5

Computer Architecture and Design

Jean-Luc Gaudiot
University of Southern California

Siamack Haghighi
Intel Corporation

Binu Matthew
University of Utah

Krste Asanovic
MIT Laboratory for Computer Science

Manoj Franklin
University of Maryland

Donna Quammen
George Mason University

Bruce Jacob
University of Maryland

Introduction

Jean-Luc Gaudiot

It is a truism that computers have become ubiquitous and portable in the modern world: Personal Digital Assistants, as well as many various kinds of mobile computing devices are easily available at low cost. This is also true because of the ever increasing presence of the Wide World Web connectivity. One should not forget, however, that these life changing applications have only been made possible by the phenomenal

0-8493-0885-2/02/$0.00+$1.50

6. Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek, Lopez-Lagunas, Abelardo, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing, in *Proc. 31st Annual International Symposium on Microarchitecture,* Dallas, TX, November 1998.
7. Intel Corporation. *Itanium Processor Microarchitecture Reference for Software Optimization.* Intel Corporation, March 2000.
8. Intel Corporation. *Intel IA-64 Architecture Software Developer's Manula, Volume 3: Instruction Set Reference.* Intel Corporation, January 2000.
9. Intel Corporation. *IA-64 Application Developer's Architecture Guide.* Intel Corporation, May 1999.
10. P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing,* 7, 1993.
11. R. E. Hank, S. A. Mahlke, J. C. Gyllenhaal, R. Bringmann, and W. W. Hwu, Superblock formation using static program analysis, in *Proc. 26th Annual International Symposium on Microarchitecture,* Austin, TX, pp. 247–255, Dec. 1993.
12. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, Effective compiler support for predicated execution using the hyperblock, in *Proc. 25th International Symposium on Microarchitecture,* pp. 45–54, December 1992.
13. James C. Dehnert, Peter Y. T. Hsu, Joseph P. Bratt, Overlapped loop support in the Cydra 5, in *Proc. ASPLOS 89,* pp. 26–38.
14. Alexander Klaiber, *The Technology Behind Crusoe Processors.* Transmeta Corp., 2000.

5.3 Vector Processing

Krste Asanovic

Introduction

For nearly 30 years, vector processing has been used in the world's fastest supercomputers to accelerate applications in scientific and technical computing. More recently vector-like extensions have become popular on desktop and embedded microprocessors to accelerate multimedia applications. In both cases, architects are motivated to include data parallel instructions because they enable large increases in performance at much lower cost than alternative approaches to exploiting application parallelism. This chapter reviews the development of data parallel instruction sets from the early SIMD (single instruction, multiple data) machines, through the vector supercomputers, to the new multimedia instruction sets.

Data Parallelism

An application is said to contain data parallelism when the same operation can be carried out across arrays of operands, for example, when two vectors are added element by element to produce a result vector. Data parallel operations are usually expressed as loops in sequential programming languages. If each loop iteration is independent of the others, data parallel instructions can be used to execute the code. The following vector add code written in C is a simple example of a data parallel loop:

```
for (i=0; i<N; i++)
     C[i] = A[i] + B[i];
```

Provided that the result array `C` does not overlap the source arrays `A` and `B`, the individual loop iterations can be run in parallel. Many compute-intensive applications are built around such data parallel loop kernels. One of the most important factors in determining the performance of data parallel programs is the range of vector lengths observed for typical data sets. Vector lengths vary depending on the application, how the application is coded, and also on the input data for each run. In general, the longer the vectors, the greater the performance achieved by a data parallel architecture, as any loop startup overheads will be amortized over a larger number of elements.

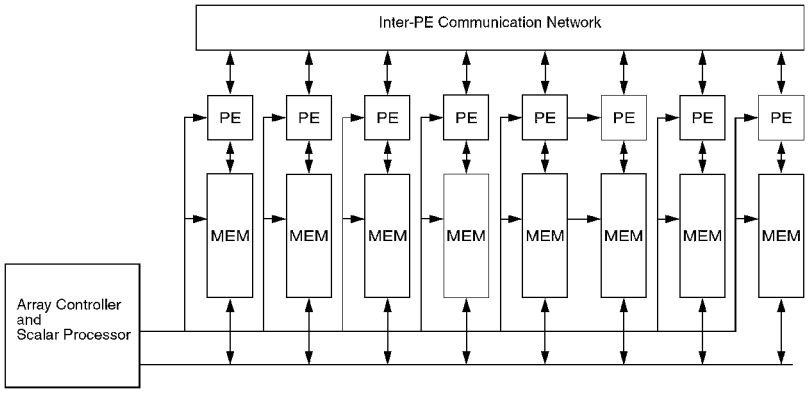

FIGURE 5.8 Structure of a distributed memory SIMD (DM-SIMD) processor.

The performance of a piece of vector code running on a data parallel machine can be summarized with a few key parameters. R_n is the rate of execution (for example, in MFLOPS) for a vector of length n. R_∞ is the maximum rate of execution achieved assuming infinite length vectors. $N\_$ is the number of elements at which vector performance reaches one half of R_∞. $N\_$ indirectly measures startup overhead, as it gives the vector length at which the time lost to overheads is equal to the time taken to execute the vector operation at peak speed ignoring overheads. The larger the $N\_$ for a code kernel running on a particular machine, the longer the vectors must be to achieve close to peak performance.

History of Data Parallel Machines

Data parallel architectures were first developed to provide high throughput for supercomputing applications. There are two main classes of data parallel architectures: distributed memory SIMD (single instruction, multiple data [1]) architecture and shared memory vector architecture. An early example of a distributed memory SIMD (DM-SIMD) architecture is the Illiac-IV [2]. A typical DM-SIMD architecture has a general-purpose scalar processor acting as the central controller and an array of processing elements (PEs) each with its own private memory, as shown in Fig. 5.8. The central processor executes arbitrary scalar code and also fetches instructions, and broadcasts them across the array of PEs, which execute the operations in parallel and in lockstep. Usually the local memories of the PE array are mapped into the central processor's address space so that it can read and write any word in the entire machine. PEs can communicate with each other, using a separate parallel inter-PE data network. Many DM-SIMD machines, including the ICL DAP [3] and the Goodyear MPP [4], used single-bit processors connected in a 2-D mesh, providing communication well-matched to image processing or scientific simulations that could be mapped to a regular grid. The later connection machine design [5] added a more flexible router to allow arbitrary communication between single-bit PEs, although at much slower rates than the 2-D mesh connect. One advantage of single-bit PEs is that the number of cycles taken to perform a primitive operation, such as an add can scale with the precision of the operands, making them well suited to tasks such as image processing where low-precision operands are common. An alternative approach was taken in the Illiac-IV where wide 64-bit PEs could be subdivided into multiple 32-bit or 8-bit PEs to give higher performance on reduced precision operands. This approach reduces $N\_$ for calculations on vectors with wider operands but requires more complex PEs. This same technique of subdividing wide datapaths has been carried over into the new generation of multimedia extensions (referred to as MX in the rest of this chapter) for microprocessors. The main attraction of DM-SIMD machines is that the PEs can be much simpler than the central processor because they do not need to fetch and decode instructions. This allows large arrays of simple PEs to be constructed, for example, up to 65,536 single-bit PEs in the original connection machine design.

Shared-memory vector architectures (henceforth abbreviated to just "vector architectures") also belong to the class of SIMD machines, as they apply a single instruction to multiple data items. The primary difference in the programming model of vector machines versus DM-SIMD machines is that vector machines allow any PE to access any word in the system's main memory. Because it is difficult to construct machines that allow a large number of simple processors to share a large central memory, vector machines typically have a smaller number of highly pipelined PEs.

The two earliest commercial vector architectures were CDC STAR-100 [6] and TI ASC [7]. Both of these machines were vector memory–memory architectures where the vector operands to a vector instruction were streamed in and out of memory. For example, a vector add instruction would specify the start addresses of both source vectors and the destination vector, and during execution elements were fetched from memory before being operated on by the arithmetic unit which produced a set of results to write back to main memory.

The Cray-1 [8] was the first commercially successful vector architecture and introduced the idea of vector registers. A vector register architecture provides vector arithmetic operations that can only take operands from vector registers, with vector load and store instructions that only move data between the vector registers and memory. Vector registers hold short vectors close to the vector functional units, shortening instruction latencies and allowing vector operands to be reused from registers thereby reducing memory bandwidth requirements. These advantages have led to the dominance of vector register architectures and vector memory–memory machines are ignored for the rest of this section.

DM-SIMD machines have two primary disadvantages compared to vector supercomputers when writing applications. The first is that the programmer has to be extremely careful in selecting algorithms and mapping data arrays across the machine to ensure that each PE can satisfy almost all of its data accesses from its local memory, while ensuring the local data set still fits into the limited local memory of each PE. In contrast, the PEs in a vector machine have equal access to all of main memory, and the programmer only has to ensure that data accesses are spread across all the interleaved memory banks in the memory subsystem.

The second disadvantage is that DM-SIMD machines typically have a large number of simple PEs and so to avoid having many PEs sit idle, applications must have long vectors. For the large-scale DM-SIMD machines, $N\_$ can be in the range of tens of thousands of elements. In contrast, the vector supercomputers contain a few highly pipelined PEs and have $N\_$ in the range of tens to hundreds of elements.

To make effective use of a DM-SIMD machine, the programmer has to find a way to restructure code to contain very long vector lengths, while simultaneously mapping data structures to distributed small local memories in each PE. Achieving high performance under these constraints has proven difficult except for a few specialized applications. In contrast, the vector supercomputers do not require data partitioning and provide reasonable performance on much shorter vectors and so require much less effort to port and tune applications. Although DM-SIMD machines can provide much higher peak performances than vector supercomputers, sustained performance was often similar or lower and programming effort was much higher. As a result, although they achieved some popularity in the 1980s, DM-SIMD machines have disappeared from the high-end, general-purpose computing market with no current commercial manufacturers, while there are still several manufacturers of high-end vector supercomputers with sufficient revenue to fund continued development of new implementations. DM-SIMD architectures remain popular in a few niche special-purpose areas, particularly in image processing and in graphics rendering, where the natural application parallelism maps well onto the DM-SIMD array, providing extremely high throughput at low cost.

Although data parallel instructions were originally introduced for high-end supercomputers, they can be applied to many applications outside of scientific and technical supercomputing. Beginning with the Intel i860 released in 1989, microprocessor manufacturers have introduced data parallel instruction set extensions that allow a small number of parallel SIMD operations to be specified in single instruction. These microprocessor SIMD ISA (instruction set architecture) extensions were originally targeted at multimedia applications and supported only limited-precision, fixed-point arithmetic, but now support single and double precision floating-point and hence a much wider range of applications. In this chapter, SIMD ISA extensions are viewed as a form of short vector instruction to allow a unified discussion of design trade-offs.

Basic Vector Register Architecture

Vector processors contain a conventional scalar processor that executes general-purpose code together with a vector processing unit that handles data parallel code. Figure 5.9 shows the general architecture of a typical vector machine. The vector processing unit includes a set of vector registers and a set of vector functional units that operate on the vector registers. Each vector register contains a set of two or more data elements. A typical vector arithmetic instruction reads source operand vectors from two vector registers, performs an operation pair-wise on all elements in each vector register and writes a result vector to a destination vector register, as shown in Fig. 5.10. Often, versions of vector instructions are provided that replace one vector operand with a scalar value; these are termed vector–scalar instructions. The scalar value is used as one of the operand inputs at each element position.

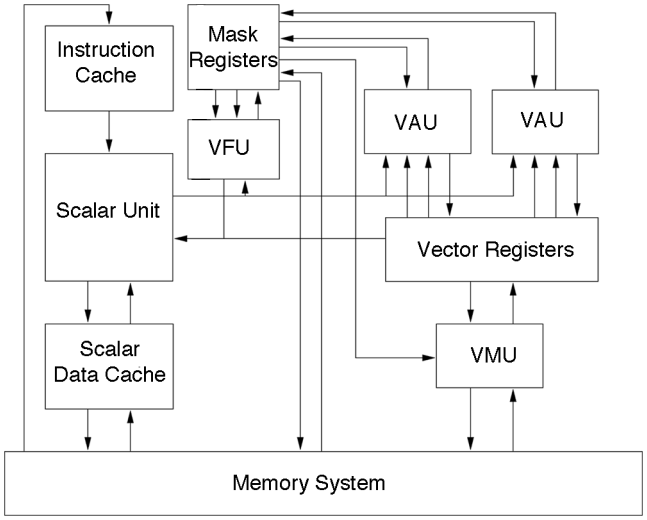

FIGURE 5.9 Structure of a vector machine. This example has a central vector register file, two vector arithmetic units (VAU), one vector load/store unit (VMU), and one vector mask unit (VFU) that operates on the mask registers. (Adapted from Asanovic, K., *Vector Microprocessors,* 1998. With permission.)

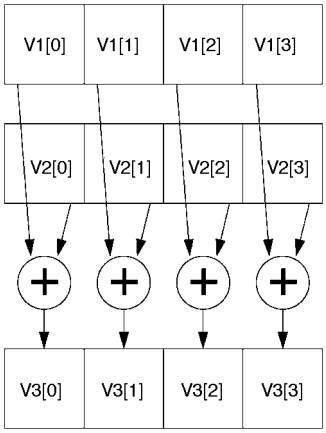

FIGURE 5.10 Operation of a vector add instruction. Here, the instruction is adding vector registers 1 and 2 to give a result in vector register 3.

The vector ISA usually fixes the maximum number of elements in each vector register, although some machines such as the IBM vector extension for the 3090 mainframe support implementations with differing numbers of elements per vector register. If the number of elements required by the application is less than the number of elements in a vector register, a separate vector length register (VLR) is set with the desired number of operations to perform. Subsequent vector instructions only perform this number of operations on the first elements of each vector register. If the application requires vectors longer than will fit into a vector register, a process called strip mining is used to construct a vector loop that executes the application code loop in segments that each fit into the machine's vector registers. The MX ISAs have very short vector registers and do not provide any vector length control. Various types of vector load and store instruction can be provided to move vectors between the vector register file and memory. The simplest form of vector load and store transfers a set of elements that are contiguous in memory to successive elements of a vector register. The base address is usually specified by the contents of a register in the scalar processor. This is termed a unit-stride load or store, and is the only type of vector load and store provided in existing MX instruction sets.

Vector supercomputers also include more complex vector load and store instructions. A strided load or store instruction transfers memory elements that are separated by a constant stride, where the stride is specified by the contents of a second scalar register. Upon completion of a strided load, vector elements that were widely scattered in memory are compacted into a dense form in a vector register suitable for subsequent vector arithmetic instructions. After processing, elements can be unpacked from a vector register back to memory using a strided store.

Vector supercomputers also include indexed load and store instructions to allow elements to be collected into a vector register from arbitrary locations in memory. An indexed load or store uses a vector register to supply a set of element indices. For an indexed load or gather, the vector of indices is added to a scalar base register to give a vector of effective addresses from which individual elements are gathered and placed into a densely packed vector register. An indexed store, or scatter, inverts the process and scatters elements from a densely packed vector register into memory locations specified by the vector of effective addresses.

Many applications contain conditionally executed code, for example, the following loop clears values of `A[i]` smaller than some threshold value:

```
for (i=0; i<N; i++)
     if (A[i] < threshold)
        A[i] = 0;
```

Data parallel instruction sets usually provide some form of conditionally executed instruction to support parallelization of such loops. In vector machines, one approach is to provide a mask register that has a single bit per element position. Vector comparison operations test a predicate at each element and set bits in the mask register at element positions where the condition is true. A subsequent vector instruction takes the mask register as an argument, and at element positions where the mask bit is set, the destination register is updated with the result of the vector operation, otherwise the destination element is left unchanged. The vector loop body for the previous vector loop is shown below (with all stripmining loop code removed).

```
lv va, (ra)          # Load slice of vector A from memory
cmp.lt.vs va, rt     # Set mask where A[i] < threshold
move.vs.m va, r0     # Clear elements of A[i] under mask
sv va, (ra)          # Store updated slice of A to memory
```

Vector Instruction Set Advantages

Vector instruction set extensions provide a number of advantages over alternative mechanisms for encoding parallel operations. Vector instructions are compact, encoding many parallel operations in a single short instruction, as compared to superscalar or VLIW instruction sets which encode each individual operation using a separate collection of bits.

They are also expressive, relaying much useful information from software to hardware. When a compiler or programmer specifies a vector instruction, they indicate that all of the elemental operations within the instruction are independent, allowing hardware to execute the operations using pipelined execution units, or parallel execution units, or any combination of pipelined and parallel execution units, without requiring dependency checking or operand bypassing for elements within the same vector instruction. Vector ISAs also reduce the dependency checking required between two different vector instructions. Hardware only has to check dependencies once per vector register, not once per elemental operation. This dramatically reduces the complexity of building high throughput execution engines compared with RISC or VLIW scalar cores, which have to perform dependency and interlock checking for every elemental result. Vector memory instructions can also relay much useful information to the memory subsystem by passing a whole stream of memory requests together with the stride between elements in the stream.

Another considerable advantage of a vector ISA is that it simplifies scaling of implementation parallelism. As described in the next section, the degree of parallelism in the vector unit can be increased while maintaining object–code compatibility.

Lanes: Parallel Execution Units

Figure 5.11(a) shows the execution of a vector add instruction on a single pipelined adder. Results are computed at the rate of one element per cycle. Figure 5.11(b) shows the execution of a vector add instruction using four parallel pipelined adders. Elements are interleaved across the parallel pipelines allowing up to four element results to be computed per cycle. This increase in parallelism is invisible to software except for the increased performance.

Figure 5.12 shows how a typical vector unit can be constructed as a set of replicated lanes, where each lane is a cluster containing a portion of the vector register file and one pipeline from each vector

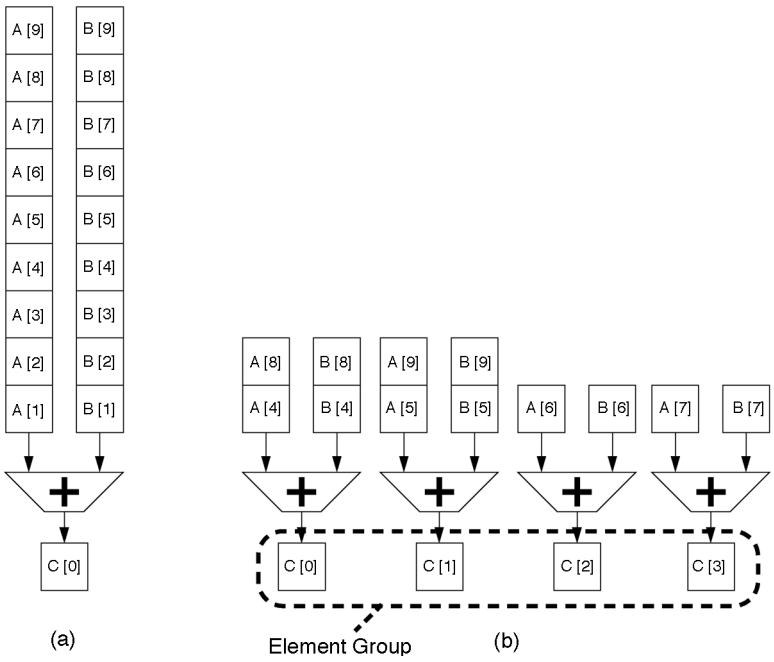

FIGURE 5.11 Execution of vector add instruction using different numbers of execution units. The machine in (a) has a single adder and completes one result per cycle, while the machine in (b) has four adders and completes four results every cycle. An element group is the set of elements that proceed down the parallel pipelines together. (From Asanovic, K., *Vector Microprocessors,* 1998. With permission.)

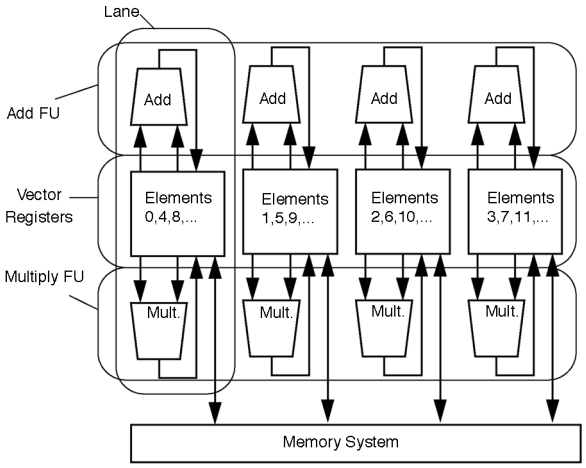

FIGURE 5.12 A vector unit constructed from replicated lanes. Each lane holds one adder and one multiplier as well as one portion of the vector register file and a connection to the memory system. The adder functional unit (adder FU) executes add instructions using all four adders in all four lanes.

functional unit. Because of the way the vector ISA is designed, there is no need for communication between the lanes except via the memory system. The vector registers are striped over the lanes, with lane 0 holding all elements 0, N, $2N$, etc., lane 1 holding elements 1, $N + 1$, $2N + 1$, etc. In this way, each elemental vector arithmetic operation will find its source and destination operands located within the same lane, which dramatically reduces interconnect costs.

The fastest current vector supercomputers, NEC SX-5 and Fujitsu VPP5000, employ 16 parallel 64-bit lanes in each CPU. The NEC SX-5 can complete 16 loads, 16 64-bit floating-point multiplies, and 16 floating-point adds each clock cycle.

Many data parallel systems, ranging from vector supercomputers, such as the early CDC STAR-100, to the MX ISAs, such as AltiVec, provide variable precision lanes, where a wide 64-bit lane can be subdivided into a larger number of lower precision lanes to give greater performance on reduced precision operands.

Vector Register File Organization

Vector machines differ widely in the organization of the vector register file. The important software-visible parameters for a vector register file are the number of vector registers, the number of elements in each vector register, and the width of each element. The Cray-1 had eight vector registers each holding sixty-four 64-bit elements (4096 bits total). The AltiVec MX for the PowerPC has 32 vector registers each holding 128-bits that can be divided into four 32-bit elements, eight 16-bit elements, or sixteen 8-bit elements. Some vector supercomputers have extremely large vector register files organized in a vector register hierarchy, e.g., the NEC SX-5 has 72 vector registers (8 foreground plus 64 background) that can each hold five hundred twelve 64-bit elements.

For a fixed vector register storage capacity (measured in elements), an architecture has to choose between few longer vector registers or more shorter vector registers. The primary advantage of lengthening a vector register is that it reduces the instruction bandwidth required to attain a given level of performance because a single instruction can specify a greater number of parallel operations. Increases in vector register length give rapidly diminishing returns, as amortized startup overheads become small and as fewer applications can take advantage of the increased vector register length.

The primary advantage of providing more vector registers is that it allows more temporary values to be held in registers, reducing data memory bandwidth requirements. For machines with only eight vector registers, vector register spills have been shown to consume up to 70% of all vector memory traffic, while increasing the number of vector registers to 32 removes most register spill traffic [9,10]. Adding more

vector registers also gives compilers more flexibility in scheduling vector instructions to boost vector instruction-level parallelism.

Some vector machines provide a configurable vector register file to allow software to dynamically choose the optimal configuration. For example, the Fujitsu VPP 5000 allows software to select vector register configurations ranging from 256 vector registers each holding 128 elements to eight vector registers holding 4096 elements each. For loops where few temporary values exist, longer vector registers can be used to reduce instruction bandwidth and stripmining overhead, while for loops where many temporary values exist, the number of shorter vector registers can be increased to reduce the number of vector register spills and, hence, the data memory bandwidth required. The main disadvantage of a configurable vector register file is the increase in control logic complexity and the increase in machine state to hold the configuration information.

Traditional Vector Computers versus Microprocessor Multimedia Extensions

Traditional vector supercomputers were developed to provide high performance on data parallel code developed in a compiled high level language (almost always a dialect of FORTRAN) while requiring only simple control units. Vector registers were designed with a large number of elements (64 for the Cray-1). This allowed a single vector instruction to occupy each deeply pipelined functional unit for many cycles. Even though only a single instruction could be issued per cycle, by starting separate vector instructions on different vector functional units, multiple vector instructions could overlap in execution at one time. In addition, adding more lanes allows each vector instruction to complete more elements per cycle.

MX ISAs for microprocessors evolved at a time where the base microprocessors were already issuing multiple scalar instructions per cycle. Another distinction is that the MX ISAs were not originally developed as compiler targets, but were intended to be used to write a few key library routines. This helps explain why MX ISAs, although sharing many attributes with earlier vector instructions, have evolved differently. The very short vectors in MX ISAs allow each instruction to only specify one or two cycle's worth of work for the functional units. To keep multiple functional units busy, the superscalar dispatch capability of the base scalar processor is used. To hide functional unit latencies, the multimedia code must be loop unrolled and software pipelined. In effect, the multimedia engine is being programmed in a microcoded style with the base scalar processor providing the microcode sequencer and each MX instruction representing one microcode primitive for the vector engine.

This approach of providing only primitive microcode level operations in the multimedia extensions also explains the lack of other facilities standard in a vector ISA. One example is vector length control. Rather than use long vectors and a VLR register, the MX ISAs provide short vector instructions that are placed in unrolled loops to operate on longer vectors. These unrolled loops can only be used with long vectors that are a multiple of the intrinsic vector length multiplied by the unrolling factor. Extra code is required to check for shorter vectors and to jump to separate code segments to handle short vectors and the remnants of any longer vector that were not handled by the unrolled loop. This overhead is greater than for the stripmining code in traditional vector ISAs, which simply set the VLR appropriately in the last iteration of the stripmined loop.

Vector loads and stores are another place where functionality has been moved into software for the MX ISAs. Most MX ISAs only provide unit-stride loads and stores that have to be aligned on boundaries corresponding to the vector length, not just aligned at element boundaries as in regular scalar code. For example, a unit-stride load of four 16-bit quantities has to be aligned at 64-bit boundaries in most MX instruction sets, although in some cases hardware will handle misaligned loads and stores at a slower rate. To help handle misaligned application vectors, various shift and align instructions have been added to MX ISAs to allow misalignment to be handled as part of the software microcoded loop. This approach simplifies the hardware design, but unfortunately these misaligned vectors are common in application code, and significant slowdown occurs when performing alignment in software. This encourages the

use of loops optimized for certain operand alignments, which leads to an increase in code size and also in loop startup time to select the appropriate routine. In certain cases, the application can constrain the layout of the data elements to ensure alignment at the necessary boundaries, but typically this is only possible when the entire application has been optimized for these MX instructions, for example, in a dedicated media player. Strided and indexed operations are also usually coded as scalar loads and stores with a corresponding slowdown over full vector mode.

Memory System Design

Perhaps the biggest difference between microprocessors and vector supercomputers is in the capabilities of the vector memory system. Vector supercomputers usually forgo data caches and rely on many banks of interleaved main memory to provide high memory bandwidth, while microprocessors rely on multilevel cache hierarchies to isolate the CPU from memory latencies and limited main memory bandwidth. A modern high-end vector supercomputer provides over 50 GB/s of main memory bandwidth per CPU, while high-end microprocessor systems provide only around 1 GB/s per CPU. For applications that require non-unit stride accesses to large data sets, the bandwidth discrepancy is even larger, because microprocessors access memory using long cache lines that waste bandwidth when there is little spatial locality. A modern vector CPU might sustain 16 or 32 non-unit stride memory operations every cycle pipelined out to main memory, with hundreds of outstanding memory accesses, while a microprocessor usually can only have a total of four to eight cache line misses outstanding at any time. This large difference in non-unit stride memory bandwidth is the main reason that vector supercomputers remain popular for certain applications, including car crash simulation and weather forecasting.

Traditional vector ISAs use long vector registers to help hide memory latency. MX ISAs have only very short vector registers and so require a different mechanism to hide memory latency and make better use of available main memory bandwidth. Various forms of hardware and software prefetching schemes have become popular with microprocessor designers to hide memory latency. Hardware prefetching schemes dynamically inspect memory references and attempt to predict which data will be needed next, fetching these into the cache before requested by the application. This approach has the advantage of not requiring changes to software, but can be inaccurate and can consume excessive memory bandwidth on misspeculated prefetches.

Software prefetching can be very accurate as the compiler knows the reference patterns of each piece of code, but the software prefetch instructions have to be carefully scheduled so that data are not brought in too early, perhaps evicting useful data, or too late, which will leave some memory latency exposed. The optimal schedule depends on the CPU and memory system implementations, which implies that code optimized for one generation of CPU or one particular memory system.

For either hardware or software prefetching schemes, it is essential that the memory controller can support many outstanding requests, otherwise high memory bandwidths cannot be sustained from a typical high latency memory system.

Future Directions

Microprocessor architects are continually searching for techniques that can take advantage of ever increasing transistor counts to improve application performance. Data parallel ISA extensions have proven effective on a wide range of applications, and hardware designs scale well to more parallel lanes. Existing supercomputers have sixteen 64-bit lanes while microprocessor MX implementations have expanded to two 64-bit lanes. It is likely that there will be further expansion of MX units to four or more 64-bit lanes. At higher lane counts, efficiencies drop, partly because of limited application vector lengths and partly because additional lanes do not help non-data parallel portions of each application.

An alternative approach to attaining high throughput on data parallel applications is to add more CPUs each with vector units and to parallelize loops at the thread level. This technique also allows

independent CPUs to run different tasks to improve system throughput. The main disadvantages of this multiprocessor approach compared to simply increasing the number of lanes are the hardware costs of additional scalar processor logic and the additional inter-CPU synchronization costs. The relative cost of adding more CPUs is reduced as lane counts grow, particularly when the cost of providing sufficient main memory bandwidth is considered. The inter-CPU synchronization cost is a more serious issue as it adds to vector startup latencies and can increase $N_{\frac{1}{2}}$ dramatically, reducing the effectiveness of multiprocessors on shorter vectors. For this reason, vector supercomputers have added fast inter-CPU synchronization through dedicated shared semaphore registers. The Cray SV1 design makes use of these registers to gang together four 2-lane processors in software to appear as a single 8-lane processor to the user. It should be expected that some form of fast inter-CPU synchronization primitive will be added to ISAs as design move to chip-scale multiprocessors, as these primitives can also be applied to many types of thread-level parallel code.

Increased CPU clock frequencies and increased lane counts combine to dramatically increase the memory bandwidth required by a vector CPU. The cost of a traditional vector style memory system will become prohibitive even for high-end vector supercomputers. Even if the cost could be justified, the high memory latency of a flat memory system will hamper performance for applications that have lower degrees of parallelism and that can fit in caches, and a continued move towards cached memory hierarchies for vector machines is to be expected leading to a merging of vector supercomputer and microprocessor design points.

MX extensions for microprocessors have undergone considerable changes since first introduced. The current designs provide low-level arithmetic and memory system primitives that are intended to be used in hand-microcoded loops. These result in high startup overheads and large code size relative to traditional vector extensions as discussed above. A possible future direction that could merge the benefit of vector ISAs and out-of-order superscalar microprocessors would be to add vector-style ISA extensions, but have these interpreted by microcode sequencers that would produce internal elemental microoperations that would be passed through the regular register renaming and out-of-order dispatch stages of a modern superscalar execution engine. This is similar to the way that legacy CISC string operations are handled by modern implementations.

Conclusions

Data parallel instructions have appeared in many forms in high-performance computer architectures over the last 30 years. They remain popular because many applications are amenable to data parallel execution, and because data parallel hardware is the simplest and cheapest way to exploit this type of application parallelism. As multimedia extensions evolve, they are likely to adopt more of the characteristics of traditional shared-memory vector ISAs to reduce loop startup overhead and decrease code size. However, these new multimedia vector ISAs will be shaped by the need to coexist with the speculative out-of-order execution engines used by the superscalar processors.

References

1. Flynn, M. J., Very high-speed computing systems, *Proc. IEEE,* 54, 1901, 1966.
2. Barnes, G. H., et al., The Illiac IV computer, *IEEE Trans. on Computers,* C-17, 46, 1968.
3. Reddaway, S., DAP–A Distributed Array Processor, in *Proc. 1st Annual Symp. Computer Architectures,* Florida, 1973, 61.
4. Batcher, K. E., Architecture of a massively parallel processor, in *Proc. Int. Symp. Computer Architecture,* 1980.
5. Hillis, W. D., *The Connection Machine,* MIT Press, Cambridge, Massachusetts, 1985.
6. Hintz, R. G. and Tate, D. P., Control data STAR-100 processor design, in *Proc. COMPCON,* 1972, 1.
7. Cragon, H. G. and Watson, W. J., A retrospective analysis: the TI advanced scientific computer, *IEEE Computer,* 22, 55, 1989.
8. Russel, R. M., The CRAY-1 computer system, *Communications of the ACM,* 21, 63, 1978.

9. Espasa, R., *Advanced Vector Architectures,* Ph.D. Thesis, Universitat Politecnica de Catalunya, Barcelona, 1997.
10. Asanovic, K., *Vector Microprocessors,* Ph.D. Thesis, University of California, Berkeley, 1998.

5.4 Multithreading, Multiprocessing

Manoj Franklin

Introduction

A defining challenge for research in computer science and engineering has been the ongoing quest for faster execution of programs. There is broad consensus that barring the use of novel technologies such as quantum computing and biological computing, the key to further progress in this quest is to do parallel processing of some kind.

The commodity microprocessor industry has been traditionally looking to fine-grained or instruction level parallelism (ILP) for improving performance, with sophisticated microarchitectural techniques (such as pipelining, branch prediction, out-of-order execution, and superscalar execution) and sophisticated compiler optimizations. Such hardware-centered techniques appear to have scalability problems in the sub-micron technology era and are already appearing to run out of steam. According to a recent position paper by Dally and Lacy [4], "Over the past 20 years, the increased density of VLSI chips was applied to close the gap between microprocessors and high-end CPUs. Today this gap is fully closed and adding devices to uniprocessors is well beyond the point of diminishing returns." We view ILP as the main success story form of parallelism thus far, as it was adopted in a big way in the commercial world for reducing the completion time of general purpose applications. The future promises to expand the "parallelism bridgehead" established by ILP with the "ground forces" of thread-level parallelism (TLP), by using multiple processing elements to exploit both fine-grained and coarse-grained parallelism in a natural way.

Current hardware trends are playing a driving role in the development of multiprocessing techniques. Two important hardware trends in this regard are single chip transistor count and clock speed, both of which have been steadily increasing due to advances in sub-micron technology. The Semiconductor Industry Association (SIA) has predicted that by 2012, industry will be manufacturing processors containing 1.4 billion transistors and running at 10 GHz [39]. DRAMs will grow to 4 Gbits in 2003. This increasing transistor budget has opened up new opportunities and challenges for the development of on-chip multiprocessing.

One of the challenges introduced by sub-micron technology is that wire delays become more important than gate delays [39]. This effect is predominant in global wires because their length depends on the die size, which is steadily increasing. An important implication of the physical limits of wire scaling is that, the area that is reachable in a single clock cycle of future processors will be confined to a small portion of the die [39].

A natural way to make use of the additional transistor budget and to deal with the wire delay problem is to use the concept of *multithreading* or *multiprocessing*[1] in the processor microarchitecture. That is, build the processor as a collection of independent *processing elements (PEs)*, each of which executes a separate *thread* or flow of control. By designing the processor as a collection of PEs, (a) the number of global wires reduces, and (b) very little communication occurs through global wires. Thus, much of the communication occurring in the multi-PE processor is *local* in nature and occurs through short wires.

In the recent past, several multithreading proposals have appeared in the literature. A few commercial processors have already started implementing some of these multithreading concepts in a single chip [24,34]. Although the underlying theme behind the different proposals is quite similar, the exact manner in which they perform multithreading is quite different. Each of the methodologies has different hardware

[1] In this section, we use the terms *multithreading, multiprocessing*, and *parallel processing* interchangeably. Similarly, we use the generic term *threads* whenever the context is applicable to processes, light-weight processes, and light-weight threads.