Simultaneous Multithreaded Vector Architecture: Merging ILP and DLP for High Performance

Roger Espasa Mateo Valero*

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

Abstract

The goal of this paper is to show that instruction level parallelism (ILP) and data-level parallelism (DLP) can be merged in a single simultaneous vector multithreaded architecture to execute regular vectorizable code at a performance level that can not be achieved using either paradigm on its own. We will show that the combination of the two techniques yields very high performance at a low cost and a low complexity: We will show that this architecture achieves a sustained performance on numerical regular codes that is 20 times the performance that can be achieved with today's superscalar microprocessors. Moreover, we will show that the architecture can tolerate very large memory latencies, of up to a 100 cycles, with a relatively small performance degradation. This high performance is independent of working set size or of locality considerations, since the DLP paradigm allows very efficient exploitation of a high performance flat memory bandwidth.

1 Introduction

Future high performance architectures will face two major bottlenecks that will ultimately limit computer performance: memory access time and wire delays inside a chip.

Memory access time is already today's most important problem when high performance microprocessors are considered. The disparity between the CPU clock frequency, in the 200Mhz–600Mhz range, and the time required to access off-chip main memory, 10 to 20 times larger, causes significant delays when the processor must idle waiting for data to arrive from memory.

Wire delays are also a major cause of concern. 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. This effect limits the maximum practical size of a fully synchronous processor.

High Performance Computing

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.

The ILP paradigm has been exploited using combinations of several high performance techniques: superscalar out-of-order execution [2, 20, 6], decoupling [18], VLIW execution [7, 5] and multithreading [1, 19, 12]. The current generation of microprocessors all use superscalar execution coupled with a complex memory hierarchy based on several cache levels to attempt executing 4 instructions per cycle.

The DLP paradigm has been exploited using vector instruction sets and appears primarily in parallel vector supercomputers (PVP's) [16, 13, 14]. Unfortunately, traditional vector supercomputers have focused exclusively on DLP and have not exploited ILP techniques as much as current microprocessors.

Future Challenges

The two paradigms for high performance face different problems when we consider the challenges of future processors. The main difference in the two approaches stems from the different semantic contents of their instruction sets. Scalar instruction sets following RISC principles have very simple instructions where each

*This work was supported by the Ministry of Education of Spain under contract 0429/95 and by the CEPBA.

instruction normally specifies one elementary operation (two or three at most). Vector instruction sets, on the other hand, also have simple instructions but each instruction specifies several independent elementary operations to be performed on independent data (the number of operations can be varied at run-time).

Memory Access Time

Memory access time is already today's most important problem when high performance microprocessors are considered [4, 17]. Currently, microprocessor performance improves at rate of 60% each year while DRAM speed only improves at less than 10% per year. To overcome the memory latency problem, 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. 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 inflight operations. Extracting such a large number of independent 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. These may be consecutive, may be separated by some fixed stride length, or may be irregularly placed in memory. Consequently, the hardware has considerable advance knowledge regarding memory references, can schedule these accesses in an efficient way, 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. These features would seem to make vector architectures ideal for the current trend toward relatively expensive memory bandwidth and longer memory latencies as measured in processor cycles.

Wire Delays

When wire delays are considered, ILP also presents clear scalability problems. Analysis of the circuit complexity of current superscalar processors [15] 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). By contrast, the DLP model allows parallel (and therefore *independent*) execution of many operations. These operations do not interchange information and can be physically partitioned into independent sub-blocks that need not communicate to each other. This property reduces the amount of wiring between blocks and allows many blocks to work at the same time almost asynchronously.

DLP drawbacks

Despite all the advantages of the DLP paradigm, it has two main drawbacks. First, it is not truly general purpose. While it matches well engineering and numerical codes that require large bandwidths and/or have large data sets it does not allow efficient execution of general purpose irregular applications. Second, recent research [9, 11, 10] has shown that in order to scale to very large memory latencies DLP base vector machines should also incorporate ILP techniques. Unfortunately, this has not been the case in vector machines to date.

What is the importance of regular, data-parallel code in the marketplace ? For many years, the majority of the scientific computing applications have fit very well in the data parallel 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 data parallel characteristics. Moreover, in recent years the fraction of applications that contain highly regular, data parallel code has increased. In particular, many DSP and multimedia applications – graphics, compression, encryption – are very well suited for vector implementation [3]. We believe that the fraction of regular vectorizable applications is important enough to deserve special attention in future processors.

Merging ILP and DLP

The goal of this paper is to show that ILP and DLP can be merged in a single architecture to execute regular vectorizable code at a performance level that can not be achieved using either paradigm on its own. We will try to show that the combination of the two techniques yields very high performance at a low cost and a 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.

It is important to clarify that we do not believe that DLP based vector architectures will replace current ILP based superscalar machines. Rather, this paper tries to make the case that, given enough transistor resources, both paradigms should be implemented together in the same chip.

This paper will present a simultaneous vector multithreaded architecture that combines the ILP and DLP paradigms. We will show that this architecture achieves a sustained performance on numerical regular codes that is 20 times the performance that can be achieved with today's superscalar microprocessors. Moreover, we will show that the architecture can tolerate very large memory latencies, of up to a 100 cycles, with a relatively small performance degradation.

2 The Simultaneous Multithreaded Vector Architecture (SMV)

The architecture we propose can be seen in figure 1. It combines the DLP paradigm with several techniques borrowed from the ILP world: out-of-order execution, 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 [19], 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 T threads and fetches several instructions (4 or 8) 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 and the number of physical registers needed to sustain good performance on each thread (16, see [11] 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 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.

2.1 Machine configurations

We will look at several different variations of the SMV architecture, changing the T, N and K parameters (refer to fig. 1). Depending on the number of threads running concurrently on the machine (either 2, 4 or 8) the hardware required varies. For example, 32 physical vector registers are enough two support execution of 2 threads (16 registers per thread), while the 8-thread machine requires 128 different vector physical registers. Also, the fetch bandwidth needs to be adjusted depending on the number of contexts. We will present data for three values of K ($K = 1$, $K = 4$ and at the very high end, $K = 8$), in order to show a range of possible performance points that future architectures combining ILP and DLP might achieve.

The three main configurations under study can be

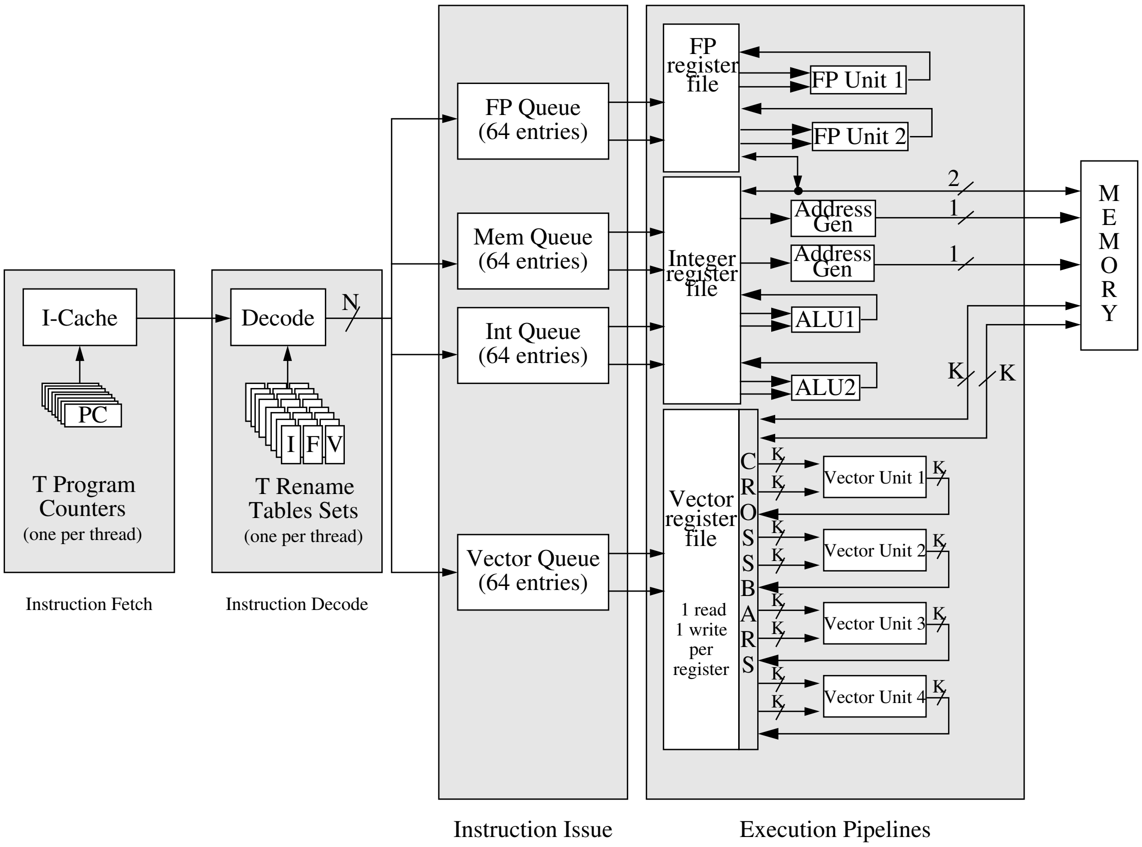

Figure 1: The Simultaneous Multithreaded vector architecture.

seen in table 1. The main differences between them is in the peak number of memory words that each machine can transfer per cycle. Configuration C1 has one port that can transfer one word per cycle, configuration C4 has a 4-wide memory port that can transfer, for vector stride-1 accesses, a peak of 4 words per cycle. Finally, configuration C16 has *two* memory ports each being 8-way wide, thus being able to reach a peak transfer of 16 words per cycle. We have chosen a 2×8 port scheme instead of having a single 16-wide memory port to maximize the efficiency of each port. Our measurements indicate that 87% of all vector memory accesses are performed using stride-1, 3.2% using stride 2, 1.7% stride 4 and the remaining 8% uses strides larger than 16. For stride-1 accesses, both schemes work equally well, provided there are always at least two vector loads ready to go. This is fairly common since we are using both multithreading and out-of-order execution to maximize occupation of all resources. For strides larger than 16 and for gather/scatter instructions, it is better two have two ports that work at 1 word per cycle than having only a 16-wide port that can only transfer one word per cycle.

In all configurations, we always maintain a 1:2 ratio between memory words that can be transferred per cycle (MPC) and the number of vector operations that can be performed per cycle (VOPC). The C1 machine can do 2 VOPC, the C4 machine does 8 VOPC and the C16 machine can perform up to 32 vector operations per cycle. In the biggest configuration under study, the C16 with 8 threads, we have increased also the number of scalar functional units, to avoid that these become a bottleneck.

From the point of view of control complexity, the 2 and 4 thread machines (in all configurations) are relatively close to today's superscalar architectures. On each cycle they fetch 4 consecutive instructions from a single thread, and decode and rename also 4 instructions per cycle. Their scalar register files and the number of scalar functional units are about the same size of a MIPS R10000 processor or of an ALPHA 21264.

3 Simulation Environment

To evaluate the performance of DLP techniques on regular code, we required a set of benchmarks that were highly vectorizable. We took the Perfect Club and Specfp92 benchmark suites and compiled all their

Config	T	Fetch (N)	Regs A	Regs S	Regs V	Scal FU A	Scal FU S	Vect FU Num	Vect FU Width (K)	Mem Ports Num	Mem Ports Width (K)'
C1	2	4	64	64	32	2	2	2	1	1	1
	4	4	64	64	64	2	2				
C4	2	4	64	64	32	2	2	2	4	1	4
	4	4	64	64	64	2	2				
C16	2	4	64	64	32	2	2	4	8	2	8
	4	4	64	64	64	2	2				
	8	8	128	128	128	4	4				

Table 1: Machine parameters for the SMV configurations under study.

Program	Suite	#insns S	#insns V	#ops V	% Vect	avg. VL
swm256	Spec	6.2	74.5	9534.3	99.9	127
hydro2d	Spec	41.5	39.2	3973.8	99.0	101
arc2d	Perf.	63.3	42.9	4086.5	98.5	95
flo52	Perf.	37.7	22.8	1242.0	97.1	54
nasa7	Spec	152.4	67.3	3911.9	96.2	58
su2cor	Spec	152.6	26.8	3356.8	95.7	125
tomcatv	Spec	125.8	7.2	916.8	87.9	127
bdna	Perf.	23.9	19.6	1589.9	86.9	81

Table 2: Basic operation counts for the Perfect Club and Specfp92 programs (Columns 3–5 are in millions).

programs on a Convex C3400 machine. Then we selected the 8 programs that achieved at least 85% vectorization.

Table 2 presents some statistics for the selected programs. Column number 2 indicates to what suite each program belongs. Next two columns present the total number of instructions issued by the decode unit, broken down into scalar and vector instructions. Column five presents the number of operations performed by vector instructions. Each vector instruction can perform many operations (up to 128), hence the distinction between vector instructions and vector operations. The sixth column is the percentage of vectorization of each program. We define the percentage of vectorization as the ratio between the number of vector operations and the total number of operations performed by the program (i.e., column five divided by the sum of columns three and five). Finally column seven presents the average vector length used by vector instructions, and is the ratio of vector operations and vector instructions (columns five and four, respectively).

This paper will use a trace driven approach to estimate the performance achievable using a simultaneous multithreaded architecture. We take the Convex vectorized binaries for the eight programs selected and process them using the Dixie tool [8]. The dixie-modified binaries are run on the Convex machine. These runs produce the desired set of traces that accurately represent the execution of the programs. The traces are then fed to a cycle level simulator of the SMV machine.

To evaluate the performance of the simultaneous multithreaded vector architecture, we run the eight benchmark programs simultaneously on the 2, 4 or 8 contexts available in the architecture and measure the 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). Then we start the simulation by running as many programs concurrently as the degree of the machine allows. When a program completes, the next yet-to-be-run program from the list is started on the hardware context that just became available. This methodology ensures that we are always performing exactly the same amount of work (that is, we always execute the 8 programs to completion) and this allows comparison of the SMV performance against a superscalar machine. We note that this procedure has an important drawback (which shows especially in the 8 thread machine): since not all programs require the same time to execute, there is a fraction of time where we have much less than 8 threads running concurrently. Consequently, the results presented should be taken as a first conservative approximation.

4 Performance Results

To be able to compare the performance of the SMV against current superscalar microprocessors we define the following indicator of performance:

$$EIPC = \frac{total\ MIPS\ R10000\ instructions}{SMV\ cycles} \quad (1)$$

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 the 8 programs on a MIPS R10000 processor. Using its hardware performance counters, we counted the total number of instructions executed (graduated) for each program. The result is shown in table 3. Then, we add up together all these instructions to get the numerator of equation 1. Table 3 also shows the total number of cycles required to execute each program (in millions) and the resulting IPC (the ratio of columns 1 and 2).

The intuitive sense of the EIPC measure is simple: an EIPC of 10 indicates that a superscalar machine

Program	R10000 instrs.	R10000 cycles	R10000 IPC
swm256	11466	20895	0.5
hydro2d	5759	5518	1.0
arc2d	5035	7144	0.7
flo52	1291	970	1.3
nasa7	6897	16830	0.4
su2cor	5135	6348	0.8
tomcatv	1453	2428	0.6
bdna	2501	2501	1.0
avg.	–	–	0.8

Table 3: Performance of the benchmarks when run on an R10000 processor. First column is number of executed instructions (in millions). Second column is total execution cycles (in millions) and column IPC is the ratio of columns 1 and 2.

should sustain a performance of 10 instructions executed each cycle to match the performance of the SMV machine introduced in this paper.

Figure 2 plots the performance of the three configurations of the SMV under study in terms of equivalent IPC. For each configuration we have performed three experiments varying main memory latency from 1 to 100 cycles, and plot results for 2, 4 and 8 threads (where applicable). As it can be seen, performance for this highly vectorizable codes is very good, and can be 20 times over the performance of a current state-of-the art microprocessor.

Starting with configuration C1, we see that exploiting both ILP and DLP yields a performance that is 3.4 times the performance of an R10000, with relatively simple hardware. That is, the peak performance of the C1-SMV architecture is somewhere over 3 operations per cycle (2 VOPC plus 1 MPC plus a few scalar instructions that are run in parallel with the vector code). In terms of EIPC, the C1-SMV achieves 2.65 for 2 threads and 2.7 for four threads. Two points are both noting: first, increasing the number of threads does not yield any specific improvement and second, memory latency does not significantly affect performance (from 1 cycle to 100 cycles there is less than a 2% degradation). The reason behind both points is that two threads executing instructions out-of-order are almost enough to saturate the memory port. Once the memory port is almost always busy, there is no point in adding more threads. Moreover, since the port is almost always sending requests, main memory latency of each individual request is hidden and the architecture becomes very tolerant to large memory latencies.

Configuration C4 has 4 times the number of resources (vector and memory) of configuration C1. With 2 threads, the EIPC is boosted up to 7.49, which represents almost a three-fold improvement over C1 (a factor of 2.82). With four threads, EIPC reaches 7.73, an improvement of 2.85 over C1. It is interesting to note that this EIPC is reached fetching and decoding just 4 instructions per cycle. The fact that the EIPC is over 4 shows the semantic advantage of exploiting DLP: although only 4 instructions are fetched per cycle, each instruction contains more operations than its scalar counterpart, and, therefore, a higher-than-4 EIPC can be achieved.

Why is the speedup of configuration C4 over C1 clearly smaller than 4 ? Applying Amdahl's law to the distribution of memory strides, we observe the following: although the bus is 4-wide, only 87% of all memory accesses make full use of it (those that are stride-1). A 3.2% only use half the available bandwidth (stride-2) and the remaining 9.7% of accesses proceed at one word per cycle (using only 1/4 of the bandwidth). Therefore, the average number of words transferred per cycle is bounded by

$$\frac{1}{0.097 + \frac{0.032}{2} + \frac{0.87}{4}} = 3.02 \qquad (2)$$

Thus, the maximum speedup possible is bounded by this factor, and the actual speedups we are observing are very close to it (2.85).

We now turn our attention to configuration C16. First, let's use again Amdahl's law to derive the maximum throughput that its two 8-way memory ports can deliver. Using similar values as above, and knowing that 1.7% of all accesses are stride-4, we can derive the maximum throughput as:

$$\frac{1}{\frac{0.08}{2} + \frac{0.017}{4} + \frac{0.032}{8} + \frac{0.87}{16}} = 9.74 \qquad (3)$$

Note that there are `two` memory ports and, hence, the 8% of accesses that have a stride larger than 8 proceed at a relative speed of 2. It is very interesting to repeat this exercise assuming a single 32-wide memory port. The resulting equation is:

$$\frac{1}{0.08 + \frac{0.017}{8} + \frac{0.032}{16} + \frac{0.87}{32}} = 8.98 \qquad (4)$$

The effective throughput of this wider bus is actually lower! This shows that there is a compromise between the amount of ILP and DLP that can be exploited in the memory system. Amdahl's law tends to favor many buses that are relatively thin, but the amount of independent busses that can be successfully exploited is limited by the available ILP in the program. Reciprocally, the DLP paradigm tends to favor a few very wide busses, but the relative importance of non-unit strides ultimately limits the maximum width that can provide speedups.

Configuration C16 achieves a maximum EIPC of 21.14, a factor of 7.83 over configuration C1 which represents an 80% of the best possible speedup. Several differences with the previous configurations appear. First, a memory latency sensibly influences performance. For example, consider the three data points for the 8-thread machine. Between the best and worst cases, there is about a 15% difference in EIPC. This is a relatively low impact considering that memory latency is being increased 100-fold and yet, it is much larger than the 2% impact seen in configurations C4 and C1. Again, this is due to the behavior of the

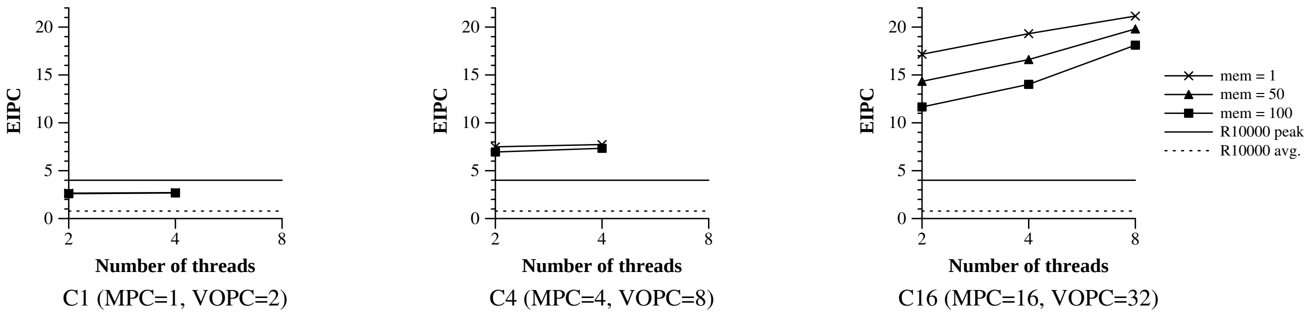

Figure 2: Performance of the SMV.

memory ports. In the C1 case, the effective throughput and maximum memory throughput are both 1. In the C4 case, the effective throughput is 3.02 and the maximum is 4, yielding a 75.5% efficiency. Unfortunately, for the C16 machine, efficiency drops down to 61% (9.74/16) and this makes memory latency more relevant. The speed at which operations are processed inside the vector pipeline exacerbates the effective latency seen for non-unit stride memory accesses. Second, adding more threads in configurations C1 and C4 did not yield any additional speedups. Here, the shape of the curves clearly shows that the more threads, the more likely it is to saturate the very wide functional units present in the architecture.

The C16 machine shows that even with a relatively simple machine (the 2 thread case) that is about the same complexity as an R10000 in the fetch and decode units, we can achieve a very good EIPC exploiting data level parallelism through the vector unit. Considering the 100 cycle memory latency, the 2-thread machine achieves an EIPC of 11.6, which is about 14 times larger than the average IPC for the R10000 (shown in table 3).

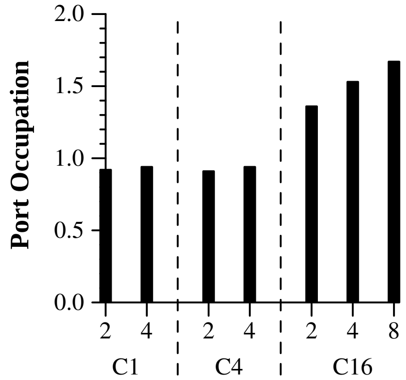

Figure 3: Memory port occupation for the different SMV configurations. X-axis indicates number of threads.

5 Memory Port Occupation

This section will look at the occupation of the memory port(s) of the three configurations under study. Previous work on the DLP paradigm [9, 10, 8] shows that the single most important resource is usually the memory port. There is a direct relationship between the occupation of the address bus and final performance.

Figure 3 plots the occupation of the memory port for each configuration. Note that for C1 and C4, the peak occupation is 1, while for C16, the peak occupation is 2. The data shows that both C1 and C4 are making an almost optimal use of their ports. The few 4–6% of port idle time is very difficult to eliminate, once again due to the application of Amdahl's law to the amount of scalar code, and due to stride behavior. For the C16 machine, unfortunately, port occupation is relatively worse, since for 2 threads only achieves 1.36 out of 2. This is due to a couple of effects. First, for such a powerful machine, two threads are not enough to continuously feed the memory ports. Second, as already mentioned, the negative effect of non-unit strides also impacts the rate at which subsequent memory operations can be dispatched.

6 Summary

This paper has 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 21 scalar instructions per cycle.

We have presented data for three different SMV configurations, ranging from a low-bandwidth machine able to move one memory word per cycle (C1) to a mid-range machine moving four words per cycle (C4) up to a very high end machine that can transfer 16 words per cycle C16). Even for the low-end machine, performance of the SMV was 3.39 times larger than the IPC achieved by an R10000. The C4 machine yielded a speedup of 9.6 and the C16 machine went up to a speedup of 27.

Changing the memory latency parameter had only a marginal effect on performance. The C1 and C4 machine showed less than a 2% performance impact when going from a memory system with a latency of 1 cycle to a latency of 100 cycles. In the C16 machine, the effect of memory latency was somewhat larger, around 15%, although small when compared to the 100 fold degradation in memory response time.

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 paper has been on fast execution of highly regular (vectorizable) code. Our case is that a vector instruction set able to exploit data level parallelism should be merged with whatever ILP paradigm provides better performance for non-regular code. The resulting architecture would have a very powerful unit (the vector unit) for executing regular numerical code and multimedia intensive tasks at high performance, and would rely on standard ILP techniques for codes not amenable to the DLP paradigm. This combination of paradigms yields a very high performance architecture that sustains very large IPC's independently of working set size or locality considerations.

References

[1] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] D. Anderson, F. J. Sparacio, and F. M. Tomasulo. The IBM System/360 model 91: Machine philosophy and instruction handling. *IBM Journal of Research and Development*, 11:8–24, January 1967.

[3] 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.

[4] 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.

[5] P. Colwell, R. Nix, J. O'Donnell, D. Papworth, and R. Rodman. A VLIW architecture for a trace scheduling compiler. In *ASPLOS-II*, pages 180–192, Palo Alto, CA, October 1987.

[6] R. Colwell and R. Steck. A 0.6-μm BiCMOS Microprocessor with Dynamic Execution. In *International Solid-State Circuits Conference*, pages 176–177, Piscataway, N.J., 1995. IEEE.

[7] J. Ellis. *Bulldog: a compiler for VLIW architectures*. MIT Press, 1985.

[8] R. Espasa. *Advanced Vector Architectures*. PhD thesis, Universitat Politècnica de Catalunya, February 1997. ftp://ftp.ac.upc.es/pub/reports/DAC/1997/UPC-DAC-1997-10.ps.Z.

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

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

[11] 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.

[12] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *ISCA*, pages 136–145, 1992.

[13] A. Iwaya and T. Watanabe. The parallel processing feature of the NEC SX-3 supercomputer system. *Intl. Journal of High Speed Computing*, 3(3&4):187–197, 1991.

[14] W. Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18(8):947–954, August 1992.

[15] 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.

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

[17] 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.

[18] J. E. Smith, S. Weiss, and N. Y. Pang. A Simulation Study of Decoupled Architecture Computers. *IEEE Transactions on Computers*, C-35(8):692–702, August 1986.

[19] 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.

[20] K. C. Yager. The Mips R10000 Superscalar Microprocessor. *IEEE Micro*, pages 28–40, April 1996.