 `load` instructions are now replaced with a single vector load instruction `vload v3,(r2),4` that accesses memory locations that are 4 bytes apart. This results in only 2 instructions being executed to carry out 64 loads and their associated address updates.

The reduction in dynamic instruction count for loop overhead is obtained by solving simple linear recurrences, a technique that is independent of instruction type and thus available to all three processors. On the other hand, the tremendous savings for the loop body is achieved by using vector instructions, a technique obviously not available for non-vector processors. In short, a stripmined vector loop of length VL can reduce the *dynamic* instruction count for that loop by a factor close to VL while keeping the *static* count close to that of the original rolled version.

It is not always possible, however, to reduce the *overall* dynamic instruction count of the scalar version by a factor of VL for several reasons. The most obvious reason is that vector loops do not always dominate the execution time. While the bulk of time is spent in vector loops for the programs in this study, a small but significant amount is spent in non-vectorizable parts of the program.

Another reason is that there is not always a one-to-one correspondence between instructions in the rolled loop version and instructions in the stripmined version. As a result, the static instruction count for the stripmined version can be slightly higher than the count for the rolled version. For example, instructions to update the vector length register must be introduced into the stripmined version. In addition, depending upon the ISA support for conditional vector execution, more vector instructions may be used to handle conditional statments than in the scalar version. A final example is when the scalar version is unrolled several times. The SGI compiler we use for the superscalar processor will unroll loops 2 to 4 times depending upon the size of the loop body.

Even though it is not always possible to achieve a reduction of VL in the dynamic instruction count, using stripmining to reduce this count is more effective than explicity unrolling. This is because VL can be much greater than the amount typically used for unrolling. Small values of VL such as 8, are large for explicit unrolling, while more typical VL values, such as 32 or 64, are ineffective for unrolling due to the tremendous growth in the number of static instructions generated as well as the number of registers required. On the other hand, support for vector execution with length 32 or 64 is not prohibitively expensive as explained in the previous section.

Figure 8 summarizes the results of our CPI analysis. To explain why the vector processors are faster than the superscalar processor, we express the average speedup of a vector processor as a function of the superscalar processor's average CPI ($\overline{\mathrm{CPI}}_{\mathrm{SS}}$) and average dynamic instruction count ($\overline{\mathrm{NI}}_{\mathrm{SS}}$). This is possible because the geometric mean is used to compute the averages and we use the multiplicative properties of the geometric mean to guarantee equivalency between the two expressions.

From Figure 8, we observe that vector processors have a significantly higher average CPI than the superscalar processor does: the average CPI of the short vector processor is 2.5 times higher while the long vector's

PROCESSOR	CYCLES PER INSTRUCTION	DYNAMIC INSTRUCTION COUNT	CYCLE COUNT	SPEEDUP OVER SUPER-SCALAR
OOO superscalar	$\overline{\text{CPI}}_{SS}$	$\overline{\text{NI}}_{SS}$	$\overline{\text{CPI}}_{SS} \times \overline{\text{NI}}_{SS}$	$1.00\times$
OOO short vector	$2.50\times\overline{\text{CPI}}_{SS}$	$0.24\times\overline{\text{NI}}_{SS}$	$0.60\times\overline{\text{CPI}}_{SS} \times \overline{\text{NI}}_{SS}$	$1.67\times$
simple long vector	$8.19\times\overline{\text{CPI}}_{SS}$	$0.045\times\overline{\text{NI}}_{SS}$	$0.37\times\overline{\text{CPI}}_{SS} \times \overline{\text{NI}}_{SS}$	$2.71\times$

Figure 8. Average Speedup Deconstructed Using CPI Equation

average CPI is substantially greater at 8.2 times that of the superscalar's CPI. On the other hand, vector processors execute far fewer instructions: the short vector processor executes, on average, about 4 times fewer instructions while the long vector processor reduces the instruction count by more than a factor of 22.

The tremendous reduction in dynamic instruction count more than compensates for the increase in CPI, resulting in the OOO short vector processor being 1.7 times faster than the OOO superscalar processor and the simple long vector being 2.7 times faster. Moreover the long vector's reduction in instruction count is sufficiently large that it is 1.6 times faster than the OOO short vector despite having a CPI that is 3.3 times higher.

4.2. CPO $\times$ Dynamic Operation Count

Using the CPI equation to deconstruct execution time explains why the vector processors are more effective than the superscalar processor when executing highly vectorizable programs. However, it does not provide much insight into how efficient a vector processor is at executing parallel loops. For example, given that individual instructions can take 1 or 8 cycles to execute in the long vector processor, a CPI less than 8 could be the result of using a significant amount of parallelism effectively; or it could be due to having far more scalar instructions being executed than vector instructions.

The reason for this ambiguity is that an instruction is no longer an appropriate unit of work for comparison purposes. The amount of work carried out by an instruction varies tremendously depending upon both the type of instruction and the vector length. When all instructions take about the same amount of time to execute, a high CPI indicates possible performance inefficiences in the implementation. On the other hand, it is possible for a long vector processor to use parallelism effectively and still have a high CPI.

Rather than using an instruction as a unit of work, we recommend using an operation which is the amount of work that is carried out by a functional unit. One scalar instruction executes one operation while one vector instruction executes VL operations where VL is the value

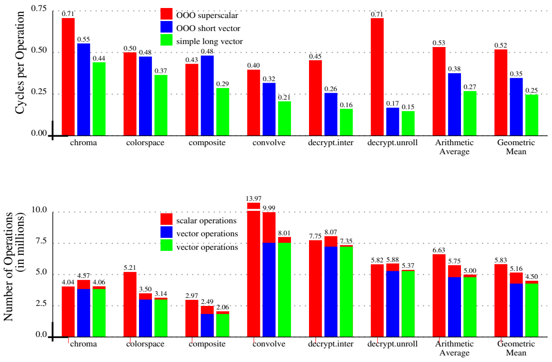

Figure 9. CPO & Dynamic Operation Count

of the vector length register when the instruction is executed.[3]

Figure 9 shows the cycles per operations (CPO) and dynamic operation count for the three processors. The data for the superscalar processor are the same as in the instruction-centric graphs. For the dynamic operation count, we also show the number of scalar and vector operations executed for each program.

Whereas the CPIs are higher, the CPOs of the vector processors are lower than the superscalar's CPO. Excluding the CPO for `decrypt.unroll`, the short vector's CPOs are up to 1.7 times lower than the superscalar's CPO while the long vector's CPOs are 1.35–2.8 times lower. For `decrypt.unroll`, the vector CPOs are more than 4 times smaller.

Recall that the higher CPI of the short vector processor indicates that it is finding less ILP than the OOO superscalar processor. Its lower CPO indicates that the short vector processor compensates for the re-

[3] This mapping is a rough measure of work. A scalar memory instruction that uses an auto-increment addressing mode could be counted as executing two operations: the memory access and the address register update. In a similar fashion, vector memory instructions could also be counted as executing $2\times$VL operations to account for both the memory accesses and the address register updates. However to keep the mapping simple, we count only one operation per pair of memory access and address register update.

duced *instruction*-level parallelism by effectively using *operation*-level parallelism. More generally because operations, unlike instructions, carry out approximately equivalent amounts of work in all three processors, the vector processors' lower CPO indicates that they are using the available parallelism in the program more effectively than the superscalar processor.

The CPO of the long vector processor is 1.1–1.7 times lower than the short vector's CPO across the programs. The lower CPO suggests that the long vector processor is able to use parallelism more effectively. By using a vector width of 8, each processor is able to take advantage of the *operational* parallelism within a vector instruction equally well. It is at the *instruction* level where the two processors differ in their use of parallelism as a consequence of their differences in issue mechanism and vector length. Figure 10 summarizes the types of instruction-level parallelism that each can use.

The complex short vector processor uses its out-of-order wide issue to execute vector instructions in parallel with scalar instructions. In addition, scalar instructions also execute in parallel. It is rare, however, that vector instructions would execute in parallel with each other for several reasons. Vector instructions take only one clock cycle to execute, and there is only one vector memory unit and one vector computational unit. Thus vector-vector ILP requires a vector memory instruction and a vector computational instruction that have no data dependencies.

It is unlikely, however, that such a pair will be found because there are often data dependences among vector instructions in the same vector strip. Vector instructions that can execute in parallel are typically from subsequent vector strips. However, a non-trivial loop usually generates a vector strip that contains more than 10 instructions that often include several memory instructions. Even with complex out-of-order issue hardware, it is unlikely that the hardware would be able to execute vector instructions from different strips in parallel.

The simple long vector processor, on the other hand, is able to execute vector memory instructions in parallel with vector computational instructions. This is because a vector instruction takes 8 cycles to execute and with chaining, even data-dependent vector instructions can overlap in execution. In addition, scalar instructions can also execute in parallel with already executing vector instructions. With in-order single issue, however, scalar instructions cannot execute in parallel.

For both processors, it is possible to use compiler techniques to overcome the inability of each to exploit a specific type of ILP. For the short vector processor, loop unrolling or the more sophisticated technique of software pipelining can be used to schedule vector instructions from different strips so that they could execute in parallel. For the long vector processor, the use of simple list scheduling within a vector strip can allow most of the scalar instructions to execute in parallel with vector instructions and thus transform potential scalar-scalar ILP into achievable vector-scalar ILP. Neither of these techniques is currently used by our vectorizing compiler.

The long vector processor's lower CPO constitutes only a portion of its speedup over the short vector processor. As before, it is important to consider dynamic operation count, the other half of the CPO equation, to account for the remaining speedup factor. The bottom half of Figure 9 shows the number of operations executed by each processor.

The long vector processor always executes fewer operations than the short vector processor. This is entirely due to the fact that the long vector processor executes about 7 times fewer scalar operations, on average, to accomplish the same task. Both vector processors execute the same number of vector operations for a given program.

The reduction in scalar operation count is a direct result of the long vector processor having a vector length that is 8 times longer than the short vector processor. Recall that different techniques are used to reduce the dynamic instruction count for the loop overhead and loop body in a stripmined vector loop: loop overhead instructions are eliminated by solving simple linear recurrences, while a loop body instruction is replaced by a single vector instruction. By implicity unrolling the loop 64 times rather than 8 times, the long vector processor executes the same number of vector operations as the short vector processor but 8 times fewer scalar operations for the vectorized loop.

The overall reduction in scalar operations is less than a factor of 8 because a small but non-trivial number of the scalar instructions are executed for non-vectorizable parts of the program.

Figure 11 summarizes the results of our CPO analysis. We use the same averaging technique as we used to summarize our CPI analysis in Figure 8. Unlike the CPI results, the CPO results show that vector processors achieve greater performance by reducing *both* components of the CPO equation. First, the short and long vector processors are more effective at using parallelism to achieve a CPO that is, on average, 1.5 and 2.1 times lower than the superscalar's CPO. Second, the vector processors execute fewer operations by stripmining vector loops: the short vector processor executes, on average, 1.1 times fewer operations while the long vector processor executes 1.3 fewer.

The CPO analysis also provides more insight into why the simple long vector processor is faster than the complex short one. The bulk of the speedup is the result of a CPO that is, on average, 1.4 times lower, with an

ILP TYPE	VECTOR PROCESSORS OOO short	simple long	COMPILER ASSISTANCE
scalar–scalar	√	×	use simple list scheduling to enable vector–scalar ILP instead
vector–scalar	√	√	
vector–vector	×	√	use loop unrolling or software pipelining to enable

Figure 10. Effective ILP in Vector Processors

PROCESSOR	CYCLES PER OPERATION	DYNAMIC OPERATION COUNT	CYCLE COUNT	SPEEDUP OVER SUPER-SCALAR
OOO superscalar	$\overline{\mathrm{CPO}}_{\mathrm{SS}}$	$\overline{\mathrm{NO}}_{\mathrm{SS}}$	$\overline{\mathrm{CPO}}_{\mathrm{SS}} \times \overline{\mathrm{NO}}_{\mathrm{SS}}$	1.00×
OOO short vector	$0.67 \times \overline{\mathrm{CPO}}_{\mathrm{SS}}$	$0.88 \times \overline{\mathrm{NO}}_{\mathrm{SS}}$	$0.59 \times \overline{\mathrm{CPO}}_{\mathrm{SS}} \times \overline{\mathrm{NO}}_{\mathrm{SS}}$	1.70×
simple long vector	$0.48 \times \overline{\mathrm{CPO}}_{\mathrm{SS}}$	$0.77 \times \overline{\mathrm{NO}}_{\mathrm{SS}}$	$0.37 \times \overline{\mathrm{CPO}}_{\mathrm{SS}} \times \overline{\mathrm{NO}}_{\mathrm{SS}}$	2.70×

Figure 11. Average Speedup Deconstructed Using CPO Equation

average 1.15x lower operation count providing the rest of the speedup. By using *instruction*-level parallelism more effectively and executing fewer *scalar* operations, the simple long vector processor is on average 1.6 times faster than the complex short vector processor.

5. Summary and Conclusions

We have presented die area measurements and simulation-based performance data to demonstrate the cost/performance benefits of using a 2-way, in-order vector microprocessor with a vector length of 64 and vector width of 8 to execute multimedia applications. Unlike a 4-way OOO superscalar processor with short vector extensions, this simple long vector processor is configured more like traditional vector architectures but with two major enhancements: much wider vectors and slightly wider instruction issue.

The cost benefits of the long vector processor are made possible by implementing it in VLSI technology. Using components from actual VLSI implementations, we showed that a conservative estimate of a high-performance implementation of this processor would be comparable in area to existing OOO 4-way superscalar processors. An area-efficient implementation would be 70–75% smaller, while a more aggressive implementation could possibly be one-third to one-half the size of an OOO superscalar processor. Furthermore, the simple long vector processor should also be easier to design, implement, and verify. This is because the bulk of its die area is used for datapath and registers rather than complex instruction-issue logic.

A VLSI implementation of a vector processor also provides performance benefits. Examples include vector units with zero-cycle recovery time, short operational latencies, and a high-bandwidth, area-efficient vector register file. Latencies are comparable to those in other microprocessor implementations, rather than to latencies in supercomputer implementations which are considered the quintessential vector technology.

Additional performance gains are achieved from the vector ISA. Although a long vector length results in a much higher CPI, it also reduces the dynamic instruction count substantially without significantly increasing the static instruction count. By using cycles per *operation* as a metric, we showed that the simple long vector processor is more effective at using vectorizable parallelism than the superscalar processors despite have a significantly higher CPI. Thus by executing more operations per cycle and fewer operations in total, the simple long vector processor is, on average, 2.7 times faster than a 4-way OOO superscalar processor and 1.6 times faster than a superscalar processor with short vector extensions.

These performance gains are obtained with an R10000-like cache-based memory system being used for all three processors. Thus it is not necessary for a vector implementation to require a costly, high bandwidth memory system to achieve high performance. In fact, a preliminary analysis of the cache performance statistics suggests that all three processors could benefit from a more sophisticated cache system.

In summary, the complexity and area cost for providing highly parallel vector support can be substantially less than the complexity and area cost for OOO superscalar support. Moreover, these savings in cost are coupled with a significant increase in performance. Thus a simple, highly parallel, long vector microprocessor represents a cost-effective alternative for executing multimedia applications.

Acknowledgements

We thank David Patterson, Krste Asanović, and the IRAM research group for their helpful comments and suggestions. We also thank Todd Mowry for his expertise in constructing simulators. This work was supported by NSERC, CITO, DARPA (DABT63-96-C-0056), NSF (CDA 94-01156), and the California State MICRO Program.

References

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

[2] Krste Asanović, James Beck, Bertrand Irissou, Brian Kingsbury, Nelson Morgan, and John Wawrzynek. The T0 Vector Microprocessor. In *Proceedings of Hot Chips VII*, pages 187–196, Stanford, CA, August 1995.

[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*, pages 182–183,362, San Fransisco, CA, February 1995.

[4] 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.

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

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

[7] Keith Diefendorff and Pradeep K. Dubey. How Multimedia Workloads Will Change Processor Design. *IEEE Computer*, 30(9):43–45, September 1997.

[8] P.J. Flemming and J.J. Wallace. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Communications of the ACM*, 29(3):218–221, March 1986.

[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*, pages 176–177,451, San Fransisco, CA, February 1997.

[10] Silicon Graphics. MIPS R10000 Microprocessor. *http://www.sgi.com/-MIPS/products/r10k/T5_Die.html*, 1996.

[11] Silicon Graphics. MIPS RISC R5000 Microprocessor. *http://www.sgi.com/MIPS/products/r5000*, 1996.

[12] Silicon Graphics. MIPS Extension for Digital Media with 3D. *http://www.sgi.com/MIPS/arch/ISA5/-index.html#MIPSV_indx*, March 1997.

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

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

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

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

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

[18] Intel. Software Benefits of Katmai New Instructions. *http://developer.intel.com/drg/news/katmai.htm*, 1998.

[19] 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.

[20] Christoforos E. Kozyrakis and David A. Patterson. A New Direction for Computer Architecture Research. To be published in *IEEE Computer*., May 1998.

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

[22] Corinna G. Lee. MIPS Vector Architecture Manual. In preparation., June 1997.

[23] 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.

[24] Ruby B. Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro*, 15(2):22–32, April 1995.

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

[26] Jon Lotz et al. A Quad-Issue Out-of-Order RISC CPU. In *Digest of Technical Papers for the International Solid-State Circuits Conference*, pages 210–211,446, San Fransisco, CA, February 1996.

[27] Motorola. Motorola AltiVec Technology: Home Page. *http://www.mot.com/SPS/PowerPC/AltiVec/*, 1998.

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

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