Adding a Vector Unit to a Superscalar Processor

Francisca Quintana
Departamento de Informatica y Sistemas,
Universidad de Las Palmas de Gran Canaria–Las Palmas, Spain
e-mail: {paqui}@ac.upc.es

Jesus Corbal, Roger Espasa and Mateo Valero
Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya–Barcelona, Spain
e-mail: {jcorbal,mateo,roger}@ac.upc.es

Abstract

The focus of this paper is on adding a vector unit to a superscalar core, as a way to scale current state of the art superscalar processors. The proposed architecture has a vector register file that shares functional units both with the integer datapath and with the floating point datapath. A key point in our proposal is the design of a high performance cache interface that delivers high bandwidth to the vector unit at a low cost and low latency. We propose a double-banked cache with alignment circuitry to serve vector accesses and we study two cache hierarchies: one feeds the vector unit from the L1; the other from the L2. Our results show that large IPC values (higher than 10 in some cases) can be achieved. Moreover the scalability of our architecture simply requires addition of functional units, without requiring more issue bandwidth. As a consequence, the proposed vector unit achieves high performance for numerical and multimedia codes with minimal impact on the cycle time of the processor or on the performance of integer codes.

1 Introduction

Scaling current superscalar processors to achieve large amounts of ILP is an area of very active research. There is a growing consensus that it can not be done by simply trying to fetch, decode and issue more and more instructions per cycle [1]. First of all, an aggressive fetch and decode engine must be designed, which is far from being trivial due to branches as well as instruction cache bandwidth issues [2, 3]. Second, an aggressive issue engine with a large instruction window is required to be able to feed a large number of functional units. The instruction window lookup time increases quadratically with window size [1]. Third, to issue large number of instructions a heavily multiported register file is needed, which can both endanger the cycle time and consume a large amount of chip area. Also, sustaining multiple memory accesses per cycle requires a multiported TLB and cache, and their cost is also proportional to the number of independent memory ports.

All these aspects make the scalability of superscalar processors expensive and strongly technology-dependent. Moreover, even if these problems can be overcome with future technology, the performance results generally do not pay off the amount of chip area and the design effort required.

In this paper we take a novel approach and we address the problem of adding a vector unit to a superscalar processor as a way to achieve large amounts of ILP while sidestepping many of the technological hurdles described above. The goal of this vector unit is to increase performance in two relatively well-defined application areas: numerical and multimedia workloads.

A fundamental problem that must be addressed for the vector unit approach to work is how to adapt current cache hierarchies to provide the required bandwidth at a low cost and with a relatively low latency. This paper addresses this question by proposing a new cache structure that delivers full cache lines to the processor on vector accesses, while still supporting traditional scalar traffic. Our solution provides large bandwidth at low cost compared to using a true multiported cache.

The vector unit we propose is feasible in current technology and it will scale very well as technology allows integrating more and more functional units on chip. The results presented show that it is possible to achieve large speedups by scaling up to sixteen parallel vector lanes while maintaining a simple control engine that fetches and decodes just four instructions per cycle.

2 Related Work

First, it is important to distinguish our proposal from the current tendency in microprocessor design targeted at the exploitation of sub-word parallelism. Most major computer vendors have recently included multimedia specific instructions in their architectures such as MMX [4], VIS [5] or Altivec [6]. Except for the Altivec case, all other extensions only offer sub-word parallelism. That is, a 64-bit register can be broken into independent entities of 8, 16 or 32 bits that are operated on in parallel. Although the term "SIMD" is used to refer to these instructions, care must be taken to distinguish traditional vector-like SIMD operation and sub-word SIMD operation. Our proposal has true vector instructions where a single instruction operates on multiple, independent 64-bit words (a vector register).

The advantage of having full-blown vector units is that numerical applications can benefit greatly from them while,

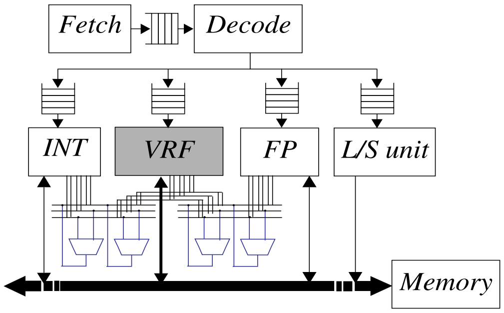

Figure 1: Modeled architecture.

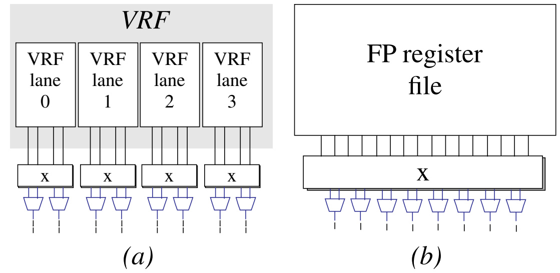

Figure 2: Example of (a) Parallel lanes in vector computing, (b) Superscalar Replication of functional units.

typically, do not take advantage of MMX-like ISAs. Furthermore, some multimedia applications that are not amenable to exploit sub-word parallelism can also take advantage of our vector units. Of course, although beyond the scope of this paper, nothing would prevent our vector units to also include sub-word parallel instructions, such as the ones provided by Altivec.

Second, we must also differentiate this paper from the work recently presented by C. Lee in [7] and [8]. Lee advocates using simple vector processors in future desktop systems. Although this may sound similar to our proposal, indeed the differences are profound. Lee argues that by using vector units, one can keep the scalar core in-order (dispensing with the expensive and slow out-of-order features) and thus favor high clock rates. Although we agree with the argument, we claim that any feature that goes against integer performance will not be adopted by chip vendors. That is, to successfully add a vector unit to general purpose microprocessors, the vector unit has to be perceived as an add-on that does not disrupt, slowdown or interfere with the performance of the integer core. Therefore, if the current tendency is to include out-of-order execution even in high frequency designs such as the Alpha lineage, the vector unit must be adapted to out-of-order execution and register renaming. In this paper we leverage from our previous studies on out-of-order vector processors [9] to fully integrate the vector unit in an out-of-order superscalar processor.

Furthermore, this paper is mainly devoted to designing a feasible cache hierarchy that fits both the scalar engine and the vector unit. Finally, as section 4 discusses, we run full multimedia applications (such as `mpeg` and `jpeg`) rather than only look at small multimedia kernels. Contrary to our expectation, the performance results are not so favorable to the vector unit when a full application with all its complexity is considered.

3 A Superscalar Processor with a Vector unit

In this section we describe in detail our proposed architecture, looking first at the datapath design, including the vector register file and the vector unit, and then describing the memory hierarchy design. We also introduce a new cache design, which we call a “vector cache”.

3.1 Datapath

Figure 1 shows the main components of the datapath for the proposed architecture. Essentially, the architecture is modeled after a MIPS R10000 processor with the addition of a vector register file connected to both the scalar and the floating point units.

The general operation of the pipelines closely follows the R10k. Instructions are fetched and sent to the decode stage where they are renamed. There are four rename tables, one for integer registers, one for floating point registers, one for vector registers and one for mask registers. Once renamed, instructions go to the appropriate queue where they wait until their operands are ready and then arbitrate for a free functional unit. Instructions in every queue can execute out-of-order. We refer the interested reader to [9] for a more in-depth discussion of the architecture.

The major addition to the superscalar processor is the vector register file (VRF) and its connections to the available functional units. Three major parameters must be chosen concerning the vector registers: (1) number of logical vector registers, (2) number of physical vector registers for renaming and (3) length of each individual vector register.

We expect that a reasonable ISA will provide 32 programmer visible vector registers, to give enough flexibility to the compiler to schedule vectorized code. Our previous studies [9] show that the number of physical registers required to support renaming must be at least twice the number of logical registers. Consequently, we settled on 64 physical vector registers. On the other hand, the length of each vector register, has to be chosen carefully. Given the area constraints of a microprocessor, we can not make our vector registers arbitrarily large. Moreover, our studies show that not all multimedia applications can take advantage of vector lengths larger than 8 or 16 elements.

For the purposes of this paper, we have chosen to study two different vector lengths: 16 and 128 elements. A vector length of 16 elements implies that the full vector register file will use 8 Kbytes of storage ($64 \times 16 \times 8$). We feel this size is a reasonable choice for current and near-future technology. However, in the mid-term, as integration increases, larger vector registers might very well fit on-chip. Therefore, we will also study a vector register file where each vector register has 128 elements (requiring a total of 64 Kbytes of storage).

The vector instructions can operate both on integer and floating point data. Therefore, the functional units of the original machine are shared also by the vector unit. In the simplest design, (shown in figure 1) the vector unit can have a maximum of two vector instruction in progress. As we scale up the architecture, we add *parallel lanes* to all functional units. This can be achieved with relatively simple logic by replicating the functional units, splitting each vector register across each lane and assigning each functional unit to a certain lane, as shown in figure 2. The different elements of a vector register are interleaved across lanes, allowing all lanes to work independently from each other. In contrast, the figure also shows how this kind of scaling would be done in a traditional superscalar design. Note the sharp difference in the complexity of each organization's crossbars.

3.2 The Vector Cache

In this subsection, we discuss the design of a cache-based memory system tuned to our vector unit. The goals of our cache design are: first, to provide high bandwidth to the vector register file. Second, to allow this bandwidth to scale up as we scale the number of functional units. Third, due to the very distinct natures of the scalar memory stream and the vector memory stream, to minimize the conflicts between them. Fourth, guarantee that the processor cycle time is not in jeopardy due to the inclusion of a high bandwidth port to the vector register file.

The bandwidth problem

The cache hierarchy approach has been the solution for superscalar architectures to overcome the ever increasing speed gap between processor and main memory. Current superscalar processors integrate at least two levels of cache hierarchy. Typically, we can find a small and fast on-chip L1 cache, backed up by a large (1–4MB) off-chip L2 cache. However, advances in logic integration will allow next generation superscalar processors (such as the Alpha 21364 [10]) to include both caches on-chip.

As the number of instructions executed in parallel increases, data caches with higher bandwidth will be required. To obtain high bandwidth from a cache, several requirements must be met. First, multiple TLB translations must be made in parallel. Second, the cache must provide multiple access ports. For a small number of ports (say 2 or 3) this is usually done by implementing a true multi-ported cache (whether time multiplexed as in the Alpha 21264 or with multiple cache copies as in the 21164). However, research shows that for large number of ports, say 4 to 16, this is not feasible and alternative designs using multiple banks or hybrids of multi-bank and multi-port must be used [11, 12].

Thus, no obvious solution seems to be available for scaling current superscalar processors up to issuing four or more memory accesses per cycle. Even though accesses tend to exhibit high spatial locality, the out-of-order nature of the references makes it difficult to take advantage of this locality in order to maximize the effective rate of data accesses. In many cases, the spatial locality often translates into bank collisions in multi-bank data caches, decreasing the effective bandwidth of the cache ports.

Opposed to scalar memory traffic, vector memory accesses offer an easy way to exploit the spatial locality of references. A vector access is characterized by three different parameters: initial address, vector length (that is, number of references of the vector access), and vector stride (that is, the distance between consecutive references inside the vector access).

If the vector access stride is 1, then we have maximum spatial locality and an opportunity to take advantage of the bandwidth available on chip. Considering the density of current chip technology, why not have a very wide path from the cache into the vector registers ? When a 16-element stride-1 vector request is launched, one can easily envision the first level cache delivering full cache lines to the processor as opposed to delivering a single element per cycle. Current superscalars can not take advantage of this, because each cache access is independent of all other cache accesses. Meanwhile, a vector access typically needs a single TLB translation[1] and then accesses multiple elements from the cache.

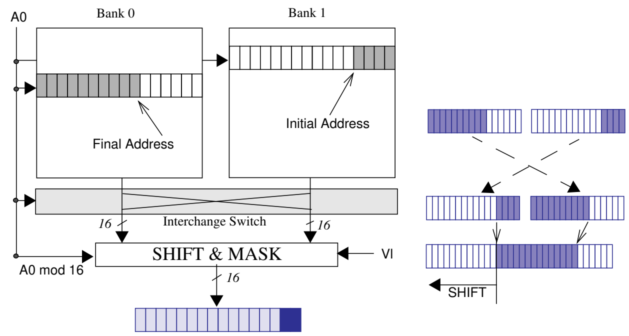

Figure 3: The dual bank structure of the vector cache.

The vector cache

We have designed a high bandwidth data cache targeted at accessing stride-one vector accesses by loading a whole cache line instead of individually loading the vector elements. Then, a shift and mask logic correctly aligns the data, eliminating residuals. The vector cache is able to achieve high bandwidth ratios for stride-one accesses, even for unaligned addresses. The two main hot spots of this design, that will be deeply addressed in following sections, are the alignment logic and the impact of strided vector accesses in overall performance.

This cache design heavily borrows from the ideas introduced in [3, 2] where a similar cache architecture was proposed to deal with the problem of unaligned accesses in instruction caches. In these previous papers, however, only the load path to the cache was under consideration, since the cache was used only to fetch instructions. Our design, by contrast, must include a store path to write data into the cache.

Figure 3 shows the proposed design. The cache is two-bank interleaved so that two consecutive cache lines can be accessed. Therefore, a whole vector access overlapped over two different lines can be accessed simultaneously. This scheme requires three different logic blocks to perform the data alignment: *an interchange switch* , since we may need to swap the position of the two lines, *a shift*, to align the lines accessed to the specified initial address, and a logic to *mask* the unused data based on the vector length of the required vector access.

Figure 4 shows the corresponding datapath that allows loading and storing from the vector cache. For load operations, the cache lines loaded from the two banks are first swapped conveniently, and then shifted and masked. Once the vector has been accessed, it is written into the appropriate vector register using a wide bus that sends all vector elements in parallel. In [2] it is claimed that a load could effectively complete in a single cycle, because: (1) the left-shift amount (A_0 mod line size) is known at the beginning of the cache access cycle, and has the entire cache access to fanout to the shifter datapath, and (2) assuming a transmission gate barrel shifter, data passes through only one transmission gate delay.

[1] When crossing page boundaries two translations would be made

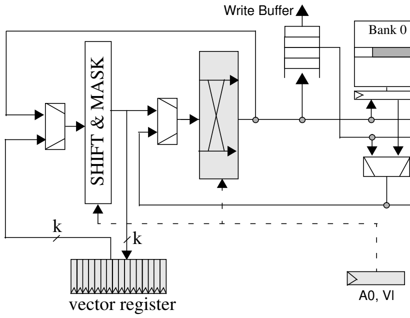

Figure 4: Load/Store paths for the vector cache.

Unfortunately, this is not possible for store operations. We need first to shift and mask the data from the vector register, and then swap the partition conveniently. Once these operations have been made, the store over the cache banks can be effectively made. Therefore, we will assume that the required time to perform a store operation is two cycles.

The write buffer

The write buffer for our vector cache has been designed following the results presented in [13]. We have implemented a coalescing write buffer with a width equal to the cache line size, since it has to be able to insert both scalar and vector accesses. The retirement order is FIFO, as usual, except for those cases where a load hits a write buffer entry. In those cases, we have chosen a flush item only policy combined with a data bypass mechanism to reduce the latency of the access. The normal retirement is produced when the number of write buffer entries is greater or equal than X (retire-at-X policy), where X is half the depth of the write buffer. Our modeled write buffer has a single port. Therefore, in the event of a cache line miss, two cycles are required to test if the desired data is located in the write buffer (since a vector access may be overlapped over two different write buffer entries).

The non-blocking mechanism

Intuitively, to support bandwidth hungry codes such as numerical and multimedia applications requires a significant degree of non-blocking operation in the cache.

We have designed a MSHR table specially targeted at our proposed vector cache. Its most salient feature is that a single vector access may cause *two* cache line misses (since we assume that the maximum vector lengths equals the size of a cache line and since we also allow misaligned accesses). Therefore, each entry of the MSHR needs to hold miss state information for *two* cache lines. Scalar accesses and vector accesses with strides larger than the cache line size, only make effective use of half of each MSHR entry.

When a line arrives from the upper level of the memory hierarchy, it clears the "pending line" bit in the appropriate MSHR entries. An entry will only be freed when none of its two line descriptors are asserted. In case of having more than one entry to be retired, the retirement is prioritized based on the age of every entry.

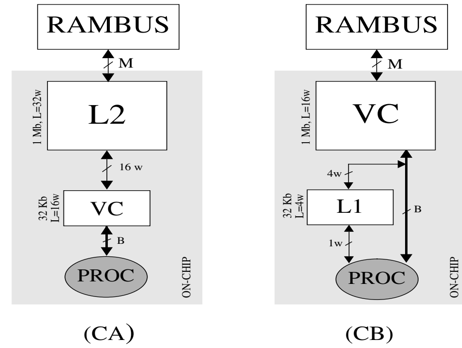

Figure 5: The vector cache data path.

3.3 Cache Hierarchy

After describing the basic principles of the "vector cache" now it is time to decide whether this vector cache should act as a first level or as a higher level cache.

The most naive approach would simply make the vector cache a first level cache and provide a wide path between this cache and the vector registers. This first design, that we refer to as the (**CA**) cache model, is shown in figure 5. Both vector and scalar accesses are made to the L1 cache using a $B\times$ 64-bit words wide bus. A stride-1 vector access of VL elements that hits in the cache uses the full bus bandwidth and takes just $\lceil VL/B \rceil$ cycles to complete. Scalar accesses and non-stride-1 vector accesses, however, can not take advantage of the full width of the bus and simply load a maximum of one word per cycle. Finally, the L2 data cache, which is assumed on-chip, is a conventional cache that will be connected by a bidirectional bus to an external RAMBUS [14] controller. Since we would like our model to be extensible to multiprocessing, our simulators faithfully model all the coherency traffic required to maintain the *inclusion* property[2].

There are several disadvantages to having the vector cache at the first level:

- First, the complexity of the vector cache could jeopardize the processor cycle time.
- It would be difficult to extend the vector cache design to a multiported configuration.
- Storing to the vector cache takes two cycles, which could potentially compromise scalar performance.
- The vector working set is expected to be several times larger than the capacity of the L1 cache yet, in many cases, might fit in a large L2 cache. If this is the case, then constantly missing in L1 to find data in L2 incurs control overheads that could be cut down if a direct path to L2 was provided.

All these problems can be overcome in the (CB) model shown also in figure 5. The CB model has a conventional single-ported L1 data cache with small lines (32 bytes). Scalar accesses are made to the L1 conventional data cache at a maximum rate of 1 element per cycle. On the other hand, vector accesses bypass the L1 conventional cache to access

[2] The data contained in L1 must be a strict subset of the data contained in L2. For instance, if a 128-byte line is evicted from L2, then care must be taken to invalidate, if present, all its four 32-byte sub-lines in L1.

directly the L2 vector cache. These accesses are performed at a maximum rate of B elements when the stride is one, and at a rate of 1 element per cycle for any other stride. The L1 data cache uses the wide path for its refills. Of course, the problem when bypassing the L1 is data coherency, an issue that we consider in the following subsection.

The (CB) model effectively "decouples" the scalar working set from the vector working set. Vector accesses pay a slightly larger latency but, in return, hit much more often in the L2. Since vector code is more latency tolerant than scalar code, this increase in latency should have minimal impact on performance, while the better hit rate should provide a very large effective bandwidth to the vector unit.

Coherency Protocol in CB Memory System

When a vector access is made to the L2 bypassing the L1, two integrity problems must be taken care of. First, as was the case in CA, we must obey the *inclusion* principle. Second, data coherency between L1 and L2 must be maintained.

Obeying the inclusion principle implies that, whenever an L2 cache line is replaced "invalidation" commands are sent to the L1 to flush the appropriate lines. Observing the inclusion principle allows us to design a very simple coherence protocol to attack the data coherency problem. Every cache line (in L1 and L2) has an *Exclusive* bit associated with it. A certain cache line can be in exclusive state either in the L1 or in the L2 caches, but never in both caches simultaneously. The idea is that a scalar access to the L1 or a vector access to the L2 can only use the data in the line being accessed if the line has the exclusive bit set. If the bit is not set, we have two different situations.

If a scalar access finds a line without the exclusive bit, it means that the line has been read or written by the vector unit using the bypass path. Therefore, the L1 requests to the L2 the line and acquires the exclusive state.

If a vector access finds a line without the exclusive bit, it means that, potentially, part of the line might have been written in the L1 by a scalar access (this could be a true hazard or a false sharing situation). The L2 sends a 'flush' command to the L1 to get the most recent copy of the line (acquiring the exclusive state) and then serves the access.

By no means this protocol is the most efficient one, but we chose it for its simplicity.

3.4 RAMBUS main memory

The *Direct Rambus DRAM* (RDRAM) [14] appears as the memory architecture of choice for the new generation of PC's and workstations, due to its dramatic improvements in bandwidth compared to current standard SDRAM memories.

We have modeled a 128MB *Rambus* main memory system which contains a *RDRAM* controller driving 8 *Rambus* chips and 16 memory banks per *Rambus* chip (128 banks total). This structure is replicated to get the desired 64-bit word at 400Mhz (the cpu is assumed to work at this frequency). The bus connecting the L2 to the Rambus is a 128-bit wide, bi-directional data bus running at 200Mhz, resulting in a memory bandwidth of approximately 3,2 Gb/s.

The L2 cache controller sends cache line requests to the *Rambus* controller, which manages the *RDRAM* modules based on the RDRAM protocol. Two optimizations have been added to the Rambus controller: first, no *ROW* command is generated on a bank hit, and, second, a certain reorganization of queued requests is allowed in order to maximize throughput. Note that a store request from the cache will have to wait for any previous load operation to avoid collisions in the main bus.

4 Performance Results

In this section we will present performance results of our proposed superscalar architecture with a vector unit and compare it to a traditional superscalar core. We are interested in two different aspects of performance: first, scalability. We wish to understand how performance scales as we add more and more functional units to the processor. To this end, we will start by presenting performance figures assuming a perfect cache system with infinite bandwidth and 100% hit rate. Second, we will present results on the performance of our cache hierarchy, discussing the relative merits of the CA and CB cache organizations.

4.1 Benchmarks and Simulation Tools

We have studied the *Perfect Club*, *SPECfp92* and *Mediabench* [15] suites, and results shown here correspond to a set of selected benchmarks that conform a representative sample of the different behaviors found in these numerical and multimedia applications. Our workload is therefore composed of four *SPECfp92* programs (`swim256`, `hydro2d`, `nasa7` and `tomcatv`), three *Perfect Club* programs (`bdna`, `arc2d` and `flo52`) and three multimedia programs (`mpeg`, `epic` and `jpeg`). Two versions of each benchmark are required for this study: a purely scalar one and a version with vector instructions.

For the scalar version, each program was compiled using GCC v2.6.3 and then simulated using the *SimpleScalar Tool Set v3.0* [16]. For the scalar programs, we always simulate 1500 million graduated instructions. For the vector enhanced version, the programs were compiled on a Convex C3400 with vectorization turned on. Some important changes have been made to the three multimedia benchmarks in order to get them to vectorize. These changes range from simple loop interchange techniques (as in `epic`) to a major rewrite of the *idct* algorithm (in `mpeg` and `epic`) following the standard specifications [17]. The resulting vector binaries are then processed using our own tracing and simulation environment, described in [9].

Comparing a scalar and a vector program in terms of IPC poses a significant problem. As it is widely known, to execute the same code, a vector machine uses much fewer instructions than a scalar machine (because each vector instruction specifies multiple operations) [18]. Therefore, using raw IPC as a performance measure would be meaningless.

The solution is as follows. First, each program is run to completion in pure scalar mode on a R10000 processor and, using the hardware counters, we collect its total number of graduated instructions. Second, all simulations done in vector mode are always run *to completion*. Then, the *Equivalent IPC* (EIPC) for each simulation of the superscalar+vector architecture (SSV) is defined as:

$$EIPC = \frac{Total\,MIPS\,R10000\,instr}{Total\,Execution\,Cycles} \tag{1}$$

The intuitive meaning of EIPC is "how well a superscalar machine should perform in order to match the performance of our proposed superscalar+vector architecture". This EIPC measure can be directly compared to raw IPC obtained using the SimpleScalar simulator.

	mem/flops	1x2	2x4	4x4	4x8	4x16	8x4	8x8	8x16	16x4	16x8	16x16	16x32
basic SS	fetch/issue	4	8	-	16	-	-	-	24	-	-	32	48
	# Mem. ports	1x1w	2x1w	-	4x1w	-	-	-	8x1w	-	-	16x1w	16x1w
	# FP units	2	4	-	8	-	-	-	16	-	-	16	32
	# INT units	2	4	-	8	-	-	-	16	-	-	16	32
	ROB size	32	64	-	128	-	-	-	256	-	-	256	512
	L/S queue	16	32	-	64	-	-	-	128	-	-	128	128
	BTB/Lbranch	0.5/4K	1/8 K	-	16/2 K	-	-	-	4/32 K	-	-	4/32 K	4/32 K
SS with Vector units	fetch/issue	4	-	4									
	# Mem. ports	1x1w	-	1 x 4 words			1 x 8 words			1 x 16 words			
	#VEC FP units	2x1	-	2x2	2x4	2x8	2x2	2x4	2x8	2x2	2x4	2x8	2x16
	# VEC INT units	2x1	-	2x2	2x4	2x8	2x2	2x4	2x8	2x2	2x4	2x8	2x16
	ROB size	64	-	64									
	L/S queue	16	-	16									
	BTB	64	-	64 entries									

Table 1: Configuration parameters for both architectures. The basic SS ports are 1 word wide, while the SS-vector machine has a single port with different widths (also measured in 64-bit words). In the SS-vector machine, there are 2 integer vector units and two vector floating point units with a varying number of lanes (from 1 to 16). Note that when running scalar operations only 1 lane is used; therefore, a maximum of 2 flops per cycle and 2 integer ops per cycle can be produced in pure scalar mode. *BTB/Lbranch* indicates number of BTB entries and size of the gshare prediction table.

4.2 Machine Configurations

We are interested in comparing the performance and scalability of a plain superscalar machine and of our proposed superscalar+vector architecture. Therefore, we look at a wide range of configurations, starting with today's typical superscalar machine (1 cache port and 2 flops per cycle) and going up to a hypothetical machine with 16 cache ports and 64 functional units (32 floating point plus 32 integer). Table 1 shows the architectural parameters of the two proposed systems for the different architectures.

4.3 Perfect Cache and Scalability

We start by exploring the performance and the scalability of our proposed architecture assuming that the cache is perfect, that is, that all memory accesses hit in the L1 and multiple accesses are served without conflicts. Scalar memory accesses all take one cycle. Vector memory accesses are served in $\lceil VL/B \rceil$ cycles, where B is the bandwidth of the single memory port (1, 4, 8 or 16 words).

Figure 6 shows the performance results for the two architectures. From these data we can clearly see that our proposed architecture achieves similar performance to the basic superscalar version for the 1x2 configuration. However, as we increase the number of processor functional units (or the number of lanes in the case of the vector functional units), the SS architecture performance levels off, whereas our proposed processor improves at a much higher rate. This is specially true for the 128-length vector registers configuration, but, nevertheless, our proposed 16-length vector registers configuration still achieves convincing performance gains with very modest investments on chip area (only 8KB for the vector register file).

On the other hand, the three multimedia benchmarks (`mpeg`, `epic` and `jpeg`) present modest performance results when compared to the numerical codes. This seems to contradict the common belief that multimedia workloads present huge levels of data level parallelism. Our analysis show that while this is true in raw kernels (such as the basic `idct` algorithm), several problems arise when one looks at complete applications.

First, the three multimedia programs are characterized by short vector lengths (between 8 and 16 words), which explains why the SSV-128 and SSV-16 lines are completely overlapped. Although we could have used techniques such as loop interchange in simple kernels in order to increase the vector length of the programs, this is not really possible in real programs because of the functional structure of the programs or because doing so would break the standard specifications.

Looking at each program individually, we can further point out the following reasons for their relatively low performance. `Mpeg` exhibits convincing performance results for the more aggressive configurations. However, the performance is not as good as expected due to artificial recurrences in the *motion estimation* algorithm, which could be eliminated by hand coded optimizations. `Epic` is characterized by a low vectorization percentage (about 45%), and therefore, as we scale the system, the non-vectorizable fraction of code becomes the real performance bottleneck due to the lack of enough issue rate and resources. `Jpeg` is a dramatic example of the mismatch between standard an optimized programs. The scalar benchmark has highly tuned algorithms to perform the *idct*, the *RGB to YCC color conversion* and the *Coefficient quantization* functions in superscalar architectures. These algorithms include table lookups to avoid performing multiplications, mathematical optimizations that reduce the number of divisions and a plethora of other well-known techniques that reduce the overall cost of the application. Meanwhile, the vector version of the program, based on the standard `jpeg` specifications without all these optimizations, executes more than twice the number of operations than the scalar version, which explains the lower EIPC results.

Overall, however, the scalability of the SSV architecture is much better than the plain superscalar machine, and, furthermore, this scalability is achieved using a much lower

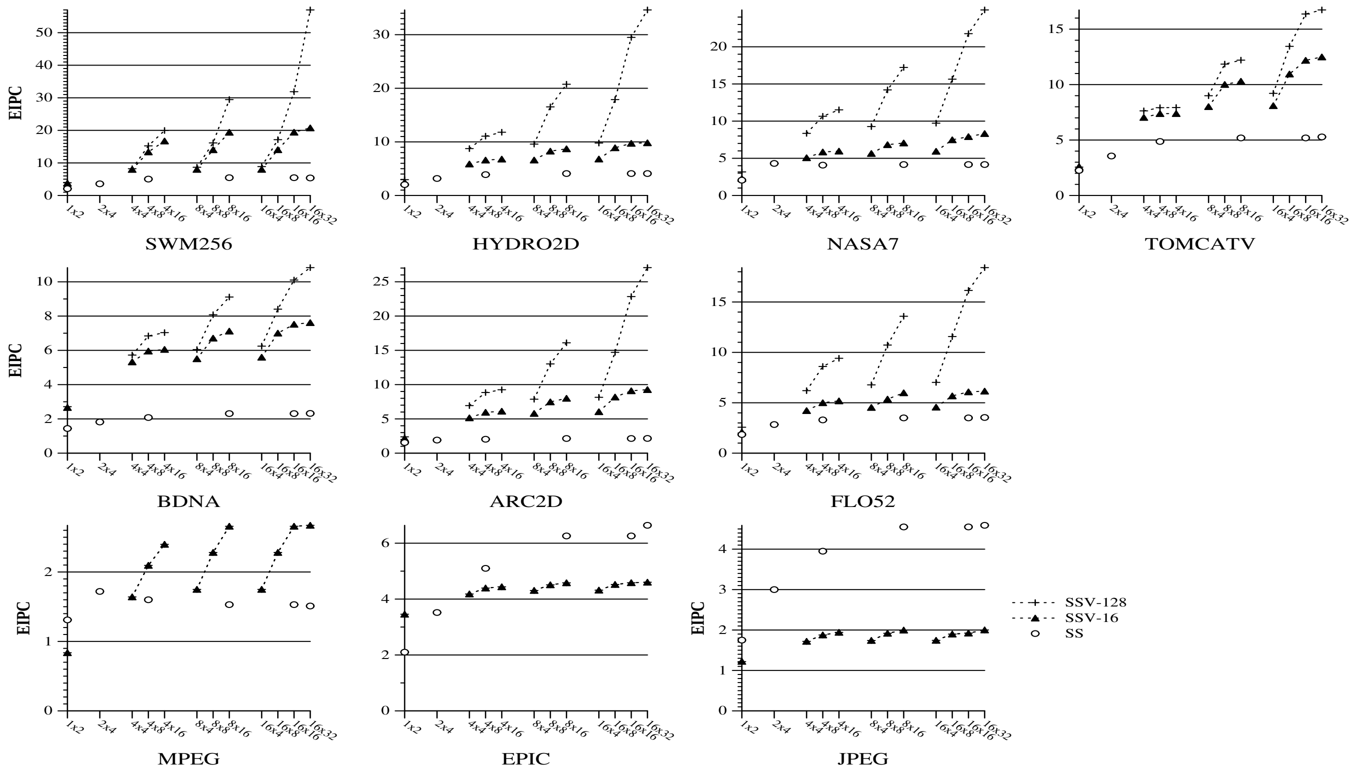

Figure 6: Performance results with an ideal memory system for: a basic superscalar processor (SS); an enhanced superscalar processor with 128-length vector registers (SSV-128); and an enhanced superscalar processor with 16-length vector registers (SSV-16).

control complexity. With a 4-way out-of-order engine we are able to successfully feed up to 32 floating point units and sustain a large fraction of peak performance.

4.4 Real Cache

We now turn to the question of the performance under a real memory hierarchy. We will study the two cache architectures described in section 3.3, CA and CB, and look at their performance studying total memory traffic, hit ratio and, finally, IPC results. We will also look at bandwidth issues at the RDRAM level, for those programs that are not very cache-friendly. The general parameters of the two memory hierarchies under study are summarized in table 2. We have tuned these parameters by doing several studies about size of the cache, size of the cache line, number of sets, associativity, and write and allocate policies.

	CA		CB	
	L1	L2	L1	L2
Size	32KB	1MB	32KB	1MB
# Sets	256	2048	1024	4096
Line Size	128B	256B	32B	128B
Associativity	1	2	1	2
Write Policy	WT	WB	WT	WB
Allocate Policy	NWA	WA	NWA	NWA
Latency (cycles)	1	4	1	4
Ld/St cycles	1/2	1/1	1/1	1/2
MSHR entries	8	8	8	8
WB depth/retire	8/4	8/4	8/4	8/4

Table 2: Cache Hierarchy Parameters: WT=write-through, WB=write-back, WA=write-allocate, NWA=no-write-allocate. *Ld/St cycles*, cycles required for a Load/Store operation. *WB depth/retire* number of Write Buffer entries/retire-at-X policy.

4.4.1 Memory Traffic Filtered

The first question we are concerned with is whether our workloads will take advantage of the caches or not. Traditionally, it has been claimed that vector workloads have low spatial and temporal locality, that this locality is well exploited by the large vector registers and that using a cache only gets in the way of memory accesses that end up accessing main memory regardless [19].

To answer this question, we have measured the percentage of 64-bit words that are filtered by the cache hierarchy, that is, the fraction of words requested by the processor that are serviced by either the L1 or the L2 and that, therefore, do not access the RDRAM main memory.

Figure 7 presents results for both cache models (CA and CB). It is important to note that the amount of on-chip memory used in both models is the same. The only difference is in how/where vector and scalar data are allocated.

For six programs the caches are very effective, servicing more than 95% of all processor requests. This is because the working set of these programs fits in the cache hierarchy. For these programs, only minor differences can be seen between the CA and CB models. The remaining benchmarks (`swm256`, `nasa7`, `tomcatv` and `arc2d`) have very large working sets and, therefore, the cache hierarchy only services around 50% of all processor requests. Interestingly, `swm256`, `nasa7` and `tomcatv` show a large difference between the CA

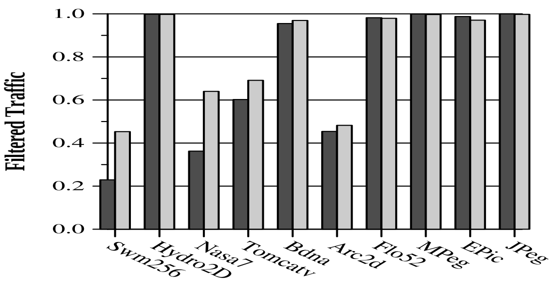

Figure 7: Traffic filtration

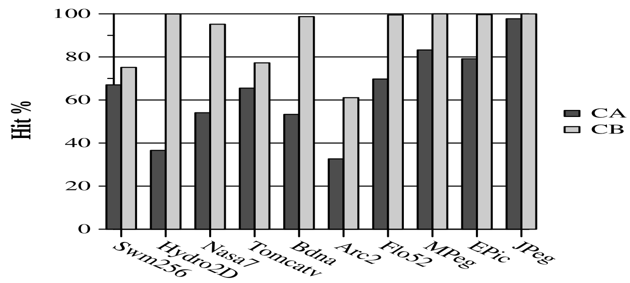

Figure 8: Full hit percentage for vector accesses

and CB models. Two different reasons can be pointed out. `Nasa7` has many large strides, which trash the cache heavily. Therefore the CB model that has double number of lines in L2 performs better. For the other programs, the difference is explained by the allocation policy. For them, no-allocate is a much better policy.

4.4.2 Cache Hit Rate

While the overall traffic filtering effect of CA and CB might be similar, the two models have distinct properties regarding hit rate. Since a vector access may request between 1 and 16 data elements, we say we have a *full hit* when all elements requested by the vector access hit in the cache. From the point of view of performance, maximizing the rate of full hits is critical. Note that partial hits (even if just one element is missing) are as bad as a full miss, since the full miss latency will be paid for that single element missing. Therefore, a good indicator of final performance is how many full hits we have for vector accesses.

Figure 8 shows the *full hit* rate for both models CA and CB. Clearly, the *full hit* rate is better in the CB memory model. This was expected since in CB, all vector accesses are served by a 1MB vector cache while in CA vector memory accesses are served by a 32KB vector cache. Will this higher hit rate provide better performance ? The L2 vector cache is 4 cycles away from the processor, while L1 is just 1 cycle away. A back-of-the-envelope calculation would tend to favor the CA model despite its lower hit rate.

However, two factors combine to make CB a better alternative cycle-wise. First, vector codes do have some inherent latency tolerance. Therefore, the longer latency might become a non-issue if, indeed, the CB can provide a higher effective sustained bandwidth. Second, communication between L1 and L2 due to L1 misses is expensive and cannot always be overlapped with other cache requests. For example, in the CA model, an unaligned vector miss to L1 will *require* by definition two different lines from L2, therefore, at the very least, bandwidth halves. Therefore, the CA model, where we have many L1 misses, delivers a lower effective bandwidth than CB.

4.4.3 Equivalent IPC

Finally, we turn to performance results in terms of IPC. Figure 9 compares the performance of the two cache models, CA and CB, against the performance of the SSV-16 machine with perfect cache.

From the results in figure 9, we can see that for eight of the ten benchmarks, the CB model clearly outperforms the CA model. On the other hand, for two multimedia benchmarks (`jpeg` and `mpeg`) , the CA model outperforms the CB model. This is due to the high number of reduction operations (such as adding all elements of a vector) which are a very common operation in image applications. The results of these reductions are individually written to their respective matrix positions using a scalar store. Therefore, in the CB model, these data items go the the L1 cache. Later, the vector unit requests a bunch of these results in vector mode to the L2, only to find that there is a coherency problem with L1. The L2 must *flush* and *invalidate* data from the L1 to service the vector request, resulting in a severe loss of performance. We have evaluated the L2 *cache stall time* due to coherence operations. While for the rest of the programs the stall time is less than 1% of the whole execution time, we found a stall time of 15% of the overall execution time for the `mpeg` benchmark and a stall time of 35% of the execution time for the `jpeg` benchmark.

Comparing the real cache models with the ideal cache model, we can divide the programs into two different groups:

In the first group, (`hydro2d`, `flo52`, `bdna`, `mpeg` and `epic`), the cache performance is very close to the ideal performance. What is more important, it can be clearly seen that the performance scales well as we increase the number of lanes per functional unit and the width of the memory port. Note that all of the five programs are characterized by fitting in L2.

The second group, (`swm256`, `nasa7`, `tomcatv`, and `arc2d`), is characterized by poor CB performance results when compared to the ideal memory system. These programs do not fit in cache and their hit rate is very low (see again figure 8). Overall performance is mostly limited by main memory bandwidth. In section 4.5 we will further explore this issue by scaling the bandwidth of the Rambus system. The performance results of `nasa7` are aggravated by the fact that only 46.8% of the accesses have stride one (i. e. only 46.8% of the accesses can be served at the maximum port bandwidth). This is the case of `jpeg`, where the CB results are close to the ideal ones for conservative configurations but scale poorly as we simulate more aggressive configurations. Again, the explanation lies in the the low percentage of the vector accesses with stride one.

We can not conclude this section without discussing why we have not presented results for a realistic plain superscalar implementation *without* vector units. The answer is simple: there exists no such thing as a ‘realistic’ superscalar that can fetch/issue/retire 48 instructions per cycle, has a 16-port L1 cache, can make 16 TLB translations per cycle, etc. The SimpleScalar simulator does not model collisions when doing multiple cache accesses, does not model a write buffer, does not model a limited MSHR file and can not simulate a multi-banked cache configuration. Essentially, SimpleScalar behaves close to an infinite bandwidth cache, which is not acceptable to compare to our highly detailed

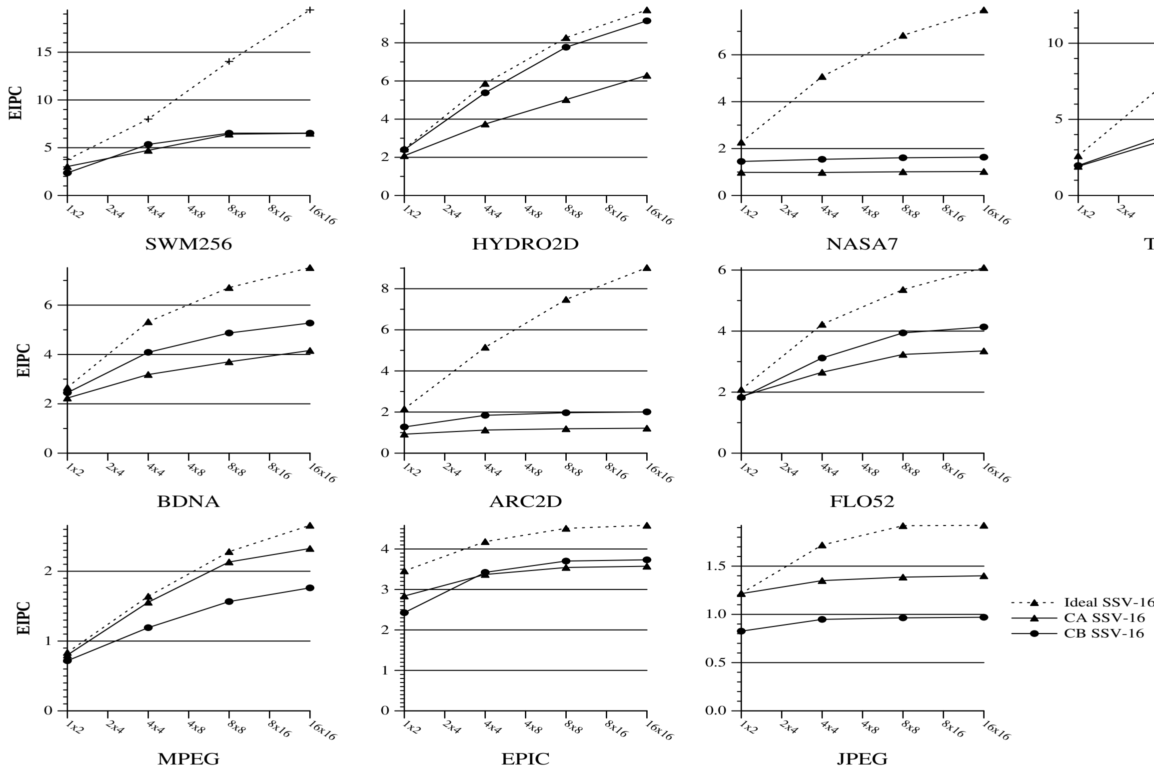

Figure 9: Performance study of CA and CB memory models

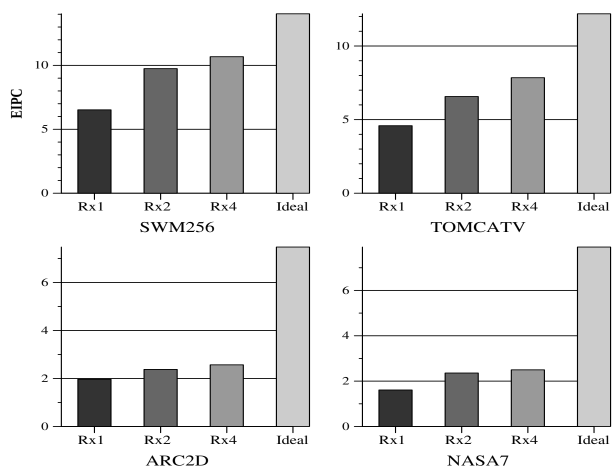

Figure 10: Performance results increasing the memory bandwidth

cache simulations for the SSV architecture.

4.5 Increasing Bandwidth

As we just discussed, four programs are limited by main memory bandwidth (`nasa7`, `swm256`, `tomcatv` and `arc2d`). For these programs, a good solution is to increase the RDRAM bandwidth. Other possibilities would be to add hardware prefetching at the L2 level to advance as much as possible requests for vector streams.

We have considered a fixed configuration (4 lanes/unit, memory port of width 8) and measured EIPC for three different RDRAM bandwidths. The *RDRAM* main memory model leverages the standard bandwidth of 3.2 Gb/s, the *RDRAMx2* main memory model leverages 6.4 Gb/s, and the *RDRAMx4* leverages a peak bandwidth of 12.8 Gb/s. Finally, we have included the performance for the ideal memory system as an upper bound.

Figure 10 presents the performance results when we increase the RDRAM bandwidth for the mentioned programs. As expected, we see two different behaviors. First, `swm256` and `tomcatv` increase their performance linearly as we increase RDRAM bandwidth. Both programs overflow in L2 but have a high spatial locality (almost 100% stride-1). Therefore, the RDRAM can easily serve their requests and the more bandwidth available in the L2-to-memory bus the better the performance. The other two programs improve very slightly but can not take advantage of the extra bandwidth due to its large strides. In each RDRAM request, at most one or two words of useful data are fetched. Therefore, the fact that the bus can deliver 2, 4 or 8 words per cycle does not matter to final performance.

5 Summary

In this paper we have proposed adding a vector unit with its vector register file to a conventional superscalar. The addition of the vector unit has minimal impact on the out-of-order core, yet provides high performance and scales very well as parallel lanes are added to the vector units.

We have designed a novel cache specially tuned to provide high bandwidth for vector accesses so that a vector register can be loaded in single cycle. We have analyzed two different cache alternatives: one based on a conventional cache hierarchy (CA) and one focused on decoupling the vector data from the scalar data by directly accessing the L2 cache for vector accesses (CB). We have shown the clear advantages of the second model over the first: scalar data fits into a small, fast level cache (L1), while vector data, larger but with better latency tolerance, is fit into a larger but slower level of cache (L2), minimizing the communication overhead between the two levels of cache. We have shown also the implementation advantages of the CB model, since it does not jeopardize the cycle time and allows the the first level cache to be multiported.

We have evaluated our proposed architecture with an ideal cache, and we have compared its performance results with a basic superscalar model. We have demonstrated that our proposal achieves scalable performance without jeopardizing the reliability of the implementation. Our proposal does not increase fetch and issue, the complexity of the functional units interconnection network is kept at a minimum, the window size is constant and the number of ports to cache and TLB is fixed and small.

Finally, we have evaluated our proposed architecture with a highly detailed and cycle-accurate cache model, and we have shown that for those programs not limited by main memory bandwidth, we achieve scalable performance. This is even better if we consider that this scalability just requires widening the processor-to-L2 bus. By contrast, similar levels of scalability are clearly beyond the implementation capabilities of current and near future plain superscalar architectures.

The performance results presented so far do not take into account cycle time. While a quantitative evaluation of cycle time is beyond the scope of this paper, it is interesting to note that in the superscalar+vector machine, control logic has been kept exactly the same across all configurations. This is possible only thanks to the properties of vector instructions. Scaling the number of functional units *does not* require adding more issue slots or larger instruction queues. Therefore it is very likely that the SSV machine will have a faster cycle time than a hypothetical equivalent plain superscalar and, at the same time, the SSV will require less area.

As a conclusion, our claim is that adding a vector functional unit to a basic superscalar processor is a suitable alternative to break the current ILP barriers found in numerical and multimedia applications. Scalable performance can be achieved without jeopardizing the cycle time and with moderate modifications to a traditional out-of-order core.

Acknowledgments

We wish to thank Jim Smith for many interesting discussions and helpful hints. We also thanks CEDEX for the use of their Convex C4 facility. Francisca Quintana is supported by the Fundacion Universitaria de Las Palmas under grant 23/97 of the INNOVA program. Jesús Corbal is supported by Direccio General de Recerca de la Generalitat de Catalunya under grant 1998FI-00260. This work has also been supported by the Ministry of Education of Spain under contracts CICYT TIC-0429/95 and TIC98-0511-C02-01, and by the CEPBA.

References

[1] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 206–218, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[2] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[3] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News*, 23(2), May 1994.

[4] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, pages 42–50, August 1996.

[5] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, pages 10–20, August 1996.

[6] Altivec Technology. Technical Report http://www.mot.com/SPS/PowerPC/AltiVec/, Motorola, Inc., 1998.

[7] 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, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[8] Corinna G. Lee and Mark G. Stoodley. Simple Vector Microprocessors for Multimedia Applications. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 330–335, Dallas, Texas, December 1998. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[9] Roger Espasa, Mateo Valero, and James E. Smith. Out-of-order Vector Architectures. In *MICRO-30*, pages 160–170. IEEE Press, December 1997.

[10] Peter Bannon. Alpha 21364: A Scalable Single-chip SMP. Technical Report http://www.digital.com/alphaoem/microprocessorforum.htm, Compaq Computer Corporation, 1998.

[11] T. Juan, J. Navarro, and O. Temam. Data caches for superscalar processors. In *Conference Proceedings, 1997 International Conference on Supercomputing*, pages 60–67, Vienna, Austria, Jul 7–11, 1997. ACM SIGARCH.

[12] J.A. Rivers, G.S. Tyson, T.M. Austin, and E.S. Davidson. On high-bandwidth data cache design for multiple-issue processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 46–56, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[13] Kevin Skadron and Douglas W. Clark. Design issues and tradeoffs for write buffers. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 144–155, San Antonio, Texas, February 1–5, 1997. IEEE Computer Society TCCA.

[14] Richard Crisp. Direct rambus technology: The new main memory standard. *IEEE Micro*, 7:18–28, November/December 1997.

[15] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[16] Doug Burger and Todd Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report UWCS-1342, University of Wisconsin, June 1997.

[17] Jerry D. Gibson, Toby Berger, Tom Lookabaugh, Dave Lindbergh, and Richard L. Baker. *Digital Compresion for Multimedia*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.

[18] Francisca Quintana, Roger Espasa, and Mateo Valero. A Case for Merging the ILP and DLP Paradigms. In *Euromicro Workshop on Parallel and Distributed Processing*. IEEE Computer Society Press, January 1998.

[19] L.I. Kontothanassis, R. A. Sugumar, G. J. Faanes, J. E. Smith, and M. L. Scott. Cache performance in vector supercomputers. In *Proceedings of Supercomputing'94*, Washington D.C., November 1994. IEEE Computer Society Press.