# EXPLOITING INSTRUCTION- AND DATA-LEVEL PARALLELISM

**Roger Espasa**

**Mateo Valero**

Polytechnic University of Catalunya-Barcelona

*Simultaneous multithreaded vector architectures combine the best of data-level and instruction-level parallelism and perform better than either approach could separately. Our design achieves performance equivalent to executing 15 to 26 scalar instructions/cycle for numerical applications.*

Historically, computer architects have taken two different approaches to high-performance computing: instruction-level parallelism and data-level parallelism. The ILP paradigm seeks to execute several instructions each cycle. It does this by exploring a sequential instruction stream and extracting independent instructions to send to several execution units in parallel. The DLP paradigm, on the other hand, uses vectorization techniques. A vector instruction specifies a series of operations to be performed on a stream of data. Each operation performed on each individual element is independent of all others, and, therefore, a vector instruction is highly parallel and can be easily pipelined. In this article, we propose a third approach to high-performance computing that combines the best of ILP and DLP techniques to provide an order of magnitude increase in performance at low complexity.

## Processor generations

Figure 1 illustrates three microarchitecture generations in the DLP world. The first vector generation, shown in Figure 1a, introduced in-order, pipelined execution of vector instructions. This generation's prototypical machine is Cray Research's Cray-1.

The second DLP generation, in Figure 1b, exploited the parallel semantics of vector instructions to implement multipipe functional units—unit replication that allows processing of more than one pair of operands per cycle. Cray Research's C90 or Nippon Electric Corp.'s SX-3 exemplified the multipipe processor. However, this generation still used the in-order execution model. Useful ILP techniques such as out-of-order execution or register renaming, which fight memory latency and improve processor throughput in the microprocessor world, have never been used in commercial vector computers.

The third DLP generation, depicted in Figure 1c, is the one we are proposing in this article. This processor merges ILP and DLP in a single-processor architecture that combines three key technologies:

- vector instructions,
- out-of-order execution with register renaming, and
- simultaneous multithreaded execution.

This combination yields a simultaneous multithreaded vector, or SMV, architecture.

## Vectorizable code

For many years, most scientific computing applications have largely followed the DLP model. Much of the vectorizable code optimized for yesterday's vector supercomputers runs on today's superscalar microprocessors. These codes still retain their DLP characteristics. Moreover, in recent years applications containing highly regular DLP code have multiplied. In particular, many DSP and multimedia applications—graphics, compression, encryption—are superbly suited for vector implementation.[1]

## SMV architectures

Vector instruction sets and vector architectures are an excellent match for the characteristics of data-parallel codes. Other architectures such as chip multiprocessors or multiscalar processors[2] are also good candidates to extract high performance from data-parallel code. However, vector instruction sets use fewer processor control resources and they directly (and easily) convey parallelism to the hardware.

The vector instruction set is thus our basic building block for high performance. Data-level parallelism, however, is not enough. For large memory latencies, the DLP model suffers severe performance degradation. This is where the ILP model's out-of-order execution

 0272-1732/97/$10.00 © 1997 IEEE

and register renaming come into play. Out-of-order execution reorders instructions and reduces the interference between computation and memory instructions. Reordering makes better use of the memory ports and masks memory latency. Register renaming, while not essential for a newly defined architecture, helps manage the processor state and provide precise exceptions, always a difficult thing in vector processors. Renaming can also improve the use of the register pool.

But even out-of-order execution is not enough when we consider large numbers of resources such as memory ports and arithmetic units. At some point, a program's intrinsic instruction-level parallelism limits the number of functional units that can be used simultaneously. Here multithreading comes to the rescue. By multiplexing several threads onto the same processor, we can fully use all resources. Simultaneous multithreading, moreover, indicates that there are no explicit context switches between threads. Rather, instructions from different threads can coexist at the same time in the reorder buffer. Simultaneous multithreading logically follows in a DLP machine that already has out-of-order execution and register renaming. A few bits of thread information in the instruction queues and per-thread rename tables are enough to achieve the desired SMV architecture.

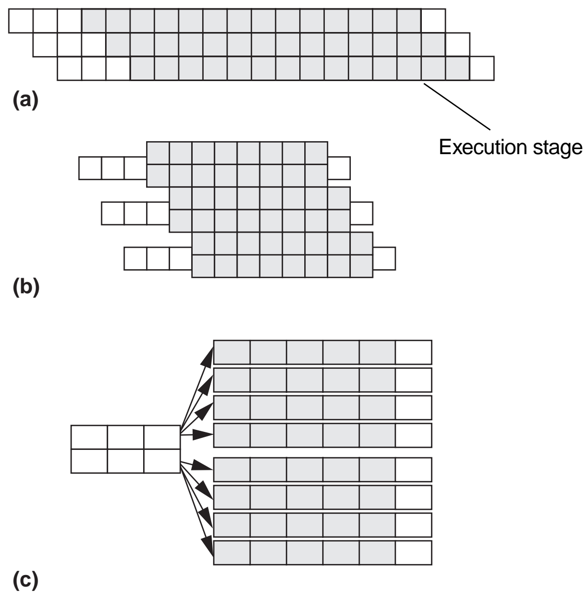



Figure 1. Three DLP microarchitecture generations: vector processor (a); multipipe vector processor (b); and simultaneous multithreaded vector (SMV) processor (c).

## Design challenges

Although the SMV approach is, as we will show, a solid candidate for handling a billion transistors on one chip, building tomorrow's billion-transistor processors means first facing three major concerns:

- *Wire delays.* The difference between transistor commutation speed and signal propagation speed on a wire determines that today's cycle times are mostly dominated by signal propagation delays. Wire delays will determine the maximum chip area that a single clock can control. It may also force the use of two-level clocking: fast local clocks in close proximity regions and slower clocks to connect distant regions inside a chip.
- *Processor-to-memory performance gap.* Memory access time is today's most important problem regarding high-performance microprocessors.[3] Currently, microprocessor performance improves at the rate of 60% each year, while dynamic RAM speed improves at less than 10% each year. Research indicates that, to reduce this gap, the best approach is to merge memory and logic on the same chip. The intelligent RAM (IRAM)[4] and PIM[5] (Processor in Memory) proposals both target this approach.
- *Parallelism inside a program.* Parallelism is defined as the number of independent tasks (instructions) that can be executed in parallel while preserving the original program semantics. If many independent operations are available, a processor might attempt to execute them simultaneously, thus improving the instructions per cycle (IPC) rate. However, there is a limit to how many independent tasks the processor can successfully detect and execute simultaneously. Without a compiler technology breakthrough, current software standards dictate that only a few instructions might be available for parallel execution.

## Scalability problems

How do the two high-performance models face future challenges? Compare one scalar instruction from the ILP paradigm to one instruction from the DLP paradigm. A scalar instruction requires many stages to be processed yet specifies only a very simple operation. Meanwhile, the vector instruction has a few setup and shutdown stages but has many more useful stages involving actual computations. In a scalar instruction, most of its execution phases are devoted to bookkeeping activities required to maintain the processor's consistency. Adding more execution resources to a superscalar processor requires many extra control resources. On the contrary, in a DLP processor we can speed up vector computations simply by replicating functional units (see Figure 1b).

To understand how these semantic differences will be affected by the three design bottlenecks, consider Equation 1. It characterizes the speed at which a computer can execute a given program.

$$T = N \times CPI \times T_c \tag{1}$$

In this equation, *CPI* (cycles per instruction) is the average number of cycles required to execute each instruction. The time to complete a program ($T$) equals the number of instructions executed ($N$) times ($CPI$) times the machine cycle time $T_c$. Note that CPI is just the inverse of the IPC rate described earlier.

**Number of instructions (*N*).** Data-level parallelism lets us reduce the number of instructions by making each vector instruction longer. However, each vector instruction performs many operations and, therefore, can take many cycles to complete, which increases CPI. Nonetheless, the higher

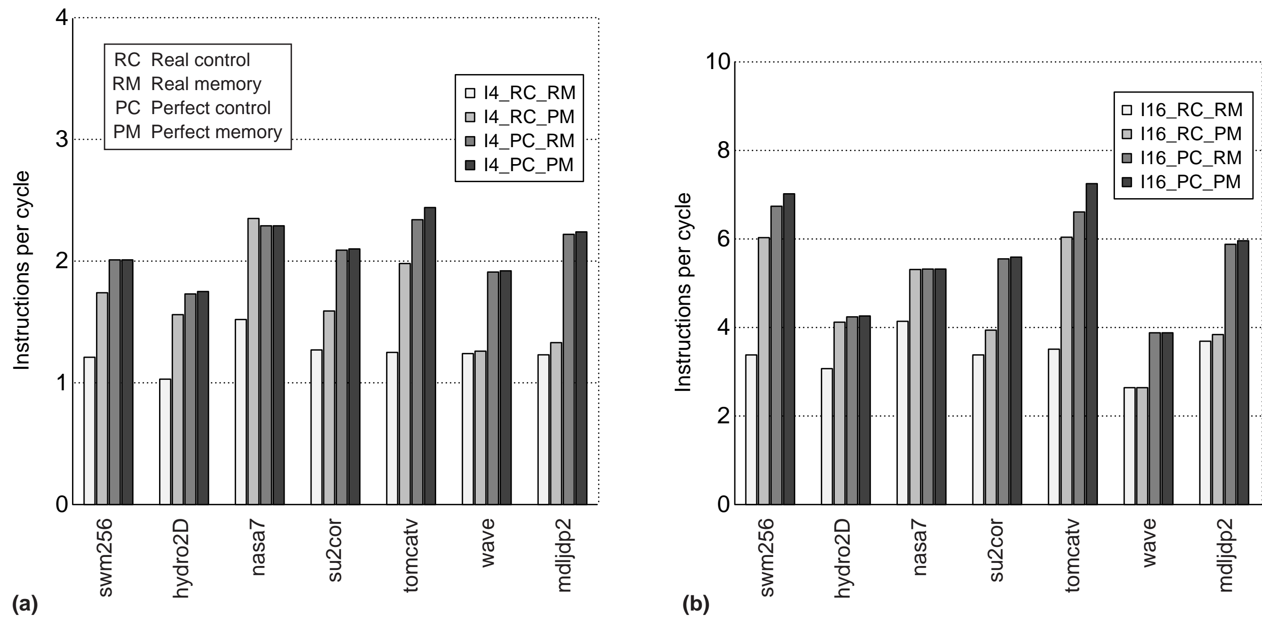



Figure 2. Performance, in terms of instructions per cycle, for a superscalar processor closely modeled after a Mips R10000. (a) It issues 4 instructions per cycle, with 2 floating-point units, 2 integer units, and 1 memory port. Performance, in terms of IPC, for a scaled version of the superscalar processor. (b) Issue width is 16; it has 8 floating-point units, 8 integer units, and 4 memory ports. The reorder buffer has been increased from 32 to 128, as has the Branch Target Buffer (from 512 entries to 2,048).

semantic content of vector instructions has indirect benefits. To perform a given task, a vector program must specify many fewer address computations, loop counter increments, and branch computations, because these are typically implicit in vector instructions. The DLP paradigm achieves an overall $N \times CPI$ product that is much better than that of an ILP machine.

**Cycle time ($T_c$).** Wire delays inside a chip take up increasing amounts of today's microprocessor cycle times, and ILP processors present clear scalability problems in this regard. Analysis of current superscalar processors'[6] circuit complexity shows that the wake-up and select logic needed in wide-issue machines scales unfavorably when designers decrease feature size. These results pose serious obstacles to the feasibility of very wide centralized superscalar machines (16- or 32-wide issue). Researchers believe that high performance will come only by replication of very fast building blocks that work more or less asynchronously. The DLP model supports independent, replicated building blocks very well. The operations inside a vector instruction do not interchange information and can be physically partitioned into independent sub-blocks that need not communicate with each other.

**Cycles per instruction (*CPI*).** Many factors affect the CPI part of the equation, but we focus on memory access time and instruction-level parallelism. (We use the terms *CPI* and *IPC* depending on the context, since both measures are equivalent.)

*Memory access time.* Memory-accessing instructions account for 30% to 40% of all instructions. Thus, the larger the difference between CPU and memory speed, the longer it takes each memory instruction to complete. This means the CPI component grows very fast. To compensate for this large performance gap, current superscalar micros use increasingly large caches to sustain performance. Nonetheless, despite out-of-order execution, nonblocking caches, and prefetching, superscalar micros do not efficiently use their memory hierarchies. Since load/store instructions are mixed with computation and setup code, dependencies and resource constraints prevent memory operations from being launched every cycle. The result is that each cache miss turns out to have very long latency. Scalar instruction sets have severe shortcomings where high memory latencies are concerned. For example, to hide a memory latency of 100 processor cycles, a superscalar machine should have more than a hundred in-flight operations. It is a daunting task to extract so many independent operations from a sequentially specified program.

*Vector advantages.* Vectors, however, have inherent advantages when it comes to memory usage. A single instruction can precisely specify a long sequence of memory addresses. Consequently, the hardware has considerable advance knowledge regarding memory references, can schedule these accesses efficiently, and need access no more data than required. In addition, a vector memory operation can amortize start-up latencies over a potentially long stream of vector elements. Research has shown that by coupling ILP techniques with a DLP engine, up to 100 cycles of main memory latency can be tolerated with only slightly degraded performance.[7] Regarding memory bandwidth, a DLP machine can much more effectively use whatever amount of bandwidth it has by requesting several data items with a single memory address.

*Parallelism inside a program.* For designers to achieve

high performance, they will have to extract large amounts of operation parallelism. Machines exploiting ILP have applied hardware techniques to discover instruction parallelism. Meanwhile, DLP machines have relied on the compiler to discover parallelism and convey it to the hardware through vector instructions. Research has highlighted the limitations to expect when designers try to exploit ILP.[8] It seems reasonable, therefore, to exploit all available parallelism in a program, at both the instruction and the data level.

**Performance analysis of ILP machines.** What happens when we scale up today's 4-wide-issue superscalar processors to 16 issues? To find out, we modeled a superscalar processor and ran it against seven highly vectorized SPECfp benchmark programs. Figure 2 presents our results in terms of IPC.

Figure 2a presents, for each program, four performance bars. The first one corresponds to a model of a real machine. It simulates both a real control—that is, assumes real branch penalties—and a real memory system. In the other three bars, we eliminated the real control and real memory restrictions and simulated ideal machines, which have either perfect control, perfect memory, or both. We achieved perfect control by assuming that all branches are correctly predicted and by enlarging the reorder buffer up to 512 entries. We simulated perfect memory by assuming that all loads and stores hit in the first-level cache with a latency of one cycle. We performed the simulations with the SimpleScalar tool set.[9]

Notice that real performance was about 25% of peak (an IPC of 1 out of 4). By assuming ideal conditions, IPC did increase substantially, but it hardly reached 50% of the peak rate. When we scaled up the machine to a 16-issue width with four times more hardware resources, performance certainly improved but efficiency worsened. That is, although we went from an average IPC of around 2 up to an average IPC closer to 5, on the 16-wide-issue machine we reached only about 30–40% of the peak execution rate.

## Proposed SMV processor

Our design approach to high-performance processing, which could accommodate a billion transistors, is based on several technology assumptions. We made these according to current technology projections:[10]

- Packaging technology will allow around 2,000 pins connected to a single die.
- A billion transistors will fit on a single die. Nonetheless, power and thermal considerations may force a restricted use of these transistors. We assume that at least half of all transistors will have to be used in memory structures, such as caches and/or DRAMs. Memory structures consume and dissipate much less heat than logic gates.
- Clock distribution will be a major design problem due to wire delays. Designs will require some degree of regularity, in the sense of independent major sub-blocks that can work almost asynchronously.
- Sending signals out of the chip—to access whatever external form of main memory is available—will take at least one order of magnitude longer than the internal CPU cycle time. Therefore, we expect external memory references to require between 20 and 100 processor cycles.

Figure 3 (next page) depicts our architecture. It combines DLP with several techniques borrowed from ILP: out-of-order execution, register renaming, and multithreading.[11] If we ignore the vector unit and multithreading features, the block diagram in this figure somewhat resembles that of a Mips R10000.

Multithreading is seen only at the fetch, decode, and commit stages. (In the commit stage, the processor retires in strict order those instructions that have been marked as "done and ready to be completed.") The fetch engine selects one of eight threads and fetches four instructions on its behalf. The decoder renames the instructions, using a per-thread rename table, then sends all instructions to several common execution queues. Inside the queues, the instructions are indistinguishable, and no thread information is kept except in the reorder buffer and memory queue. Register names preserve all dependencies. Independent threads use independent rename tables, which prevents false dependencies and conflicts from occurring.

Except for the fetch, decode, and commit stages, where thread information is important, all other stages look like those of today's superscalar microprocessors. Register files hold pools of physical registers, shared dynamically among threads. At any time, the same or different threads can use all functional units.

**Vector organization.** The vector unit has 128 vector registers, each holding 128 64-bit registers, and has four general-purpose, independent functional units. The number of registers is the product of the number of threads (8) and of physical registers (16) required to sustain good performance on each thread.[7] The length of each register (128 elements) was selected to match that of typical vector machines, such as the Convex C34 or Cray T90. Each vector functional unit processes $K$ pairs of operands and produces $K$ results per cycle. For large values of $K$, for example 8, the designer divides each register into $K$ "lanes," where each lane holds every $K$th element from each vector register. Assuming $K = 8$, lane 0 would hold register elements 0, 8, 16, and so forth, while lane 1 would hold register elements 1, 9, 17, and so on. The key to this replication strategy is that all lanes work completely synchronously and completely independently. Therefore, the dispatch logic in the vector queue only recognizes four vector units.

**Memory port organization.** In Figure 3, the maximum bandwidth for scalar memory references is two words per cycle. Clearly, much more bandwidth is required to keep the $K$-wide vector functional units busy, even for low values of $K$. The solution is to provide two $K$-word-wide data paths connecting the memory system and the vector unit.

On vector memory references, then, the processor can take advantage of stride information. (Stride is the distance between consecutive memory accesses.) For stride-1 accesses, only every $K$th element needs to be requested to the memory system to get $K$ words of data back. Similarly, stride-2 accesses proceed at a rate of $K/2$ words per cycle, stride-4 at $K/4$, and so forth. Once the stride is equal or larger than $K$, the memory operation proceeds at a maximum of one word per cycle. This scheme, while not the most powerful, has two advantages: It cuts down the number of address pins required, and it allows a very inexpensive memory system

Figure 3. Simultaneous multithreaded vector architecture.

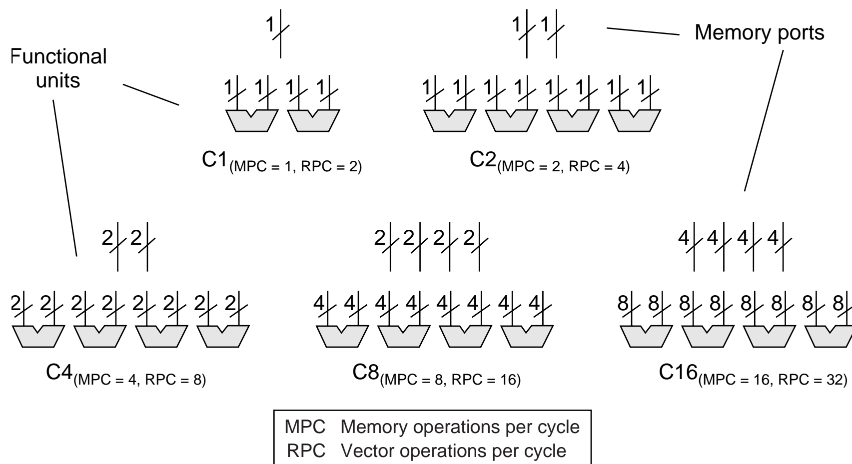



Figure 4. Several possible configurations for future SMV machines, each with a larger number of resources.

to be designed around the SMV. Using the same DRAM technology of current superscalar workstations, we could use the DRAM lines to provide the $K$ words required for stride-1 accesses.

**Execution.** On each cycle, the fetch unit fetches four instructions on behalf of a certain thread. These instructions are renamed and can be any mixture of scalar or vector instructions. Once renamed, each type of instruction moves to its corresponding queue. From the instruction queues, any combination of instructions that are independent and that may or may not belong to the same thread can be dispatched to execute onto the functional units. Out-of-order execution happens

between all types of instructions: scalar and vector. However, the individual operations inside vector instructions are executed in order, albeit in a *K*-way parallel mode. That is, once a vector instruction seizes a resource, it will use it until completion, without allowing other vector instructions to share it.

## SMV performance

Our proposed architecture could be implemented in a variety of configurations, each with different performance potential. The possibilities range from small SMV processors that might appear in a few processor generations up to very large configurations that will require a billion transistors. We discuss the simulations we ran on several possible SMV configurations (Figure 4). The figure shows only the logical organization of the vector unit.

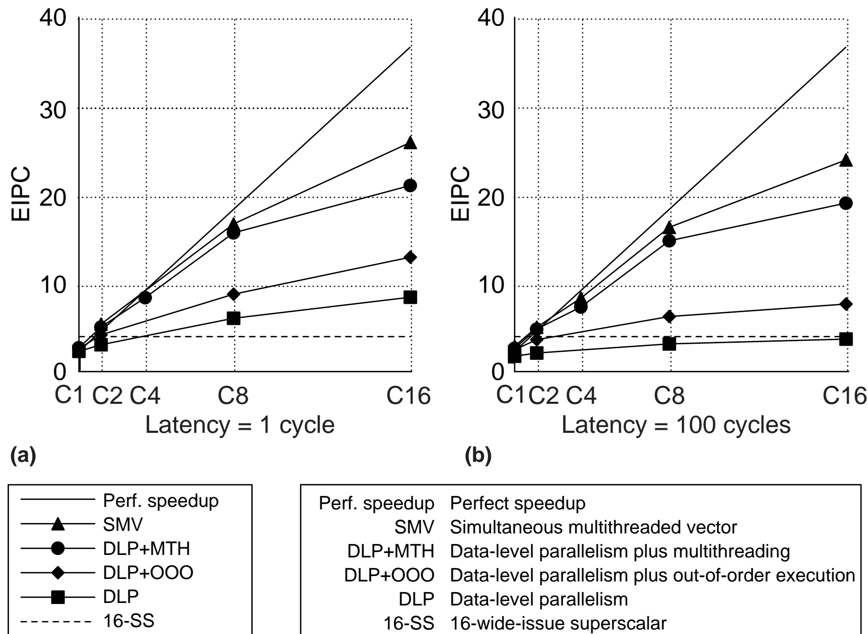



Figure 5. Performance of several architecture paradigms for five possible configurations.

**EIPC measure.** We used a performance measure that indicates how well an ILP processor should perform to match the speed of the SMV machine under study. The measure is EIPC, or equivalent IPC, where IPC indicates the number of instructions executed per cycle in the machine. We define EIPC as follows:

$$\text{EIPC} = \frac{\text{total Mips R10000 instructions}}{\text{SMV cycles}} \tag{2}$$

To compute EIPC, we ran our benchmark programs on a Mips R10000 processor. Using its hardware performance counters, we counted the total number of instructions executed (graduated) for each program. Then, we added all these instructions to get the numerator of Equation 2. Simply, an EIPC of 10 indicates that a superscalar machine should sustain a performance of 10 instructions executed each cycle to match SMV machine performance.

**SMV simulation methodology.** We used a trace-driven approach to estimate the performance achievable with an SMV architecture. We vectorized eight programs, selected from the Perfect Club and SPECfp92 suites, on a Convex C34 machine. We processed them with the Dixie tool,[12] which generates suitable traces for simulation. Then we simultaneously ran all benchmark programs several times on the eight hardware contexts available in the architecture and measured total execution cycles.

We began by randomly ordering the eight programs—flo52, swm256, su2cor, tomcatv, nasa7, hydro2d, bdna, arc2d—to generate a list. Next, we replicated this list three times. We began the simulations by running the first eight programs from the list. When a program completed, we started the next yet-to-be-run program from the list on the hardware context that just became available. We continued until all programs were fully executed at least once.

**EIPC results.** Figure 5 plots the performance results we obtained from the SMV for the five configurations shown in Figure 4. In this figure we assumed two different (extreme) values for memory latency, 1 and 100 cycles.

In the figure, the "perfect speedup" curve plots the best possible speedup achievable when scaling up Figure 4's C1 configuration 16 times. Curve SMV plots the performance of the architecture described previously.

For a memory latency of one cycle, the scalability of the SMV architecture is very good, running almost parallel to the perfect speedup line up to configuration C8. When memory latency increases to 100 cycles, as in Figure 5b, the SMV only degrades its performance by 8%, which is trivial when compared to a 100-fold increase in memory latency.

**Performance factors.** Multithreading, out-of-order execution, and vector instructions all contribute heavily to performance. To find out how much, we tested the three other architectures in Figure 5. Curve DLP represents a traditional, in-order vector architecture with no special ILP features. Curve DLP + OOO represents a vector architecture augmented with out-of-order execution and register renaming.[7] Curve DLP + MTH represents a vector architecture augmented with multithreading (but no out-of-order execution).[12] Finally, for the sake of comparison, a dashed horizontal line, labeled 16-SS, at the bottom of each graph plots the best-case performance of the 16-wide-issue superscalar machine we described earlier.

Each feature contributes different amounts to final performance. Out-of-order execution yields a speedup of 1.47 over pure in-order DLP execution, while multithreading outperforms pure DLP by a factor of 2.39 and DLP + OOO by a factor of 1.63. The SMV machine outperforms DLP + MTH by a

factor of 1.23. Although the SMV machine does not reach the speedup that we would expect from a linear combination of the MTH and OOO features, it is actually quite close.

With a memory latency of 100 cycles, the behavior of all architectures changes substantially. Performance of the DLP, DLP + OOO, and DLP + MTH suddenly degrades severely. The SMV machine, however, experiences only an 8% drop in performance. This difference in behavior results because no single ILP technique can successfully tackle very large memory latencies and, at the same time, fully saturate many resources. Note that, for configurations C1, C2, and C4, performance degradation is not significant in the 100-cycle latency case, but the degradation rises sharply in C8 and C16.

Our simulations confirm that, together, multithreading, vector instructions, and out-of-order execution are necessary for a high-performance machine in an environment with large memory latencies.

## Transistor budget

Given our technology assumptions, one might ask if the SMV is even feasible. Concerning I/O pins, the C16 machine requires a minimum of 4 data ports × 4 (words/port) × 64 (bits/word) = 1,024 pins, plus 4 address ports × 64 = 256 pins, totaling 1,280 pins. This value must be increased to account for error-correcting codes and should then include all necessary power, ground, and clock pins. Although the grand total would be very near the 2,000 limit we have assumed, it might well be possible to fit that many I/O pins into a chip.

With respect to transistor count, we must distinguish between functional units and register space. We considered all functional units in the architecture to be fully general purpose. In the maximum configuration, we have 32 vector units, plus 2 integer and 2 floating-point units. Two floating-point units of the IBM Power2 floating-point chip use 1.3 million transistors in 0.45-μm technology.[13] We will charge 0.7 million transistors for every single functional unit. Therefore, the 36 functional units present in the C16 SMV configuration would require 25.2 million transistors (0.7 × 36).

For storage, we have 128 × 128, or 16,384, individual registers in the vector register file, plus 256 registers in the two scalar register files. At 64 bits, each register holds a total of 1 Mbit. Assuming an equivalent of 10 transistors per bit, the register files would hold 10 million transistors.

Finally, the crossbar connecting the vector register files and the vector functional units would also take a significant amount of space. There are eight read crossbars and eight write crossbars, one pair in each vector subunit. Each read crossbar connects 128 registers to eight inputs to the functional units; each write crossbar connects four functional unit outputs to the 128 registers.

The transistor count here is somewhat insignificant, but the area taken by the connecting wires is crucial. In a very simplistic design, the area would comprise 128 × 64, or 8,192, metal wires coming out of the registers. These would go in the *x*-axis direction, crossed by 8 × 64, or 512, metal wires going in the *y*-axis direction and feeding the functional units. If we use perhaps two metal layers to compact this crossbar layout, and we assume that each metal wire requires 10λ, we have an area close to 40,960λ × 2,560λ, or $104.9 \times 10^6\lambda^2$. To get a transistor equivalent, we divide by the area occupied by a medium-sized transistor, about $500\lambda^2$, yielding an equivalent of 0.21 million transistors per read crossbar. If we factor the necessary cuts, control lines, and switches, we are probably underestimating the total cost by a factor of 4. Therefore, we will charge a total of 4 × 16 × 0.21, or 13.44, million transistors for our 16 crossbars.

Current-generation 4-wide-issue microprocessors are implemented using 2 to 4 million transistors in the processor's logic; the rest is spent in cache implementation. Since the control complexity of our SMV machine is slightly more than that of today's microprocessors, we charge 16 million transistors for the SMV control.

The grand total of our estimates adds up to around 55 million transistors, well under the budget of a billion.

## Using the extra transistors

Assuming our transistor estimates were a little low, we could have roughly 900 million unused transistors if we were to implement the biggest—to date—SMV machine. What would we do with the extra transistors?

**Three possibilities.** First, we could further expand configuration C16 into a hypothetical C32 or C64 machine. However, this would be very difficult, since doubling the number of pins, as C32 would require, greatly exceeds the 2,000 limit dictated by reasonable projections.

A second possibility is multiprocessing: putting more than one SMV processor in the same chip. This possibility is not very attractive because a single SMV processor already uses up 80% of all available cycles on the address pins. Thus, a second SMV processor on the same chip would not have enough address bandwidth.

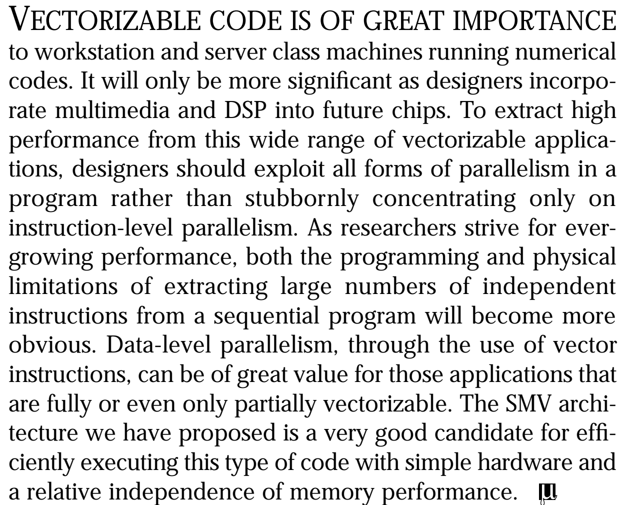
Third, and most likely, we could use the extra transistors to provide a very large cache or memory structure. DRAM structures are dense, and we could conceivably store 1 bit of information in around 1.2 transistors. This could allow roughly 80 to 90 Mbytes of information to be stored inside the processor itself.

**Memory effect.** The most positive effect of large memory on our architecture would be in providing an enormous extra memory bandwidth. Also, researchers looking at the IRAM and PIM proposals claim that a second very positive effect would be reduced latency. However, we have already seen that we incur only an 8% performance hit when going from 1 to 100 cycles of memory latency. For SMV architectures, latency is clearly not an issue.

With the extra bandwidth, we could develop a C32 or C64 configuration inside the chip and use the C16 memory structure to go out of the chip. This scheme would be an alternative to driving our vector unit with four 4-word-wide memory ports. Such a scheme would provide two to four times more bandwidth inside the chip than outside. Simulations show that performance for the C32 machine, with eight ports each 4 words wide (at a memory latency of 1 cycle), has an equivalent IPC of 38. If we go to the C64 machine, for eight ports, each 8 words wide, we reach an equivalent IPC of 48.

**Ramifications.** With the results achieved thus far, we can predict the performance of future SMV processors. For example, if large on-chip RAM is available, the performance would

be 40 to 50 EIPC. The performance would be around 23 EIPC without a RAM structure. This measure would also hold if a certain application did not fit inside the 90 Mbytes of available space and had to go outside the chip.



VECTORIZABLE CODE IS OF GREAT IMPORTANCE to workstation and server class machines running numerical codes. It will only be more significant as designers incorporate multimedia and DSP into future chips. To extract high performance from this wide range of vectorizable applications, designers should exploit all forms of parallelism in a program rather than stubbornly concentrating only on instruction-level parallelism. As researchers strive for ever-growing performance, both the programming and physical limitations of extracting large numbers of independent instructions from a sequential program will become more obvious. Data-level parallelism, through the use of vector instructions, can be of great value for those applications that are fully or even only partially vectorizable. The SMV architecture we have proposed is a very good candidate for efficiently executing this type of code with simple hardware and a relative independence of memory performance.

## Acknowledgments

We thank the anonymous referees for many valuable comments and especially our two editors, Jim Goodman and Doug Burger, for extensive help, feedback, and dedication. We also thank Francisca Quintana for providing the ILP simulations. This work was supported by the Ministry of Education of Spain under contract TIC 0429/95 and by the CEPBA (European Center for Parallelism of Barcelona).

## References

1. J. Wawrzynek et al., "Spert-II: A Vector Microprocessor System," *Computer*, Mar. 1996, pp. 79–86.
2. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1995, pp. 414–425.
3. A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration," *Proc. 23rd Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1996, pp. 90–101.
4. D. Patterson et al., "A Case for Intelligent RAM," *IEEE Micro*, Mar./Apr. 1997, pp. 34–44.
5. D. Burger, S. Kaxiras, and J.R. Goodman, "DataScalar Architectures," *Proc. 24th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1997, pp. 338–349.
6. S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. 24th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1997, pp. 206–218.
7. R. Espasa, M. Valero, and J.E. Smith, "Out-of-Order Vector Architectures," *Proc. 30th Int'l Symp. Microarchitecture*, IEEE Press, Piscataway, N.J., 1997, to appear.
8. N.P. Jouppi and D.W. Wall, "Available Instruction Level Parallelism for Superscalar and Superpipelined Machines," *Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-III Proc.)*, ACM Press, New York, 1989, pp. 272–282.
9. D. Burger and T. Austin, "The SimpleScalar Tool Set, V. 2.0," Tech. Report UWCS-1342, Computer Sci. Dept., Univ. of Wisconsin-Madison, June 1997.
10. A. Yu, "The Future of Microprocessors," *IEEE Micro*, Dec. 1996, pp. 46–53.
11. D.M. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1996, pp. 191–202.
12. R. Espasa and M. Valero, "Multithreaded Vector Architectures," *Proc. Third Int'l Symp. High-Performance Computer Architecture (HPCA-3)*, IEEE Computer Society Press, Los Alamitos, Calif., 1997, pp. 237–249.
13. S.W. White and S. Dhawan, "POWER2: Next Generation of the RISC System/6000 Family," *IBM J. Research and Development*, Vol. 38, No. 5, Sept. 1994, pp. 489–648.

**Roger Espasa** is an assistant professor in the Computer Architecture Department at the Polytechnic University of Catalunya-Barcelona. His research interests are high-performance architectures and memory systems with special emphasis on instruction-level parallelism and efficient use of memory hierarchies. Espasa received an MS and a PhD in computer science from the Polytechnic University of Catalunya-Barcelona.

**Mateo Valero** is a professor in the Computer Architecture Department at the Polytechnic University of Catalunya-Barcelona. His research interests are high-performance architectures with a special emphasis on processor organization, memory hierarchy, interconnection networks, compilation techniques, and computer benchmarking.

Valero received an MS from the Polytechnic University of Madrid and a PhD from the Polytechnic University of Barcelona, both in telecommunications. He is a member of the IEEE, director of the Catalan Center for Computation and Communications, and a member of the Spanish Engineering Academy.

Direct questions to Roger Espasa, Campus Nord, C6-E202, Dept. Arquitectura Computadors, Universitat Politècnica de Catalunya, Gran Capita S/N, 08034 Barcelona, Spain; roger@ac.upc.es.

## Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 153 Medium 154 High 155