Tarantula: A Vector Extension to the Alpha Architecture

Roger Espasa, Federico Ardanaz, Joel Emer[‡], Stephen Felix[‡], Julio Gago, Roger Gramunt,
Isaac Hernandez, Toni Juan, Geoff Lowney[‡], Matthew Mattina[‡], André Seznec[‡*]

Compaq–UPC Microprocessor Lab
Universitat Politècnica Catalunya
Barcelona, Spain

[‡]Alpha Development Group
Compaq Computer Corporation
Shrewsbury, MA

Abstract

Tarantula is an aggressive floating point machine targeted at technical, scientific and bioinformatics workloads, originally planned as a follow-on candidate to the EV8 processor [6, 5]. Tarantula adds to the EV8 core a vector unit capable of 32 double-precision flops per cycle. The vector unit fetches data directly from a 16 MByte second level cache with a peak bandwidth of sixty four 64-bit values per cycle. The whole chip is backed by a memory controller capable of delivering over 64 GBytes/s of raw bandwidth. Tarantula extends the Alpha ISA with new vector instructions that operate on new architectural state. Salient features of the architecture and implementation are: (1) it fully integrates into a virtual-memory cache-coherent system without changes to its coherency protocol, (2) provides high bandwidth for non-unit stride memory accesses, (3) supports gather/scatter instructions efficiently, (4) fully integrates with the EV8 core with a narrow, streamlined interface, rather than acting as a co-processor, (5) can achieve a peak of 104 operations per cycle, and (6) achieves excellent "real-computation" per transistor and per watt ratios. Our detailed simulations show that Tarantula achieves an average speedup of 5X over EV8, out of a peak speedup in terms of flops of 8X. Furthermore, performance on gather/scatter intensive benchmarks such as Radix Sort is also remarkable: a speedup of almost 3X over EV8 and 15 sustained operations per cycle. Several benchmarks exceed 20 operations per cycle.

1. Introduction

As CMOS technology progresses, we are able to integrate an ever-growing number of transistors on chip and the interconnect within the die can be used to achieve massive amounts of bandwidth from on-chip caches. The challenge architects face is governing effectively this large amount of computation resources that CMOS makes available.

If a large number of resources such as functional units, memory ports, etc. are to be controlled individually, the amount of real estate devoted to control structures grows non-linearly. The lack of control aggregation results in long, slow global wires that limit overall performance and hinder scalability. Both wide superscalar and VLIW architectures suffer from this problem, due to the small granularity of their instructions. Scalar instructions typically only encode one, sometimes two, operations to be performed by the processor (an add, an add and a memory access, a multiply-add, etc.). Yet, the number of control structures required to execute each instruction is growing with processor complexity and frequency.

In contrast, vector ISAs provide an efficient organization for controlling a large amount of computation resources. Vector instructions offer a good aggregation of control by localizing the expression of parallelism. Furthermore, vector ISAs emphasize local communication and provide excellent computation/transistor ratios. These properties translate in regular VLSI structures that require very simple, distributed control. Combining the parallel execution capabilities of vector instructions with the effectiveness of on-chip caches, good speedups over a conventional superscalar processor can be obtained with a main memory bandwidth on the order of 1 byte per flop. Since increasing memory bandwidth is one of the most expensive pieces of a system's overall cost, maximizing the computation capabilities attached to a given memory bandwidth seems the right path to follow. In particular, a large L2 cache can provide tremendous bandwidth to a vector engine. Such an L2 will naturally have a long latency. Nonetheless, the latency-tolerant properties of vector instructions offsets this downside while taking advantage of the bandwidth offered. The

[*]While on sabbatical from INRIA

main difficulties in this mixture of vectors and caches are non-unit strides and gather/scatter operations.

Tarantula is an aggressive floating point machine targeted at technical, scientific and bioinformatics workloads, originally planned as a follow-on candidate to the *EV8* processor [5, 6]. Tarantula adds to the *EV8* core a vector unit capable of 32 double-precision flops per cycle. The vector unit fetches data directly from a 16 MByte second level cache with a peak bandwidth of sixty four 64-bit values per cycle. The whole chip is backed by a memory controller capable of delivering over 64 GBytes/s of raw bandwidth.

Tarantula's architecture and implementation have a number of features worth highlighting. The vector unit can be tightly integrated into an aggressive out-of-order wide-issue superscalar core with a narrow, streamlined interface that consists of: instruction delivery, kill signals, scalar data and a handshaking protocol for cooperative retirement of instructions. The new vector memory instructions fully integrate into the Alpha virtual-memory cache-coherent system without changes to its coherency protocol. Our design shows that the vector unit can be easily integrated into the existing *EV8* physical L2 design simply by duplicating some control structures and changing the pipeline flow slightly. An address reordering scheme has been devised to allow conflict-free vector requests to the L2. This technique provides very good bandwidth on both unit and non-unit strides. *Tarantula* also provides high bandwidth for gather and scatter instructions and smoothly integrates them into the aggressive out-of-order memory pipeline. The vector execution engine uses register renaming and out-of-order execution and supports the SMT paradigm [18, 19] to easily integrate with the *EV8* core.

Finally, the question of commercial viability must also be addressed from two points of view: first, the cost of converting the installed customer base to a vector ISA and second, the market demand for a chip such as *Tarantula*.

Despite all its advantages, a vector ISA extension doesn't come without a cost. The major investment comes from software compatibility: new vector instructions require application recompilation and tuning to effectively use the on-chip vector unit. Compiler support to integrate tiling techniques with vectorization techniques is needed to extract maximum performance from the memory hierarchy. Also, our experience coding benchmarks for *Tarantula* shows that proper data prefetching is of paramount importance to achieve good performance.

Concerning market demand, we note the scientific computing segment still represents a multi-billion dollar market segment. Many of the applications in this domain exhibit a large degree of vector (data) parallelism and manipulate large volumes of data. This vector parallelism is not correctly accommodated by the small first level caches of current off-the-shelf microprocessors: poor cache behavior ruins performance. We believe performance of chip multiprocessors on vector codes will suffer from the same difficulty: processors will compete for the L2 and contention will lead to poor performance. So, although the rapidly growing workstation and PC market segment has oriented the whole computer industry (including the server industry) towards using standard off-the-shelf microprocessors, we believe the time is ready again for architecture specialization to better match application needs.

2. Instruction Set Extension

Tarantula adds to the Alpha ISA new architectural state in the form of 32 vector registers (`v0`..`v31`) and their associated control registers: vector length (`vl`), vector stride (`vs`), and vector mask (`vm`). Each vector register holds 128 64-bit values. The `vl` register is an 8-bit register that controls the length of each vector operation. The `vs` register is a 64-bit register that controls the stride between memory locations accessed by vector memory operations. The `vm` register is a 128-bit register used in instructions that operate under mask.

To operate on this new architectural state, 45 new instructions (not counting data-type variations) are added to the instruction set. The new instructions can be broadly grouped into five categories: vector-vector operate (VV), vector-scalar operate (VS), strided memory access (SM), random memory access (RM), and vector control (VC). Figure 1 presents the semantics of a representative instruction of the first four groups. The instructions semantics are straightforward extensions of the existing scalar Alpha instructions to allow operating on the vector registers.

A novel feature of the ISA is the approach to vector mask computation. To avoid long latency data transfers back and forth between the *Vbox* ALUs and the *EV8* scalar register file (a 20-cycle round-trip delay), vector comparisons store the resulting boolean vector in a full vector register. This allows coding complex if-statements without vector-scalar communication. For example, the translation of `A(i).ne.0.and.B(i).gt.2` would be:

```
vloadq     A(i)   --> v0
vloadq     B(i)   --> v1
vcmpne     v0, #0 --> v6
vcmpgt     v1, #2 --> v7
vand       v6, v7 --> v8
setvm      v8     --> vm
```

The final *setvm* instruction indicates that the `v8` register is to be copied into the `vm` register. Subsequent instructions that use the "under-mask" specifier will use this `vm` value. Since the `vm` register is renamed in the *Vbox*, the next mask value can be pre-computed while using the current one and

Group	Example	Semantics

VV	`VVADDQ Va, Vb, Vc`	`for (i = 0; i < vl; i++) {`
		`  Vc[i] = Va[i] + Vb[i]`
		`}`
		`for (i = vl; i < 128; i++) {`
		`  Vc[i] = <UNPREDICTABLE>`
		`}`

VS	`VSMULQ Va, Fb, Vc`	`for (i = 0; i < vl; i++) {`
		`  Vc[i] = Va[i] * Fb`
		`}`
		`for (i = vl; i < 128; i++) {`
		`  Vc[i] = <UNPREDICTABLE>`
		`}`

SM	`VLOADQ Vc, off(Rb)`	`S = GEN_RANDOM_PERMUT(0,VL-1)`
		`foreach i in S {`
		`  ea = Rb + off + (i * vs)`
		`  Vc[i] = MEM[ea]`
		`}`
		`for (i = vl; i < 128; i++) {`
		`  Vc[i] = <UNPREDICTABLE>`
		`}`

RM	`VSCATQ Va, Rb, Vc`	`S = GEN_RANDOM_PERMUT(0,VL-1)`
		`foreach i in S {`
		`  ea = Va[i] + Rb`
		`  MEM[ea] = Vc[i]`
		`}`

Figure 1. The four major instruction groups in *Tarantula*. For each group, a representative instruction is shown and its semantics presented.

compilers can interleave instructions from two separate if-then-else statements in a loop body.

Following the Alpha tradition, register `v31` is hardwired to zero. Therefore, vector prefetches, including gather and scatter prefetches, can be trivially crafted by using `v31` as the destination register. As in most Alpha implementations, page faults and TLB misses caused by vector prefetches are simply ignored.

Tarantula provides a precise exception model at the instruction level granularity. If a vector instruction causes a trap (TLB miss, divide-by-0, etc.), the system software will be provided with the PC of the faulting instruction, but no extra information on which element (or elements) within the vector caused the fault.

3. Tarantula Architecture

3.1. Block Diagram

Figure 2 shows a high-level block diagram of the *Tarantula* processor. The *EV8* core is extended with a vector unit (*Vbox*) that handles the execution of all new vector instructions. The *EV8* core performs all scalar computations and is also responsible for fetching, renaming and retiring vector instructions on behalf of the *Vbox*. Furthermore, *EV8* also notifies the *Vbox* whenever instructions must be killed due to mispredicts or faults. *Vbox* and the *EV8* core also cooperate for retiring vector memory writes.

Tarantula reuses the memory controller (*Zbox*) and the inter-processor router (*Rbox*) from the *EV8* design. We do however assume that in the 2006 time frame we should be

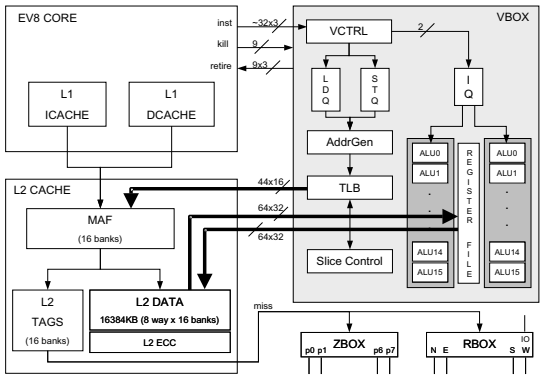

Figure 2. Block diagram of the *Tarantula* processor

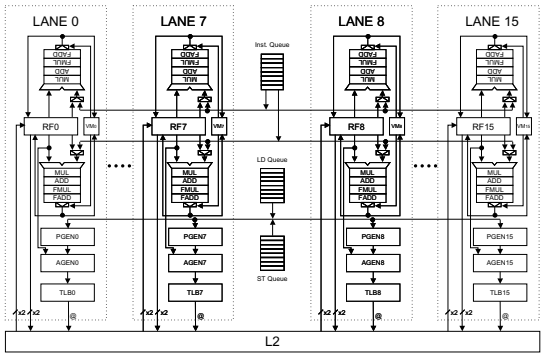

Figure 3. *Vbox* Lanes.

able to roughly quadruple main memory bandwidth from the current 12.8 GBytes/s to over 50 GBytes/s. While the exact memory technology to be employed is not currently known, for the purposes of this study we assumed the *Zbox* would control 32 RAMBUS channels, grouped as eight ports (roughly 64 GBytes/s raw bandwidth assuming 1066 Mhz parts).

3.2. Vector Execution Engine

The vector execution engine is organized as 16 lanes (see Figure 3). Each lane has a slice of the vector register file, a slice of the vector mask file, two functional units, an address generator and a private TLB. Notice the regularity of the overall engine: all lanes are identical and the instruction queues are located in the middle. There is no communication across lanes, except for gather/scatters (see below).

Regularity and design simplicity are not the only advantages that lanes give us. The schedulers that govern the allocation of this large number of functional units are very simple too. To them, the 32 functional units appear only as just two resources: the north and south issue ports. When an

instruction is launched onto one of the two ports, the sixteen associated functional units work fully synchronously on the instruction. Thus, the port is marked busy for $\lceil vl/16 \rceil$ cycles (typically, 8 cycles). To put it in another way: a simple dual-issue window is able to fully utilize 32 functional units.

The fact that the full register file is sliced into sixteen pieces is yet another advantage of the vector instruction paradigm. The bandwidth between the vector register file and the functional units is 64+32 operands per cycle. A unified register file would be totally out of the question. In contrast, the sliced register file enables that each lane only needs 4R+2W for the functional units[1]. Note that, as opposed to clustering techniques [11], writes into one register file lane do not need to be made visible to other lanes.

Finally, the vector mask register file is also sliced across the 16 lanes. The mask file is very small compared to the vector register file (256 bits per lane, including all rename copies per thread) and only requires three 1-bit read ports and two 1-bit write ports.

3.3. *Vbox*-Core Interface

Another positive aspect of the vector extension is that the *Vbox* can be integrated with the core with a relatively small interface (see Figure 2). In *Tarantula*, a 3-instruction bus carries renamed instructions from the *EV8* renaming unit (*Pbox*) to the *Vbox*. Routing space was scarce and, hence, a larger bus seemed impractical to use. When instructions complete in the *Vbox*, the vector completion unit (VCU) sends back to the *EV8* core their instruction identifiers (3x9 bits). Final instruction retirement is performed by the *EV8* core, which is responsible for reporting to the system software any exceptions occurred in the vector instruction flow. The other major interface is a double bus to carry two 64-bit values from the *EV8* register file to the *Vbox*. We note that all vector instructions except those of the VV group require a scalar operand as a source operand. These two buses are used to supply this data. Finally, the *EV8* core must also provide a kill signal, so that the vector unit can squash misspeculated instructions. Of course, tightly coupling to *EV8* also imposed some constraints: for example, to avoid excessive burden onto the operating system, the *Vbox* was also multithreaded. This decision forced using a much larger register file.

3.4. Vector Memory System

The *Tarantula* processor was targeted as a replacement for *EV8* in Compaq's mid-to-high end range servers and had to be a board-compatible replacement for *EV8*. Consequently, *Tarantula* had to integrate seamlessly into the Alpha virtual memory and the cache coherency architecture of *EV8*. Of course, these requirements completely ruled out exotic packaging technologies from vector supercomputers that provide more than 10,000 pins per cpu. Vector memory accesses had to be satisfied from the on-chip caches. We faced several important challenges to meet these requirements.

First, load/store bandwidth into the vector register file had to be an adequate match to the 32 flops of the vector engine. In *Tarantula*, we set our design goal to a 1:1 ratio between flops and bandwidth to the cache for unit stride cache accesses (i.e., a 64-bit datum for every flop). Attaching the vector engine to the L1 cache did not seem a very promising avenue. Typical L1 sizes are too small to hold the working sets of common engineering, scientific and bioinformatics applications. Moreover, re-designing *EV8*'s L1 cache to provide sixteen independent ports to support non-unit strides was out of the question. We were left with the obvious choice of having the *Vbox* communicate directly to the L2 cache (as already proposed in [15]).

Second, dealing with non-unit strides was a central problem in the design. Despite unit-strides being very common, they only account for around 80% of all vector memory accesses. Large non-unit strides account roughly for another 10%, while stride-2, the next most common case, accounts for a 4% [20, 14]. Cache lines and non-unit strides clearly don't blend together very well. Previous research on the topic either focused on providing good bandwidth only for unit stride vector accesses [15, 8] or simply went for a classical cache-less design [1, 22]. The Cray SV1 system sidestepped the problem by using single-word cache lines [4]. In *Tarantula*, we developed an address reordering scheme, described below, that enabled a 1:2 ratio between flops and cache accesses for non-unit stride instructions (i.e., sixteen independent 64-bit words per cycle from the cache).

Third, we had to integrate gather/scatter instructions smoothly into the pipeline. Gather/scatter instructions contain random addressing patterns to which the reordering scheme can not be applied. *Tarantula* employs a conflict-resolution unit, also described below, that sorts gather/scatter addresses into bank conflict-free buckets. Then, these sorted addresses can be sent as normal vector requests to the L2 cache.

Reusing *EV8*'s L2 design

Analyzing *EV8*'s L2 physical design we realized it already contained an enormous number of independent banks each with its own address decoder: *EV8*'s 4 Mbyte cache was physically laid out as 128 independent banks (8 ways times 16 banks per way). In addition, the design called for cycling eight banks in parallel on an L2 access, and then selecting the correct way.

[1] There are also 2R+2W ports to support stores and loads.

Thus, both from a structural and a power perspective, *EV8*'s L2 could easily accommodate reading 16 independent cache lines and then selecting one quadword[2] from each cache line provided each cache line was located on a different physical bank. The problem of delivering high bandwidth to non-unit stride requests reduced then to generating groups of 16 addresses that were cache-bank conflict-free. This is a variation of an old problem in vector memory design for supercomputers [20, 21], and next section describes the particular solution employed in *Tarantula*.

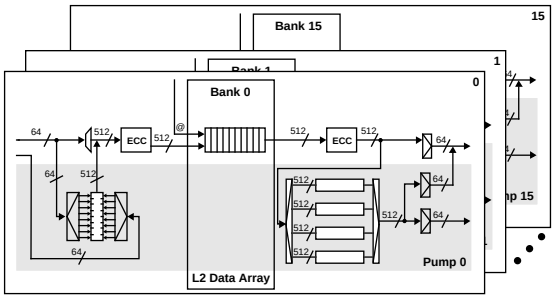

Figure 4. Pump structure used to accelerate stride-1 reads and writes.

Conflict-free Address Generation

As discussed in the previous section, the key to high performance is the ability to read in parallel sixteen cache lines from the sixteen L2 cache banks. Two conditions must hold to allow this parallel read: first, the addresses must be tag-bank and data-bank conflict free (since both arrays are equally banked, these two conditions are equivalent). Second, once the sixteen data items have been read from the L2, they must be written into the sixteen lanes of the vector register file without conflicts. That is, each lane can only accept one quadword from the memory system per cycle.

We proved that, for any 128 consecutive elements of any vector with stride $S = \sigma \times 2^s$ with σ odd and $s \leq 4$, there exists a requesting order that can group these 128 elements into 8 groups (each with 16 addresses) which are both L2-bank conflict free and register-lane conflict-free. This order can be implemented using a ROM distributed across the lanes that contains 2.1 Kbytes of information and a specialized 64-by-7 multiplier in each lane that uses the ROM contents to compute the starting address for each address generator.

The downside of our algorithm is that, as elements in the vectors are accessed out-of-order, we must wait for the full 128 elements to come back from the L2 before chaining a dependent operation. Consequently, vector instructions with vector length below 128 still pay the full eight cycles to generate all their addresses. The detailed address reordering algorithm is further described and analyzed in [16].

A group of 16 conflict-free addresses is called a "slice". Note that a slice need not be fully populated. Some of the addresses in it may have the valid bit clear (due to `vl` being less than 128 or due to a masked memory operation). All the vector memory pipeline is built around the concept of slices. Each slice is tagged with a slice identifier when it is created in the address generators and this tag is used throughout the memory pipeline to track it.

[2] A quadword (abbrv. "qw") is defined in the Alpha Architecture Reference Manual to be a 64-bit object.

Stride-1 Double Bandwidth Mode

Stride-1 instructions are treated specially to take advantage of their spatial locality. If properly aligned, the 128 quadwords requested by a stride-1 instruction are contained in exactly 16 cache lines (17 if the base address is not aligned to a cache line boundary). Rather than generating 8 slices, each with 16 addresses, we changed the address generation control to produce the *starting* address of each of the sixteen cache lines required instead. We then set the "pump" bit on the resulting slice[3].

The PUMP, shown in Figure 4, is a new structure located at the output of each bank in the L2 data array dedicated to accelerate both read and write stride-1 requests. Strides marked with the "pump" bit read out 16 cache lines from the data array just like any other slice does. But, as opposed to normal slices, the sixteen *full* cache lines are latched into one of the four registers (16x512 bits each) of the PUMP structure. From there, a sequencer in each bank reads *two* quadwords per cycle and sends them to the *Vbox*. The write path works similarly: the *Vbox* sends 32 quadwords worth of data every cycle, which get written into the accumulator register (to the left on Figure 4). When all 128 quadwords have been received (i.e., four cycles later), the PUMP will ECC the full register and write it in a single cycle into the data array. We note that using the PUMP, we can sustain a bandwidth of 64 qw/cycle (32 from a stride-1 read and 32 from a stride-1 write).

Gather/Scatters and Self-Conflicting Strides

Gather and scatter instructions do not use a stride value to generate their addresses. Rather, the programmer supplies a vector register that contains arbitrary addresses. Consequently, addresses do not form an arithmetic series and our reordering algorithm does not apply. However, in order

[3] For misaligned stride-1 cases, the address generators will be forced to generate two slices, both with the pump bit set.

to integrate them into the memory pipeline, we must pack these random addresses into slices.

Our solution is to take the 128 addresses of a gather/scatter and feed them to a piece of logic, the conflict resolution box (CR), whose goal is to sort the addresses into buckets corresponding to the sixteen cache banks. As a group of sixteen addresses comes out of the address generators, their bank identifiers (bits <9:6> of each address) are sent to the CR box. The CR box compares all sixteen bank identifiers with each other and selects the largest subset that are conflict-free. The resulting subset is packed into a slice and sent down the memory pipe for further processing. As new addresses (rather, bank identifiers) become available, the CR box will run again a selection tournament across whatever addresses where left from the previous round and as many new bank identifiers up to a limit of sixteen. By repeating this tournament procedure, eventually all addresses of the gather/scatter instruction will be packed into slices (worst case, when all addresses map to the same bank, an instruction may generate 128 different slices).

Self-conflicting strides are those strides that cause all addresses to map to only a handful of banks. More formally, strides $S = \sigma \times 2^s$ with σ odd and $s > 4$ are considered self-conflicting. Any instruction with such a stride is treated exactly like a gather/scatter and run through the CR box (instead of applying the reordering algorithm).

Virtual Memory

To keep the large number of functional units in the *Vbox* busy, it is important to avoid TLB misses. Piggy-backing on other work developed at Compaq to support large pages, the *Tarantula* architecture adopted a 512 Mbyte virtual memory page size [2, 3, 9].

The vector TLB is a parallel array of sixteen 32-entry fully associative TLBs, one TLB per lane. Each TLB is devoted to mapping the addresses generated by the address generator in its lane. Whenever a slice experiences a TLB miss, control is transferred to system software to initiate a TLB refill. System software may follow two strategies to refill the missing mappings: (1) it may simply look at the missing translations and refill those lanes where the miss has occurred, or (2) PALcode may peek at the `vs` value and refill the TLBs with all the mappings that might be needed by the offending instruction.

While we would rather use sixteen direct-mapped TLBs from a cost and power perspective, we note that a complication appears with some very large strides: a programmer could easily craft a stride that referenced 128 different pages which mapped onto the same TLB index (even with 512 Mbyte pages). Consequently, each TLB must be at least 8-way set-associative to guarantee that forward progress is possible on instructions with such strides. Given this restriction, a CAM-based implementation seemed more effective and we compensated the extra power consumption by choosing a small TLB per lane (32 entries only).

Servicing Vector Misses

Another interesting challenge appears when a vector memory instruction experiences a cache miss: The *Vbox* has sent a read slice to the L2 cache and several of its sixteen addresses miss in the lookup stage. How do we deal with this vector miss?

Our solution was to treat the slice as an atomic entity. If one of the addresses in a slice experiences one or more misses, the slice waits in the L2 cache (in the Miss Address File, MAF) until the corresponding system requests are made and all the missing cache lines are received from their home nodes. The slice is "put to sleep" in the MAF and a "waiting" bit is set for each of its sixteen addresses that missed. As each individual cache line arrives to the L2 from the system, it searches the MAF for matching addresses. For each matching address, its "waiting" bit is cleared. When all "waiting" bits are clear, the slice wakes up and goes to the Retry Queue (a structure within the L2 cache itself). From there, the slice will retry, walk down the L2 pipe again and lookup the tag array a second time. The hope is this second time the slice will succeed in reading/writing its data. To avoid the potential livelock we introduced a replay threshold value. If a slice replays more times than the threshold, the MAF enters "panic mode" and starts NACKing all L1, *Vbox* and interprocessor requests that may prevent forward progress for that slice. Only when the slice is finally serviced, the MAF resumes normal operation.

Scalar-Vector Coherency

Coherency problems appear in *Tarantula* because the *EV8* core is reading and writing from the L1 cache and the *Vbox* is reading/writing into the L2 cache behind its back. We must ensure that the data read/written both by *EV8* and the *Vbox* are the same as if both where writing sequentially into the same memory space.

The protocol used to achieve the desired scalar-vector coherency is based on the ideas presented in [15]. Each tag in the L2 cache is extended with a "presence" bit (P-bit). The presence bit indicates whether the cache line was loaded into the L2 due to a request from the *EV8* core or not. In essence, the P-bit is like a soft-ownership bit and is set whenever the *EV8* core touches a cache line. The P-bit is used whenever the *Vbox* is checking the L2 tags to know whether there is the danger that the cache lines being read/written might also be in the L1 cache. If the P-bit is set, then invalidate commands must be sent to the L1 to synchronize the state of both caches. The invalidate commands

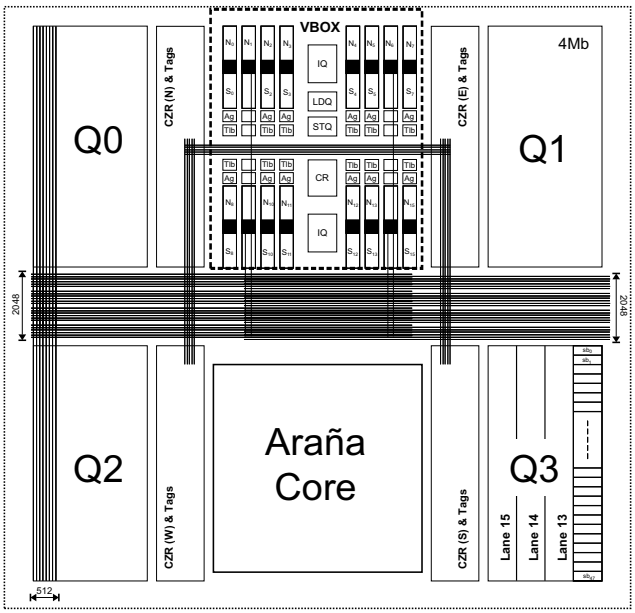

Figure 5. *Tarantula* **floorplan.**

either remove the line from the L1 if it is clean, or force a write-through of the line to L2 if it's dirty. Also, when a line is evicted from the L2, if it's P-bit is set, it will also cause an Invalidate command to be sent to the *EV8* core.

The P-bit solves most of the coherency problems that arise due to the out-of-order execution of vector and scalar read/write instructions. However, one case is not covered and requires programmer intervention: a scalar write followed by a vector read[4]. In *EV8*, a scalar store sits in the store queue, potentially with its associated data, until it retires. At that time, it moves from the store queue into the write buffer *without informing* either the L1 cache or the L2 cache. A younger vector read going out to the L2 has no visibility into either the write buffer or the store queue. It may well be that no P-bit is set in the L2. Therefore, the vector load has no way of knowing it's reading stale data. The programmer/compiler must insert a special memory barrier called *DrainM* to solve this problem. When *DrainM* is about to retire, it sends a purge command to the write buffer. This purge forces all previous stores out of the store queue and into the cache hierarchy, and also updates their associated L2 P-bits. When the purge is complete, the *DrainM* retires and causes a *replay trap* on the following instruction. Thus, all instructions younger than the *DrainM* are killed and re-fetched, ensuring correct behavior.

[4] A scalar write followed by a vector write is correctly handled by having all scalar writes write-through to the L2 before actually letting the vector write proceed. This behavior is only forced for those threads that have vector instructions.

	CMP-*EV8*		*Tarantula*	
Circuitry	Area (%)	Power (W)	Area (%)	Power (W)
Core	42	54,3	15	22,2
IO Drivers		26,5		26,5
IO logic	14	6,6	8	4,3
L2 cache	33	5,1	43	7,6
R/Z Box	5	6,3	7	10,1
Vbox	–	–	15	30,9
Other	6	7,9	12	18,2
Total (+20%)		128,0		143,7
Die Area	250 mm^2		286 mm^2	
Peak Gflops	20		80	
Gflops/Watt	0,16		0,55	

Table 1. Power and area estimates for a CMP-*EV8* processor and for *Tarantula*. The "Total" line includes a 20% extra power attributed to leakage.

4. Floorplan

The current *Tarantula* floorplan is shown in Figure 5. As it can be seen, the floorplan is highly symmetric. The cache is located at the outer corners of the die. The cache holds a total of 16 MBytes and is split into four quadrants, indexed by bits <7:6> of the address. Each quadrant holds 4 cache lanes, selected by bits <9:8>, for a total of 16 lanes. Each cache lane holds 48 stacked banks, over which run 512 wires to read/write the cache line data. The wiring uses a coarse-level metal because the distance to the central bus area is rather large.

The central bus area implements the crossbar between the cache lanes and the *Vbox* lanes. As already discussed, a quadword from any of the cache lanes may have to go to any of the *Vbox* lanes. We take advantage of the fact that bits flowing from the quadrants have to "take two turns" on their path to the *Vbox*. This means that the point were the north-south wires have to be connected to the east-west wires is an ideal place to put the different crossbar connections. The central bus itself carries 4096 bits, but is folded onto itself by using alternate East-West metal layers, so that it uses an area equivalent to a 2048-bit bus.

The floorplan also illustrates how the different *Vbox* lanes are organized in four groups of four lanes. Each lane shows a shaded area in the middle, the register file, and a north and south functional unit. The instruction queue is replicated and located in between the lanes to minimize wiring delays. In each lane, one can see an address generator. The alignment of the address generators with the lanes is fundamental to ease the wiring for the scatter/gather addresses. Again, the load and store queues are located at the center of the address generators to equalize delays. The CR box is also located at the center of all address generators.

Benchmark	Description	Inputs	Comments	Pref?	DrainM?	Vect. %
MicroKernels						
STREAMS [12]	Copy, Scale, Add, Triadd Kernels	Reference	Padding=65856 bytes			99.5
`RndCopy`	B(i) = A(index(i))	A,B=4096000 elements	Prefetched into L2	yes		99.9
`RndMemScale`	B(index(i)) = B(index(i)) + 1	B=512000 elements	All data from memory	yes		99.9
SpecFP2000						
`swim`	Shallow Water Model	Reference	Tiled following [17]			
`art`	Image Recognition/Neural Networks	Reference			yes	99.3
`sixtrack`	High Energy Nuclear Physics	Reference				93.7
Algebra						
`dgemm`	Dense, Tiled, Matrix Multiply	640x640		yes		99.0
`dtrmm`	Triangular matrix multiply	519x603	Dense, Tiled	yes		98.9
`Sparse MxV`	Sparse matrix-vector product	24696x24696	887937 non-zeroes	yes		99.3
`fft`	Radix-4 Fast Fourier Transform	5120 FFTs	1024 elements per FFT	yes		98.7
`lu`	Lower-Upper Matrix decomposition.	519x603	Tiled Version		yes	98.6
`Linpack100`	Dense Linear Equation Solver	100x100	No code reorganization		yes	85.5
`LinpackTPP`	Dense Linear Equation Solver	1000x1000	Tiled		yes	96.5
Bioinformatics						
`moldyn`	Molecular Dynamics	500 molecule system				99.5
Integer						
`ccradix`	Tiled Integer Sort	2000000 elements	From [10]			98.0

Table 2. Benchmark descriptions. Columns "Prefetch" and "DrainM" indicate whether the benchamrk uses those features.

The CR box needs the bank information from each address generator to sort the addresses into conflict-free slices.

5. Power Estimates

We estimated the power of the *Tarantula* processor by scaling *EV7*'s power and area estimates down to 65 nm technology. Table 1 presents a breakdown of power and area estimates assuming a voltage slightly under 1V and a clock frequency of 2.5 Ghz. The table presents estimates for two different architectures: a CMP-style processor based on two *EV8* cores[5] and the *Tarantula* processor described so far. Both processors use the same L2 cache and memory subsystem. The power consumption of the *Vbox* is extrapolated using the power-density of the floating point units of *EV7*, and thus should be considered a lower bound, since the TLBs and address generation logic are not properly accounted for.

Our estimates show that *Tarantula* is 3.4X better in terms of Gflops/Watt than a CMP solution based on replicating two *EV8* cores. We note that adding floating point multiply-accumulate units (FMAC) to *Tarantula*, this rate could be doubled with very little extra complexity and power. In contrast, adding FMAC instructions that require an extra third operand to *EV8* would require an expensive rework of *EV8*'s instruction queue and renaming units.

[5]Notice that in the *Tarantula* floorplan, a *Vbox* or an *EV8* core could be used interchangeably, and yet keep the same memory system.

6. Performance Evaluation

Evaluation Methodology

The *Tarantula* processor is targeted at scientific, engineering and bioinformatics applications. Consequently, we selected a number of applications and program kernels from these areas to use them as our workload. Unfortunately, no vectorizing compiler that could generate the new *Tarantula* instructions was available. Hence, we used profiling to determine the hot routines of each benchmark and, then, these were coded in vector assembly by hand. All programs were compiled using either Compaq's C or Fortran, versions 5.9 and 5.2 respectively. The benchmarks chosen are described in Table 2.

For our simulations, we used the ASIM infrastructure [7] developed by Compaq's VSSAD group. Included in the ASIM framework is a cycle-accurate *EV8* simulator, that is validated against the *EV8* RTL description. We modified this base code to derive the *Tarantula* simulator, by hooking up the *Vbox* to it and modifying the *Cbox* model to accept vector requests. No changes were made to the internals of the *Zbox* or *Rbox* (except, of course, adding more ports). All our simulations run in single-thread, single-processor mode.

The *Tarantula* architecture's main parameters are shown in Table 3. The other architectures under study are also included: *EV8* is the baseline against which we compare the *Tarantula* processor. *EV8+* is an *EV8* processor equipped with *Tarantula*'s memory system. Finally, we include also *Tarantula4*, an aggressively clocked *Tarantula* processor.

Symbol	*EV8*	*EV8+*	T	T4
Core Speed (Ghz)	2.13	2.13	2.13	4.8
Ops per Cycle:				
Core Issue	8	8	8	8
Vbox Issue	–	–	3	3
Peak Int/FP	8/4	8/4	32	32
Peak Ld+St	2+2	2+2	32+32	32+32
EV8-Vbox latency	–	–	10	10
L1 assoc	2	2	2	2
L1 line (bytes)	64	64	64	64
L2 size (x2^{20})	4	16	16	16
L2 assoc	8	8	8	8
L2 line (bytes)	64	64	64	64
L2 BW (GB/s)	273	273	1091	2457
L2 Load-to-use lat.				
Scalar Req	12	12	28	28
Vector Stride-1	–	–	34	34
Vector Odd stride	–	–	38	38
RAMBUS:				
Ports	2	8	8	8
Speed (Mhz)	1066	1066	1066	1200
BW (GB/s)	16.6	66.6	66.6	75.0

Table 3. Characteristics of the four architectures under study. Note that the L2 BW refers to maximum sustainable bandwidth: for *EV8*, both a line read and a line write can proceed in parallel. For *Tarantula*, in stride-1 mode, 16 lines can be read every 4 cycles and, in parallel, 16 lines can be written every 4 cycles. Latencies are given in cycles.

The frequency for each processor is derived from the corresponding RAMBUS frequency by using either a 1:2 ratio or a 1:4 ratio.

Memory System Microbenchmarks

Table 4 presents the performance of the three microbenchmarks targeted at measuring memory system behavior running on *Tarantula*.

To understand the copy loop bandwidth, note that out of a peak raw bandwidth of 66.625 GB/s (8 channels at 1066 Mhz), 1/3 is consumed by directory updates. The loop kernel is composed of a memory read, a *wh64* instruction that generates a directory transition from Invalid to Dirty (i.e., a read from RAMBUS), and the store that eventually causes a memory write. Thus, 2/3 are "useful" bandwidth. Out of this peak, read-to-write transitions on the RAMBUS bus limit our achieved bandwidth to 90%, or 40.3 GB/s. As a reference, the top STREAMS copy bandwidth for a single cpu, as of Oct 31^{st}, 2001 [12], corresponds to the NEC SX/5 [13] with a value of 42.5 GB/s. The best single-cpu non-vector result is a Pentium4 1.5 Ghz system manufactured by Fujitsu with a copy bandwidth close to 2.4 GB/s.

The `RndCopy` microkernel tests our gather/scatter bandwidth from L2 cache (no TLB misses and no L2 cache misses). Here, the limitation are the bank conflicts encountered by the CR box when dealing with a random address stream. The bandwidth delivered corresponds to an address generation bandwidth of around 4.3 addresses/cycle.

STREAMS	Streams BW	Raw BW
`Copy`	42983	64475
`Scale`	41689	62492
`Add`	43097	57463
`Triadd`	47970	63960
`RndCopy`	73456	NA
`RndMemScale`	7512	50106

Table 4. Sustained bandwidth in MBytes/s on *Tarantula* for several microkernels. The "Raw" column includes all memory transactions performed at the RAMBUS controller, including those necessary to update the directory information. The "Streams" column reports useful read/write bandwidth using the STREAMS method without accounting for directory traffic.

The `RndMemScale` microkernel tests the random bandwidth achievable from RAMBUS. Here, the limitation is not only the address generation, but the fact that randomly touching RAMBUS "pages" causes an extra amount of "opening" and "closing" of memory pages: compared to STREAMS `Copy`, `RndMemScale` performs 2.5X more row activates and 2X more row precharges per memory request.

Benchmark Performance

We turn now to the performance of the remaining benchmarks. The results of our simulations are shown in Figure 6. For each benchmark, we present the number of sustained "operations per cycle" (OPC) broken into three categories: flops per cycle (FPC), memory operations per cycle (MPC) and other (including integer arithmetic and scalar instructions).

The results show that for most benchmarks, *Tarantula* sustains over 10 operations per cycle, and it's not uncommon to exceed 20 operations per cycle. The benchmarks with less parallelism are, not surprisingly, those where gather/scatter operations dominate: the sparse matrix-vector multiply and the two versions of radix sort. Also note how short vector length also impacts performance: `linpack100` is significantly slower than the TPP counterpart.

As opposed to a conventional cache-less vector machine, *Tarantula*, like any conventional superscalar or CMP design, must pay careful attention to exploiting data reuse at

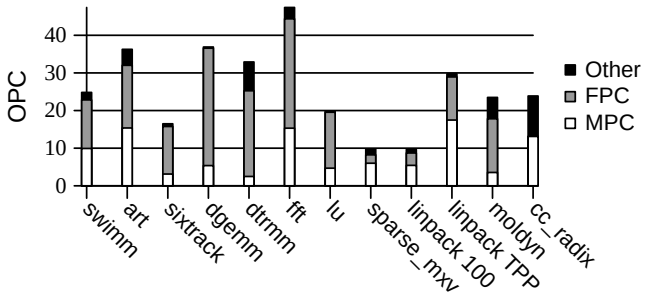

Figure 6. Operations per cycle sustained in *Tarantula*. Each bar is broken down into Flops per cycle (FPC), Memory Operations per Cycle (MPC) and Other (which includes integer and scalar operations).

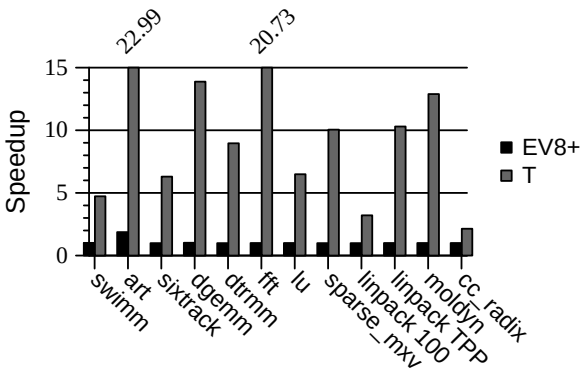

Figure 7. Speedup of *EV8+* and *Tarantula* over *EV8*.

the memory hierarchy level. Consequently, all our benchmarks are either cache-friendly or have been turned into cache-friendly codes by using standard tiling techniques. To stress the importance of tiling for the memory hierarchy in *Tarantula*, we also run a "naive" non-tiled versions of `swim` (not shown in the graph). The slowdown was significant: the non-tiled version was almost 2X slower. Also, benchmarks `LU` and `LinpackTPP` perform very similar tasks, yet `LinpackTPP` shows 50% more operations per cycle. The reason is that we performed register tiling for `LU` but not for `LinpackTPP`, thus reducing `LU`'s memory demands.

Figure 7 presents the speedup of the *Tarantula* processor over *EV8*. As a reference, the speedup of *EV8+* is also included. The speedup results show that, typically, *Tarantula* achieves a speedup of at least 5X over *EV8*. Given that these are floating point applications, the nominal speedup based on raw floating point bandwidth is 8X. Thus, *Tarantula* delivers a very good fraction of its promised peak performance. Furthermore, as the *EV8+* results show, this performance advantage can not be attributed to the bigger cache and better memory system alone: it's the use of vector instructions that enables squeezing maximum performance from this improved memory system.

Interestingly, six applications exceed this 8X speedup factor. There are a number of reasons for this: First, *Tarantula* has a better flop:mem ratio than *EV8* (32:64 versus 4:4) for those programs that use mostly stride 1. Second, *Tarantula* has many more registers available, which turns into more data reuse and less memory operations when tiling the iteration space. Third, all these programs have been hand vectorized and hand tuned. To be fair to *EV8*, the same care should be applied optimizing the scalar inner loops. Moreover, the *EV8* versions are compiled using an EV6 scheduler because no EV8 scheduler was yet available. This is specially significant in `dgemm` where *EV8* only reaches 2.5 flops/cycle. Scheduling specifically targeted to EV8 would most likely increase the flop rate to the peak and, consequently, reduce *Tarantula*'s speedup. Fourth, vectorization provides advantages because it reduces the impact of branches and pointer-maintenance instructions. For programs like `fft`, where lots of ILP are available, if *EV8* used all issue slots to execute 4 flops and 4 memory operations, none would be left to execute loop-related control instructions. Vector masks are also a significant source of speedup in `moldyn`: by executing under mask, *Tarantula* avoids hard-to-predict branches and obtains some extra speedup. Finally, the *Tarantula* versions use aggressive prefetching techniques: note that a single vector load with a stride of 64 bytes can preload a total of 128 cache lines, or 8 KB worth of data. In comparison, *EV8* needs a separate prefetch instruction for every cache line it wants to preload. A similar argument explains the speedup of `sparsemxv`: Driving an eight-channel RAMBUS memory controller is quite difficult for a superscalar machine that can generate at most 64 misses before stalling. In contrast, a handful of gather instructions can easily generate 1024 distinct cache line misses, and, consequently, drive the RAMBUS array to full usage.

Scalability

We also explored how well performance scales as frequency increases. Figure 8 presents the results. The main consequence of increasing frequency is that the processor-to-Rambus ratio grows very fast. Hence, memory operations take substantially longer to complete and performance will only scale if (a) the program is working mostly from cache or (b) there's enough parallelism and prefetching capabilities to cover up for the increased latency.

Our simulations show that, as expected, those programs

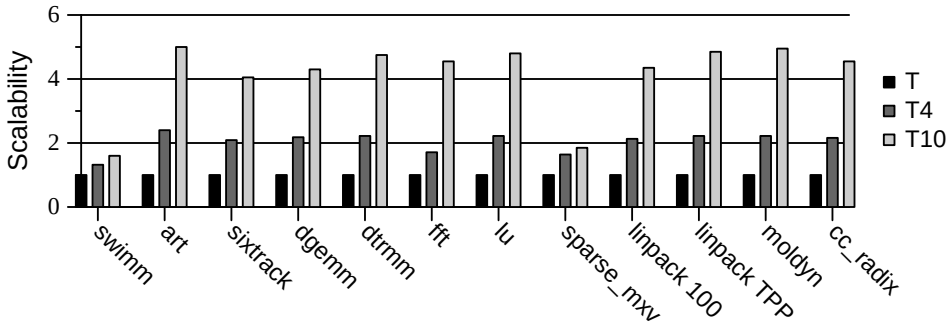

Figure 8. Performance Scaling when Frequency is increased to 4.8Ghz (T4) and to 10.6Ghz (T10). The latter frequency results from a 1:8 ratio to a RAMBUS chip running at 1333Mhz.

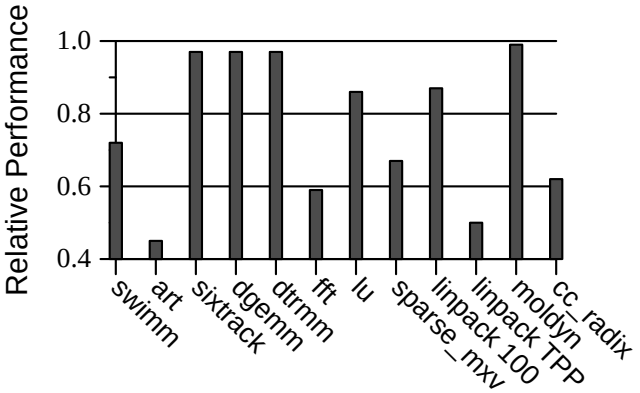

Figure 9. Slowdown due to disabling the stride-1 double-bandwidth mode.

that mostly access the L2 cache scale very well. In contrast, `sparsemxv` barely reaches speedups of 1.6 and 1.8 when scaling the frequency by 2.2X and 5X respectively. The results are encouraging, as they show that a *Tarantula* core could have a very long life by shrinking and, thus, the design cost could be amortized over multiple technology generations.

Stride-1 Double Bandwidth mode

The final experiment we present is a set of simulations that measure the effect of using the PUMP structure to accelerate stride-1 requests. The simulation results are shown in Figure 9.

Naturally, the programs that did not have their iteration space tiled suffer the most when stride-1 bandwidth is dropped from thirty-two 64-bit words per cycle down to sixteen. A second effect that must be taken into account is that the pressure on the MAF grows by a factor of 8X, since, without the pump, each stride-1 request now consumes eight MAF slots for its eight slices as opposed to a single slot with the PUMP scheme. Note the performance of `sparsemxv` drops significantly without the pump. This may appear surprising, but the algorithm we tested makes good use of stride-1 performance. Also, one would expect `ccradix` to be dominated by gather/scatters, but stride-1 performance is important here as well.

7. Conclusions

There still exist a wide number of applications where a vector processor is the most efficient way of achieving high performance. In this paper we have presented *Tarantula*, a vector extension to the Alpha architecture that can achieve up to 104 operations per cycle (96 vector operations; 32 integer or floating point arithmetic operations, 32 loads and 32 stores; and 8 scalar instructions). *Tarantula* is the proof that the vector paradigm can be fully exploited in a real microprocessor environment.

The huge bandwidth required to feed the 32 functional units of *Tarantula* can only be provided from a large L2 cache. Given the restrictions of a microprocessor design, the main challenges that had to be solved to achieve the target performance are: (1) integration into the existing Alpha virtual-memory cache-coherent system, (2) good performance for non-unit strides, (3) support for gather and scatter instructions, and (4) reuse as much as possible the *EV8* core to reduce design and development time.

Our performance studies are very promising. The sustained number of operations per cycle ranges from 10 to almost 50. This performance translates into speedups from 2 to 20 over *EV8* (an aggressive high performance superscalar microprocessor that could execute up to 8 instructions per cycle). Also, when coupling our performance results with the initial power estimates, we believe the performance-per-watt would hardly be achievable by any other kind of architecture.

Generating the right code for *Tarantula* is fundamental to fully benefit from the L2 cache bandwidth. Our experience in hand coding the benchmarks studied indicates that both tiling and aggressive prefetching are fundamental to achieve the performance levels achieved.

Credits and Acknowledgments

G. Lowney and J. Emer have pushed this project forward from its creation up to the moment the Alpha team was transferred to Intel. Without their encouragement and technical insights, the project would have not been possible.

The core *Tarantula* team is: R. Espasa: Lead Architect. F. Ardanaz: *CR box*, benchmarking. S. Felix: floorplan, wireplan and power estimates. J. Gago: *Cbox*, benchmarking. R. Gramunt: *Vbox*, masks, benchmarking. I. Hernandez: *Zbox*, benchmarking. T. Juan: floorplan, quad-cache proposal. M. Mattina: *Cbox*. A. Seznec: memory accessing scheme, memory coherency.

We would like to give very special thanks to the ASIM team, for a terrific modeling environment that made this project so much easier: J. Emer, P. Ahuja, N. Binkert, E. Borch, R. Espasa, T. Juan, A. Klauser, C.K. Luk, S. Manne, S. Mukherjee, H. Patil, and Steven Wallace. Also, thanks to M. Jiménez for tiling most of our benchmarks and J. Corbal for the Rambus model.

Finally we would also like to thank the following individuals for their valuable input: T. Fossum, P. Bannon, S. Root, J. Leonard, B.J. Jung, R. Foster, G. Chrysos, S. Samudrala, R. Weiss, B. Noyce, J. Piper, B. Hanek, and the GEM compiler team.

References

[1] K. Asanovic, J. Beck, B. Irissou, B. Kingsbury, N. Morgan, and J. Wawrzynek. The T0 Vector Microprocessor. In *Hot Chips VII*, pages 187–196, August 1995.

[2] P. Bannon. Alpha 21364: A Scalable Single-chip SMP. In *Microprocessor Forum*, October 1998.

[3] P. Bannon. EV7. In *Microprocessor Forum*, October 2001.

[4] Cray, Inc. Cray SV1. In *http://www.cray.com/products/systems/sv1/*, 2001.

[5] K. Diefendorff. Compaq Chooses SMT for Alpha. *Microprocessor Report*, 13(16):5–11, December 1999.

[6] J. Emer. Simultaneous multithreading: Multiplying alpha's performance. In *Microprocessor Forum*, October 1999.

[7] J. Emer, P. Ahuja, N. Binkert, E. Borch, R. Espasa, T. Juan, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, and S. Wallace. Asim: A Performance Model Framework. *IEEE Computer*, 35(2):68–76, February 2002.

[8] R. Espasa and M. Valero. Exploiting Instruction- and Data-Level Parallelism. *IEEE Micro*, pages 20–27, September/October 1997.

[9] L. Gwennap. Alpha 21364 to Ease Memory Bottlenck. *Microprocessor Report*, 12(16):12–15, October 1998.

[10] D. Jimenez-Gonzalez, J. J. Navarro, and J. Larriba-Pey. The effect of local sort on parallel sorting algorithms. In *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, January 2002.

[11] J. Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. In *Microprocessor Forum*, October 1996.

[12] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE TCCA Newsletter, see also http://www.cs.virginia.edu/stream*, December 1995.

[13] NEC, Inc. NEC SX5, Vector Technology. In *http://www.nec.com.au/hpcsd/vector.htm*, 2001.

[14] F. Quintana. *Vector Accelerators for Superscalar Processors*. PhD thesis, Universidad de las Palmas de Gran Canaria, 2001.

[15] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a Vector Unit to a Superscalar Processor. In *International Conference on Supercomputing (ICS)*. ACM Computer Society Press, 1999.

[16] A. Seznec and R. Espasa. Conflict free accesses to strided vectors on a banked cache. *In Preparation*.

[17] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–228, 1999.

[18] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA*, pages 392–403, 1995.

[19] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 191–202. ACM Press, May 1996.

[20] M. Valero, T. Lang, J. Llaberia, M. Peiron, and J. Navarro. Increasing the number of strides for conflict-free vector access. *International Symposium on Computer Architecture*, pages 372–381, 1992.

[21] M. Valero, T. Lang, M. Peiron, and E. Ayguadé. Conflict-Free Access for Streams in Multimodule Memories. *IEEE Transactions on Computers*, 44(5):634–646, May 1995.

[22] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. Spert-II: A vector microprocessor system. *Computer*, 29(3):79–86, 1996.