Exploiting Instruction and Data Level Parallelism in Future High Performance Processors

Roger Espasa Mateo Valero

Computer Architecture Dept.
Universitat Politècnica de Catalunya-Barcelona
{roger,mateo}@ac.upc.es
http://www.ac.upc.es/hpc

1 Introduction

Historically, there have been two different approaches to high performance computing: *instruction-level parallelism (ILP)* and *data-level parallelism (DLP)*. The ILP paradigm seeks to execute several instructions each cycle by exploring a sequential instruction stream and extracting independent instructions that can be sent to several execution units in parallel. The DLP paradigm, on the other hand, uses vectorization techniques to specify with a single instruction (a *vector* instruction) a large number of operations to be performed on independent data. A few of these vector instructions running concurrently can provide a large operation parallelism for many consecutive cycles.

Figure 1 graphically presents the different microarchitecture generations that have appeared to date in the DLP world. The first DLP machines appeared shortly after the introduction of pipelining in the ILP world. The prototype example of the first machines from this generation is the Cray-1 [12]. Machines exploiting DLP do so by using vector instructions. 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 easily pipelineable and highly parallel. The next generation of DLP machines exploited this parallel semantics to implement multi-pipe functional units; that is, replication of units that allowed processing more than one pair of operands per cycle. However, this batch of machines still used the in-order execution model. ILP techniques such as out-of-order execution or register renaming, so useful to fight memory latency and improve processor throughput in the microprocessor world, have never been used in commercial vector computers.

The importance of Vectorizable Code

What is the importance of regular, vector code in the marketplace ? For many years, the majority of the scientific computing applications have fit very well in the DLP model. There is a large body of vectorizable code that was optimized for yesterday's vector supercomputers which is being run on today's superscalar microprocessors. These codes still retain their DLP characteristics. Moreover, in recent years the fraction of applications that contain highly regular, DLP code has increased. In particular, many DSP and multimedia applications – graphics, compression, encryption – are very well suited for vector implementation [1]. We believe that the fraction of regular vectorizable applications is important enough to deserve special attention in future processors.

Merging the ILP and DLP concepts

We believe that future architectures should merge ILP and DLP to execute regular vectorizable code at a performance level that can not be achieved using either paradigm on its own. The combination of the two techniques yields a single processor with very high performance at low cost and low complexity: the resulting architecture has a relatively simple control unit, tolerates very well memory latency and can be easily partitioned into regular blocks to overcome the wire delay problem of future VLSI implementations. Also, the control simplicity and the implementation regularity both help in achieving very short cycle times. Whatever microarchitecture offers the best performance by the time a billion transistors are available, it should be merged with a vector extension to execute the vector portions present in numerical codes and in multimedia-heavy applications.

SMV Architectures

Our proposal for merging ILP and DLP is a simultaneous multithreaded vector architecture (SMV) which merges three key technologies: vector instructions, out-of-order execution with register renaming and simultaneous multithreaded execution.

Vector instruction sets are a very good match to the characteristics of data-parallel codes. Although other paradigms such as chip multiprocessors or multiscalar processors [15] are very good candidates to extract high performance from these types of codes, vector instruction sets do so very efficiently by using less processor control resources and by directly conveying the notion of parallelism to the hardware in a simple way. Thus, the vector instruction set is our basic building

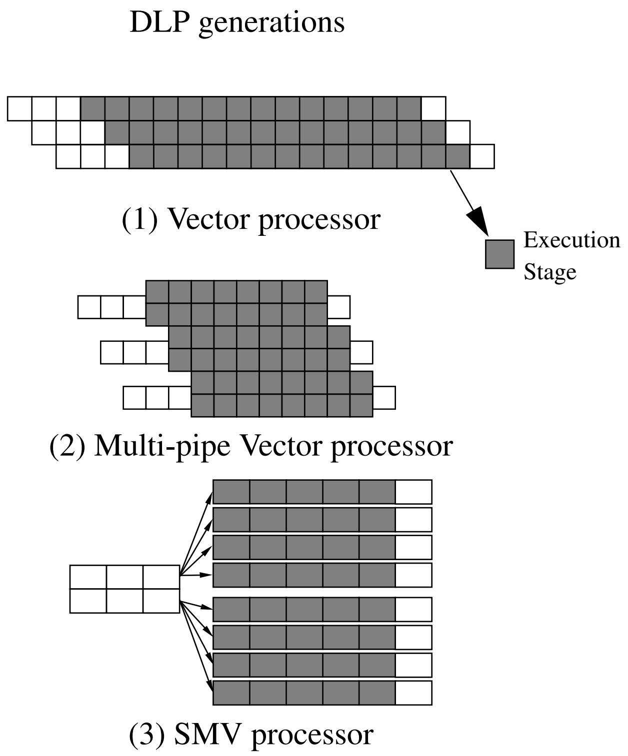

Figure 1: DLP microarchitecture generations.

block for high performance. However, data level parallelism is not enough. For large memory latencies the DLP model on its own suffers a severe performance degradation [5]. At this point out-of-order execution and register renaming come into play. Out-of-order execution reorders instructions and diminishes the interferences that computation and memory instructions cause among themselves. This reordering helps in better using the memory ports and masking memory latency. Register renaming, while not absolutely necessary for a newly defined architecture, helps in managing the processor state and providing precise exceptions, always a difficult thing in vector processors, and can also help in better using the register pool.

When large numbers of resources are considered (memory ports and arithmetic units), out-of-order execution is not enough either. At some point the intrinsic instruction level parallelism of a program limits the number of functional units that can simultaneously be used. Here multithreading can come to a rescue. By multiplexing several threads onto the same processor, all resources can be fully used. Why simultaneous multithreading ? Simultaneous multithreading 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 is the logical step that follows 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.

We will show that, although pure DLP machines are good, adding out-of-order execution to them improves their performance. We will also show that adding multithreading to a pure DLP machine is also very good performance-wise. However, we will show that combining all techniques in a single SMV architecture yields the best performing combination of ILP and DLP.

2 Future Challenges

What are the implications of current technology trends ? Three major areas of concern can be identified if one attempts building billion transistor processors:

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 can be put under control of a single clock, and may 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 already today's most important problem when high performance microprocessors are considered [3, 13]. Currently, microprocessor performance improves at rate of 60% each year while DRAM speed only improves at less than 10% per year. Recent research has pointed out that to reduce this gap the best approach is to merge on the same chip memory and logic. The IRAM [10] and PIM [4] proposals both aim at this direction.

Parallelism inside a program: Parallelism inside a program is understood to be 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 at executing them simultaneously, thus improving the IPC rate. However, there is always a limit as to how many independent tasks can successfully be detected and executed simultaneously. Unless some compiler technology breakthrough happens, current software standards dictate that only a handful of instructions might be available to be executed in parallel.

3 The Scalability Problem of ILP machines

How do the two high performance models face future challenges ? Compare one scalar instruction from to one instruction from the DLP paradigm. A scalar instruction requires many stages to be processed and yet, only specifies a very simple operation. Meanwhile, the vector instruction has a few setup and shutdown stages but many more useful stages where actual computations are carried out. In a scalar instruction, the majority of its execution phases are devoted to bookkeeping activities required to maintain the consistency of the processor. Adding more *execution* resources to a superscalar processor requires many extra *control* resources. On the contrary, in a DLP processor one can speed up vector computations simply by *replicating* functional units (see fig. 1-(2)).

To understand how these semantic differences interact with the three future major bottlenecks, consider the equation that characterizes the speed at which a certain program can be executed on a computer:

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

The time taken to complete a program (T) is equal to the number of instructions executed (N) times the average number of cycles required to execute each instruction (CPI) times the machine cycle time (T_c). Note that CPI is just the inverse of the IPC rate described so far.

Number of Instructions (N)

The DLP paradigm allows reducing the number of instructions (N) by making each vector instruction longer. However, each vector instruction performs many operations and, therefore, can take many cycles to complete (an increase in CPI). Nonetheless, the higher semantic content of vector instructions has indirect benefits: to perform a given task, a vector program has to specify many fewer address computations, loop counter increments and branch computations since these are typically implicit in vector instructions.

Although the reduction of the N factor comes with an increase in the CPI term of the equation, the parallelism implicit in each vector instruction can drastically reduce the CPI factor, simply by adding parallel functional units that process the instruction. 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 are an ever growing fraction of today's microprocessor's cycle time. Regarding wire delays, ILP processors present clear scalability problems. Analysis of the circuit complexity of current superscalar processors [9] show that the wakeup and select logic needed in wide issue machines does not scale favorably when feature size is decreased. These results pose serious questions on the feasibility of very wide centralized superscalar machines (16- or 32-wide). It is widely accepted that high performance will only come with replication of very fast building blocks that work more or less asynchronously. The DLP model matches very well this idea of independent, replicated building blocks. The operations inside a vector instruction do not interchange information and can be physically partitioned into independent sub-blocks that need not communicate to each other.

Cycles Per Instruction (CPI)

Many different factors impact the CPI term of the equation[1] but we will focus on memory access time and instruction level parallelism.

Memory Access Time

Considering that 30% to 40% of all instructions are memory accessing instructions, the larger the difference between CPU and memory speed the longer the time it takes for each memory instruction to complete and hence, the CPI component of the equation grows very fast. In an attempt to compensate for this large performance gap, current superscalar micros use increasingly large caches to keep up performance. Nonetheless, despite out-of-order execution, non blocking caches and prefetching, superscalar micros do not make an efficient use of their memory hierarchies. Since load/store instructions are mixed with computation and setup code, dependencies and resource constraints prevent memory operations to be launched every cycle. The result is that each cache miss turns out to be a very long latency operation.

[1] We will use CPI and IPC depending on the context, since both measures are equivalent.

When high memory latencies are considered, scalar instruction sets have severe shortcomings: to cover up a memory latency of 100 processor cycles, a superscalar machine should have more than a hundred in-flight memory operations. Extracting such a large number of independent memory operations from a sequentially specified program is a daunting task.

On the contrary, vectors have inherent advantages when it comes to memory usage. A single instruction can exactly specify a long sequence of memory addresses. Consequently, the hardware has considerable advance knowledge regarding memory references, can schedule these accesses in an efficient way [11], and needs to access no more data than is actually needed. In addition, a vector memory operation is able to amortize startup latencies over a potentially long stream of vector elements. Recent studies [5, 7] have shown that by using some ILP techniques coupled with a DLP engine, up to 100 cycles of main memory latency can be tolerated with a very small performance degradation. Regarding memory bandwidth, a DLP machine can make a much more effective usage of whatever amount of bandwidth it is provided with, by requesting several data items with a single memory address.

Parallelism Inside a Program

To achieve high performance it will be necessary to extract large amounts of *operation* parallelism. Machines exploiting ILP have focused on hardware techniques to discover instruction parallelism. Meanwhile, DLP machines have relied on the compiler to discover parallelism and to convey this parallelism to the hardware through vector instructions. Research has pointed out the limitations that can be expected when trying to exploit instruction level parallelism [8]. It seems reasonable to try to exploit all kinds of parallelism available in a program, both at the instruction and at the data level.

3.1 Performance Analysis of ILP machines

Before going to our proposal, let's analyze how current paradigms behave when one tries to scale up their performance. Let's take a model of today's 4-wide issue superscalar processors. What happens when we scale it up in a straightforward way up to 16 issue ? Figure 2 presents the results in terms of instruction per cycle (IPC).

In the left graph of fig. 2 we have the performance for 7 highly vectorizable floating point programs belonging to the Specfp suite. For each program, four bars are plotted. The first one corresponds to a model of a real machine. It simulates both a real control (RC), that is, assumes real branch penalties, and a real memory system (RM). In the following bars, the real control and real memory restrictions are eliminated and we simulate ideal machines which have either Perfect Control (PC) or Perfect Memory (PM) or both. Perfect Control is achieved by assuming that all branches are correctly predicted and by enlarging the reorder buffer up to 512 entries. Perfect Memory is simulated by assuming that all loads and stores hit in the first level cache with a latency of 1 cycle. Simulations were performed using the SimpleScalar toolset [2].

The first observation is that *real* performance is about 25% of peak (an IPC of 1 out of 4). Only by assuming ideal conditions IPC increases substantially but hardly reaches 50% of the peak rate. When the machine is scaled up to a 16-wide issue with 4 times more hardware resources performance certainly improves, but efficiency is worse. That is, although we have gone from an average IPC of around 2 up to an average IPC closer to 5, on the 16-issue machine we only reach about 30–40% of the peak execution rate.

4 A High Performance Billion Transistor Processor

We now describe a processor that could be implemented using a billion transistors.

4.1 Technology Assumptions

Our views of future architectures are based on several technology assumptions. Based on current technology projections [18] we assume:

- Packaging technology will allow around 2000 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 (that consume and dissipate much less heat than logic gates), such as caches and/or DRAM memories.
- Clock distribution will be a major design problem due to wire delays. Design will require some degree of regularity, in the sense of independent major sub-blocks that can work almost asynchronously.
- The time to send signals out of the chip and access whatever external form of main memory is available at that time will be at least one order of magnitude larger than the internal cpu cycle time. Therefore we expect external memory references to cost between 20 to 100 processor cycles.

4.2 Simultaneous Multithreaded Vector Architecture (SMV)

The architecture we propose can be seen in figure 3. It combines the DLP paradigm with several techniques borrowed from the ILP world: out-of-order execution,

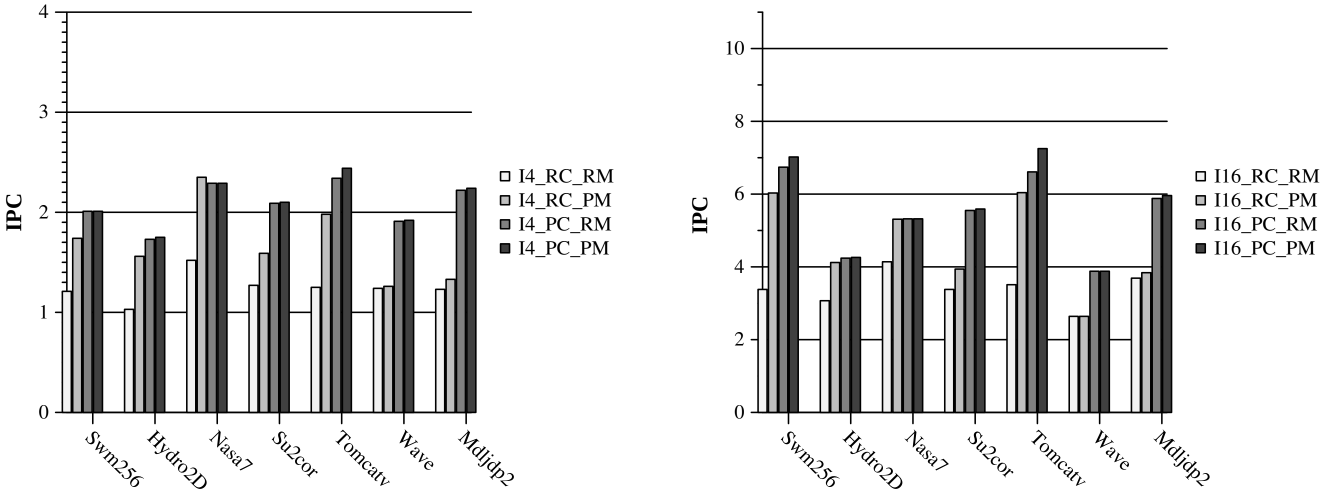

Figure 2: **Left**: Performance, in terms of *instructions per cycle* (IPC) for a superscalar processor. This processor is a close model of a MIPS R10000, issuing 4 instructions per cycle and with 2 floating point units, 2 integer units and 1 memory port. **Right**: Performance, in terms of IPC, for a scaled version of a the superscalar processor. 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 up to 128, and so has been the BTB (from 512 entries up to 2048).

register renaming and multithreading. The combination of all this techniques is rather straightforward. As it can be seen from the figure, if we ignore the vector subunit and the multithreading features, the block diagram of the architecture is almost that of a MIPS R10000.

Multithreading and out-of-order execution are combined as it has been proposed for superscalar processors [16], to yield a simultaneous multithreaded vector architecture. The idea is that multithreading is only seen at the fetch, decode and commit stages. The fetch engine selects one out of 8 threads and fetches several instructions (4) on behalf of it. The decoder renames the instructions using a per-thread rename table and, once renamed, instructions are all thrown into several common execution queues. Inside the queues all instructions are indistinguishable and almost no thread information is kept (only in the reorder buffer and memory queue). All dependences are preserved through register names. Since independent threads use independent rename tables, no false dependences or conflicts can arise.

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

The vector unit is described as having 128 vector registers, each holding 128 64-bit registers, and has four independent functional units (all fully general purpose). Each vector unit processes 8 pairs of elements and generates 8 results per cycle. In order to implement such wide units, *replication* is used. The idea is that each vector register is chopped in as many pieces as required. If one desires K-way parallel vector units, then each vector register is chopped in K pieces. We will refer to each piece as a *vector lane*. Assuming $K = 8$, lane 0 would hold register elements 0, 8, 16, etc., while lane 1 would hold register elements 1, 9, 17, etc. The important feature of this replication strategy is that all lanes work completely synchronously and completely independently. Therefore, from a logical point of view, the dispatch logic in the vector queue only "sees" four vector units.

Execution proceeds as follows. On each cycle, the fetch unit fetches 4 instructions on behalf of a certain thread. These instructions are renamed and can be any mixture of scalar, memory or vector instructions. Once renamed, each type of instruction goes to its corresponding queue. From the instruction queues any combination of instructions that are independent and that can belong or not to the same thread can be dispatched to execute onto the functional units. Out-of-order execution happens between all types of instructions, scalar, memory 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.

The selection of parameters is as follows: the number of physical vector registers is essentially the product of the number of threads (8) and the number of physical registers needed to sustain good performance on each thread (16, see [7] for a discussion). Note that 16 vector register is twice the number of logical registers in the Convex C34 architecture, which is the base for our performance simulations. The length of each vector register (128) has been chosen to match

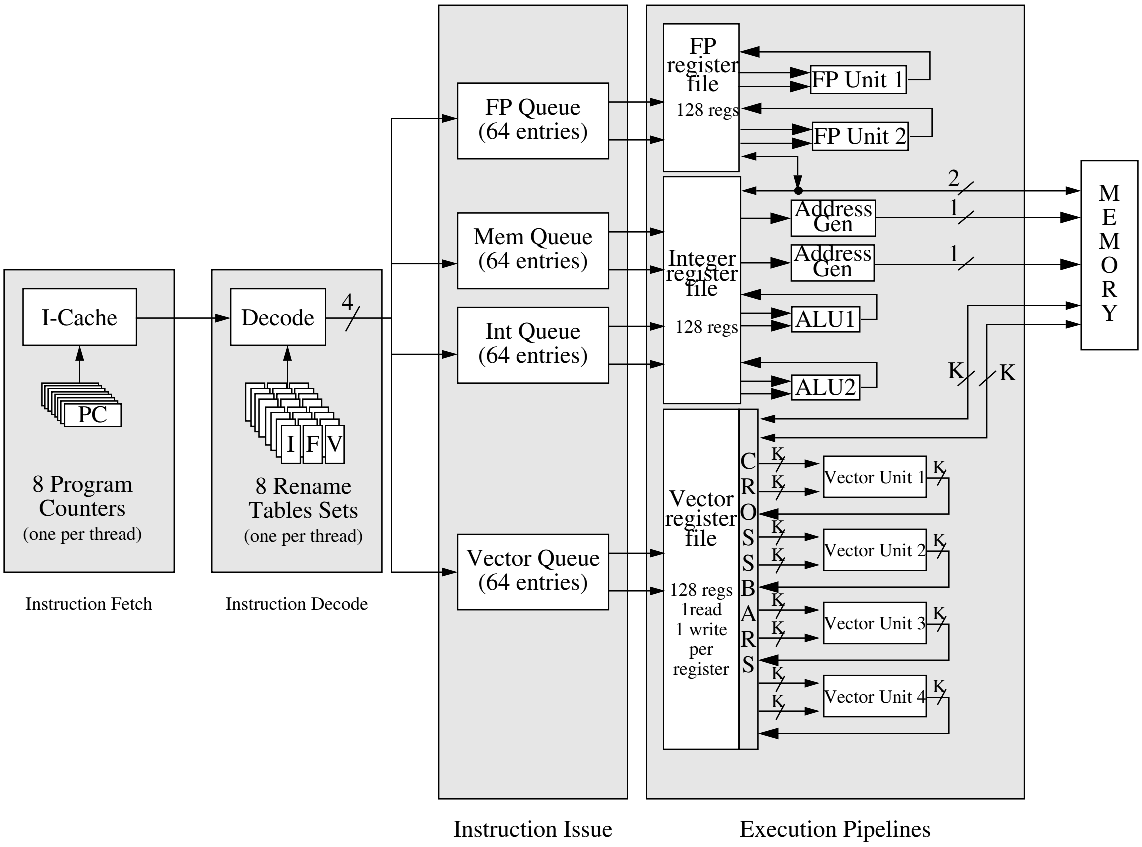

Figure 3: The Simultaneous Multithreaded vector architecture.

that of the Convex registers (and is the same as in the Cray C90 and T90). Simulations indicate that, with ILP techniques, the length of each vector register is less of an issue and that, in some cases, it can be reduced down to 16 or 32 elements without impacting performance.

The organization of the memory ports deserves special attention. In the example presented the maximum bandwidth for scalar memory references is two words per cycle. Clearly, a lot more bandwidth is required in order to keep the K-wide vector functional units busy, even for low values of K. The solution is to provide two K words wide data paths connecting the memory system and the vector unit. The idea is that on vector memory references, the processor only specifies every K^{th} element and the memory system responds by providing between 1 and K words (depending on stride). Stride-1 memory accesses proceed at K words per cycle, stride-2 proceed at $K/2$ words per cycle, stride-4 accesses proceed at $K/4$, and so forth. Once the stride is equal or larger than K, the memory operation proceeds at a maximum of 1 word per cycle. This scheme, while not the most powerful, has two advantages. First, it cuts down the number of *address* pins required. Second, it allows a very cheap memory system to be designed around the SMV. Using the same DRAM technology of current superscalar workstations, the DRAM lines could be used to provide the K words required for stride-1 accesses.

5 Performance of an SMV

We are interested in the performance of a wide range of configurations of the SMV machine, going from small SMV processors that might appear in a few processor generations up to very large configurations that will require a billion transistors.

Figure 4 graphically presents a series of possible configurations for SMV architectures. The figure shows only the logical organization of the vector subunit.

5.1 The EIPC measure

We would like to present a performance measure that indicates how well should an ILP processor perform in order to match the speed of the SMV machine under study. To do so, we define the following indicator of performance:

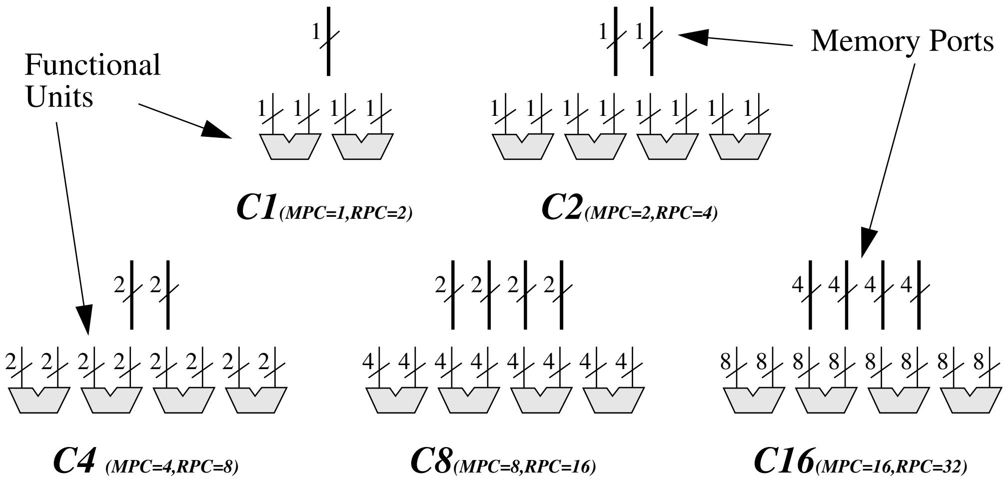

Figure 4: Several possible configurations for future SMV machines, each with a larger number of resources (MPC indicates *memory operations per cycle* and RPC indicates *vector operations per cycle*).

$$EIPC = \frac{total\ MIPS\ R10000\ instructions}{SMV\ cycles} \qquad (2)$$

EIPC stands for "Equivalent IPC" where IPC indicates the number of instructions executed per cycle in the machine. To compute this measure of performance, we run 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 add up together all these instructions to get the numerator of equation 2. The intuitive sense of the EIPC measure is simple: an EIPC of 10 indicates that a superscalar machine should sustain a performance of 10 instructions executed each cycle to match the performance of the SMV machine.

5.2 SMV simulation methodology

We use a trace driven approach to estimate the performance achievable using a SMV architecture. We take the Convex vectorized binaries for eight programs selected from the Perfect Club and Specfp92 suites and process them using the Dixie tool [5], which generates suitable traces for simulation. Then we simultaneously run all benchmark programs several times on the hardware contexts available in the architecture (8) and measure total execution cycles. The simulation procedure is as follows: first we order the 8 programs in some random way (flo52, swm256, su2cor, tomcatv, nasa7, hydro2d, bdna, arc2d) to generate a list and we replicate this list three times. Then we start the simulation by running the first eight programs from the list. When a program completes, the next yet-to-be-run program from the list is started on the hardware context that just became available. We proceed this way until all programs have *fully* executed at least once.

5.3 EIPC results

Figure 5 plots the performance of the SMV for all configurations presented in the previous section and assuming two different (extreme) values for memory latency (1 and 100 cycles).

Let's start by looking only at the top two curves in each graph, labeled "Perf. Speedup" and "SMV". Curve "Perf. Speedup" plots the best possible speedup achievable when scaling up the C1 configuration 16 times. Curve "SMV" plots the performance of the architecture described in the previous sections. Note that in configurations C1–C4, only 4 threads are used in the experiments, while in configurations C8–C16 8 threads were used. The reason is that for the smaller configurations 4 threads are more than enough to saturate the available memory resources.

For a memory latency of 1 cycle, the scalability of the SMV architecture is very good, running almost parallel to the Perfect Speedup line up to configuration C8. When increasing memory latency to 100 cycles (graph on the right) the SMV only degradates its performance by a 8%, which is a small hit when compared to a 100-fold increase in memory latency.

Where does the performance of the SMV architecture come from ? Or, to put it in other words, how much does each of its major features (multithreading, out-of-order execution and vector instructions) contribute to performance ? To answers these questions we have included in figure 5 curves for three other architectures. 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) [6]. Finally, for the sake of comparison, a dashed horizontal line at the bottom of each graph, labeled "16-SS" plots the *best case* perfor-

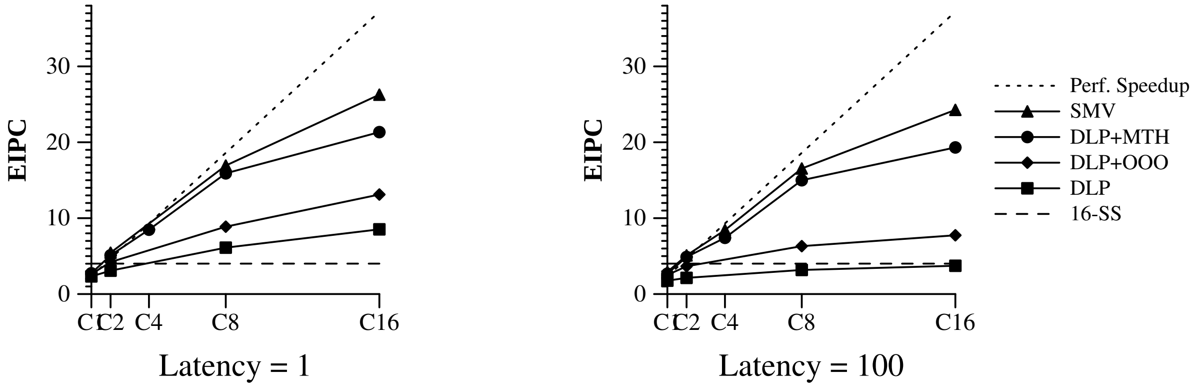

Figure 5: Performance of several architecture paradigms for five possible configurations.

mance of the 16 wide superscalar machine described in section 3.1.

Looking at the ideal memory latency case, we can see that 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 one would expect from a linear combination of the MTH and OOO features it is actually quite close.

When looking at a memory latency of 100 cycles the behavior of all architectures changes substantially. The DLP, DLP+OOO and DLP+MTH suddenly degradate their performance very severely, while the SMV machine doesn't take a large hit in performance (an 8%, as already mentioned). This different behavior is due to the fact that no single ILP technique can successfully tackle very large memory latencies and, at the same time, fully saturate a large number of resources. Note that, for configurations C1, C2 and C4 performance degradation is not very high in the 100 latency case, but the degradation rises sharply in C8 and C16.

The simulations show that the joint use of multithreading, vector instructions and out-of-order execution is necessary if one desires a high performance machine in an environment where large memory latencies are the norm.

6 Transistor Budget

We now turn to the question whether the SMV is feasible under our technology assumptions.

Regarding I/O pins, the C16 machine requires a minimum of $4\, data\, ports \times 4\frac{words}{port} \times 64\frac{bits}{word} = 1024\, pins$, plus $4\, address\, ports \times 64 = 256\, pins$, totaling 1280 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 quite close to the 2000 limit we have assumed, we believe that it might be possible to fit that amount of I/O pins in a chip.

Regarding transistor count, we must distinguish between functional units and register space. All functional units in the architecture were considered 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 (FPU) are implemented using 1.3 million transistors in $0.45um$ technology [17]. We will charge 0.7 million transistors for every single functional unit. Therefore, the 36 functional units present in the C16 SMV would require $0.7 \times 36 = 25.2$ million transistors.

Regarding register storage space, we have $128 \times 128 = 16384$ individual registers in the vector register file plus 256 registers in the two scalar register files. At 64 bits each register yields a total of 1 Mbits. Assuming an equivalent of 10 transistors per bit, the register files would take around 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 8 read crossbars and 8 write crossbars, one pair in each vector subunit. Each read crossbar connects 128 registers to 8 inputs to the functional units and each write crossbar connects 4 functional unit outputs to the 128 registers. Let's consider the cost of the read crossbar. Its transistor count is not really important. Rather, the area taken by the connecting wires is crucial. Taking a very simplistic design, the area would comprise $128 \times 64 = 8192$ metal wires coming out of the registers going in the X-axis direction being crossed by $8 \times 64 = 512$ metal wires going in the Y-axis direction and feeding the functional units. If we use several metal layers (say 2) to compact this crossbar layout and we assume that each metal wire requires 10λ, we end up with an area close to $40960\lambda \times 2560\lambda = 104.9 \times 10^6\lambda^2$. 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 $4 \times 16 \times 0.21 = 13.44$ million transistors for our 16 crossbars.

Current generation 4-wide issue microprocessors are being implemented using 2 to 4 million transistors in the logic part of the processor (the rest is spent in cache implementation) [14]. Since the control complexity of our SMV machine is slightly over 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 billion budget. Next section will discuss possible uses of the remaining transistors.

7 Using the extra transistors

Even if the estimates of previous section are a little bit low, we could have between 800 and 900 million transistors unused after implementing the biggest SMV machine proposed. What would we do with the extra transistors ?

Three possible uses come to mind. First, we have the possibility of further expanding configuration C16 into a hypothetical C32 or C64 machine. However, this path is very difficult since doubling the number of pins (as C32 would require) is way beyond the 2000 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 due to the fact that 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 this is the more likely path, the extra transistors can be used to provide a very large cache or memory structure. DRAM memory structures are very dense, and it is conceivable to store 1 bit of information in around 1.2 transistors. This could allow around 80–90Mb of information to be stored inside the processor itself.

How would the presence of this very large memory affect our architecture ? The most positive effect would be to provide an enormous extra memory bandwidth. Also, researchers looking at the IRAM and PIM proposals claim that a second very positive effect would be latency reduction. However, we have already seen that we only take around a 8% performance hit when going from a 1 cycle to 100 cycles memory latency. Therefore, we think that, for SMV architectures, latency is clearly not an issue.

With the extra bandwidth, instead of driving our vector unit with four 4-wide memory ports, we could develop a C32 or C64 configuration inside the chip and use the C16 memory structure to go out of chip. This scheme would provide 2 to 4 times more bandwidth inside the chip than outside. Simulations show that performance for the C32 machine, with 8 ports each 4-words wide (at a memory latency of 1 cycle), is around an EIPC of 38. If we go to the C64 machine, for 8 ports each 8-words wide we reach an EIPC of 48.

With the results presented so far we can outline the performance of future SMV processors: if large RAM on-chip is available, performance would be around 40–50 EIPC, while if no RAM structure was present or a certain application did not fit inside the 90Mb of available space and had to go out of chip, performance would be around 23 EIPC.

8 Summary

We have presented a simultaneous multithreaded vector architecture that merges the instruction level parallelism and data level parallelism paradigms. We have shown that the joint use of ILP and DLP techniques on highly regular code (vectorizable code) yields a performance much larger than current superscalar microprocessors. Using the EIPC measure, simulations have shown that the SMV machine achieves an equivalent of 15 to 26 scalar instructions per cycle. Moreover, changing the memory latency parameter had only a marginal effect on performance. For the C16 configuration, for example, the performance difference between a 1 cycle memory system and a 100 cycle memory system was only around 8%. The performance achieved does not require very complex fetch and decode hardware. On the contrary, we have shown that with a modest issue control unit EIPCs over 20 are feasible. This control simplicity would have its pay off in allowing a faster cycle time.

The emphasis of this proposal has been on fast execution of highly regular (vectorizable) code. We believe that the fraction of vectorizable code is important enough to deserve special attention, and, moreover, will only get larger as multimedia and DSP functions are incorporated into future chips. Our case is that all forms of parallelism present in a program should be exploited rather than stubbornly concentrating only on exploiting 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 and 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 presented is a very good candidate for efficiently execution this types of codes with simple hardware and a relative independence of memory performance.

Acknowledgments

We would like to thank all the anonymous referees for many valuable comments and especially our two editors, Jim Goodman and Doug Burger, for extensive help, feedback and dedication. We would also like to 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.

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] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report UWCS-1342, University of Wisconsin, June 1997.

[3] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *23rd Annual International Symposium on Computer Architecture*, pages 78–89, Philadelphia, Pennsylvania, May 22–24, 1996.

[4] D. Burger, S. Kaxiras, and J. R. Goodman. DataScalar Architectures. In *24rd Annual International Symposium on Computer Architecture*, Denver , Colorado, June 2–4, 1997.

[5] R. Espasa and M. Valero. Decoupled vector architectures. In *HPCA-2*, pages 281–290. IEEE Computer Society Press, Feb 1996.

[6] R. Espasa and M. Valero. Multithreaded vector architectures. In *HPCA-3*, pages 237–249. IEEE Computer Society Press, Feb 1997.

[7] R. Espasa, M. Valero, and J. E. Smith. Out-of-order Vector Architectures. Technical Report UPC-DAC-1996-52, Univ. Politècnica de Catalunya–Barcelona, November 1996.

[8] N. P. Jouppi and D. W. Wall. Available instruction level parallelism for superscalar and superpipelined machines. *ASPLOS*, pages 272–282, 1989.

[9] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *24rd Annual International Symposium on Computer Architecture*, Denver , Colorado, June 2–4, 1997.

[10] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keaton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, pages 34–44, March/April 1997.

[11] M. Peiron, M. Valero, E. Ayguadé, and T. Lang. Vector multiprocessors with arbitrated memory access. In *22nd Annual International Symposium on Computer Architecture*, pages 243–252, Santa Margherita Ligure, Italy, June 22–24, 1995.

[12] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[13] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *23rd Annual International Symposium on Computer Architecture*, pages 90–101, Philadelphia, Pennsylvania, May 22–24, 1996.

[14] M. Slater. The microprocessor today. *IEEE Micro*, pages 32–44, December 1996.

[15] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 22–24, 1995.

[16] 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 *23rd Annual International Symposium on Computer Architecture*, pages 191–202, Philadelphia, Pennsylvania, May 22–24, 1996.

[17] S. W. White and S. Dhawan. POWER2: Next Generation of the RISC System/6000 family. *IBM Journal of Research and Development*, 38(5):489–648, September 1994.

[18] A. Yu. The future of microprocessors. *IEEE Micro*, pages 46–53, December 1996.