Spatially Aware Decentralized Computing

Andrew "bunnie" Huang, Ben Vandiver, Jeremy Brown, J.P. Grossman and Tom Knight

The Q-Machine is a spatially aware, decentralized massively parallel computer system that achieves latency reduction (as opposed to latency hiding) and enhanced fault tolerance through a virtual machine interface. Without obscuring the high-level architecture, details of the machine are hidden by the virtual machine interface, so that objects and threads can be efficiently migrated to reduce latency and to avoid congested or failing nodes. It also features enhanced synchronization decoupling, software pipelining and congestion tolerance through architecturally visible queues. The architecture is structured so that it is easy for compilers to perform standard optimizations and is easy for cleanly implemented object-oriented languages such as Java to target. The Q-Machine also features a hybrid fat-tree/multibutterfly network topology designed for wave-propagation limited performance and good fault tolerance. Finally, the architecture is aimed at implementation on cost-effective, readily available CMOS foundry processes with a minimum amount of risky full-custom design.

Introduction

The idea of having a central processing unit (CPU) has run out of gas. Wiring delays have become the dominant component of a processor's cycle time in deep sub-micron technology. As a result, the partitioning of computers into separate memory and processor components has become the architectural bottleneck. Also, the complex memory hierarchy required to hide memory access latency hinders fast multiprocessor synchronization, limiting system scalability. Figure 1 illustrates the evolution of L1 cache delay versus feature size scaling. Several years ago, single-cycle 128 kB L1-caches were common; now, three-cycle 32 kB L1-caches are the norm. Clearly, the ability to hide latency is diminishing and it is no longer acceptable to abstract away the spatial organization of a computer.

Another important issue is that large systems are impossible to manufacture perfectly and maintain at full operational capacity. The problem extends from the manufacturing yields and reliability of individual chips to the integration of cabinets and cables, and it must be addressed as an integral part of any massively parallel architecture. This problem is confounded by the fact that the average service life of a massively parallel machine is fairly long, and maintaining a base of exact replacement parts becomes difficult as manufacturing processes evolve. The Q-Machine addresses both of these issues with a single mechanism, the virtual machine interface (VMI). The VMI abstracts away the physical details of the machine's implementation without obscuring the overall organization of the machine from the programmer. This abstraction allows the operating system to migrate data out of failing nodes so that they can be shutdown and replaced, and it also allows the failed node to be replaced with a node implemented using the technology *du jour*.

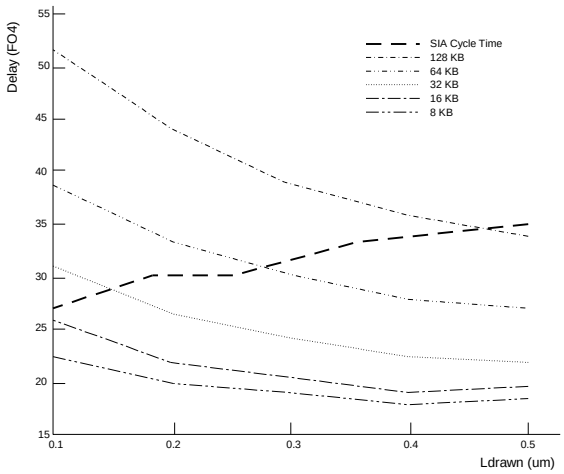

Figure 1: Cache delay versus feature size, overlayed with cycle time according to the SIA roadmap.[1]

Architecture

The virtual machine interface programming model is summarized in Figure 2. Instead of a register file, programmers are given a large (roughly 2^{40}) set of thread contexts, each with 128 explicit queues. The tails of these queues can be mapped to other contexts and memory; thus, these queues form the basic synchronization primitive of the machine. By providing a synchronization namespace separate from the memory namespace, machine implementation is simplified and single-threaded code performance can be kept high by offloading the burden of synchronization to a co-

processor. The ISA of the Q-Machine is similar to that of a standard RISC microprocessor, so most standard compilation techniques can be applied.

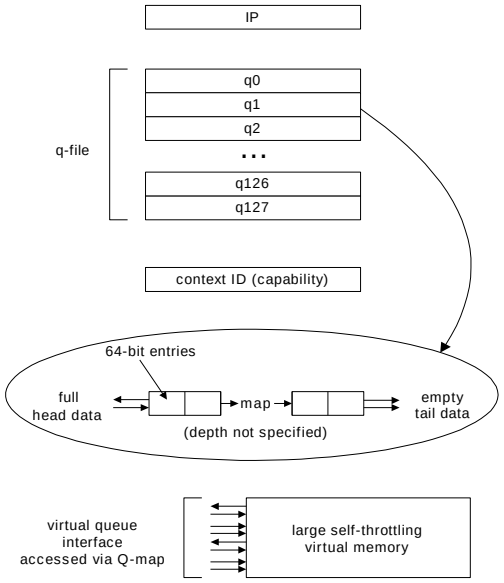

Figure 2: Virtual machine interface summary.

The context identifier for each thread is synonymous with its heap allocation pointer; an efficient hardware-assisted capability-based addressing mechanism allows the backing storage for spilled queues and variable storage to share this pointer. This combination of computation state and synchronization namespace reduces the task of object migration to one of garbage collection. The garbage collector considers the physical layout of the machine, remote communication statistics and processor node utilization, and data is incrementally redistributed about the machine to reduce access latencies and balance the computational load across all nodes. In addition, the use of queues to access memory allows for greater latency tolerance and the transparent implementation of smart memories.

The ability to map queues across threads also allows the compiler or programmer to build efficient streaming computation pipelines and hide access latencies when performing computations with regular data access patterns. Finally, many standard compiler optimizations are easy to perform on the Q-Machine: software pipelines reduce to a chain of threads, procedures can be effectively curried, and load hoisting is simple to implement, to name a few. The current implementation scheme calls for the Q-Machine to be programmed using a Java-like language with extensions and libraries that allow, but do not require, the programmer to explicitly manage queues.

The VMI intentionally leaves the number of implemented processors unspecified. While any Q-Machine program could be run on a single node, the architecture is geared to exploit the parallelism available in a multi-thousand node machine. It is also geared to exploit additional processors that become available as a result of "hot" machine upgrades.

Implementation

The basic organization of a single Q-Machine node is illustrated in Figure 3.

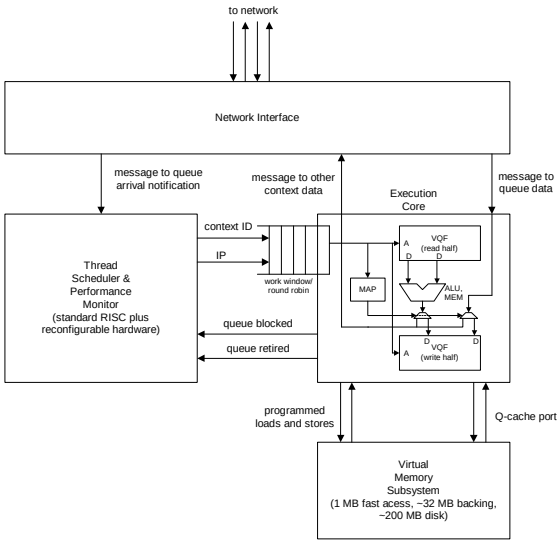

Figure 3: Single Q-Machine node overview.

A Q-Machine node utilizes a thread scheduler and performance monitor to handle synchronization events. It is implemented using a standard RISC core with reconfigurable hardware enhancements, such as those available through Tensilica. These cores can achieve 320 MHz performance in as little as 0.7 mm^2 in a 0.18µ process.[2] Thread scheduling is performed in a manner similar to the H- and V- thread scheme employed on the MIT M-Machine.[3]

The execution core executes threads out of the work queue prepared by the thread scheduler. Because the execution core does not have to worry about synchronization or network issues, it has an

organization similar to a standard RISC processor and it runs very fast. The queue file—referred to as the Virtual Queue File (VQF)—is implemented using a caching scheme similar to the Named State Register File (NSRF).[4] The VQF implementation is sized so that when running single-threaded code, queue thrashing is eliminated and high performance is maintained. It also turns out that having shallow (4- to 8-deep) queues in place of registers has a small area impact. This is due to the fact that register files today are primarily wire-dominated, and populating the unused silicon area underneath the wires yields queues "for free".

The memory subsystem of a single Q-Machine node is similar in many ways to a conventional memory hierarchy, with the exception of the data cache. The execution core is expected to be implemented using primarily behavioral HDL code; thus, its performance will be on par with the 1-T SRAM cores offered by MoSys. These innovative memory cores are fabricated on a TSMC logic processes and offer 450 MHz performance with 2-3 cycle random access latencies and densities comparable to those of some vendor's embedded DRAM processes at the 0.13µ node.[5] Since the execution core's performance will closely match the memory's performance, no data cache is required. However, only a small (~8 Mb) amount of memory can be integrated next to the processor at these performance levels; hence, a virtual memory hierarchy with off-chip DRAM and shared hard drive storage is implemented to provide a gradual performance roll-off in the case that a processor node fills up faster than the garbage collector can migrate data out of the node.

The network that connects the Q-Machine nodes is a hybrid multipath fat-tree / multipath multi-butterfly topology with path expansion and maximal fanout. An integral property of this topology is that no single component failure will leave any node unreachable. The network protocol is source-responsible, so routers are simple and fast. The propagation of data around the Q-Machine is essentially limited by the speed of electromagnetic wave propagation as a router hop incurs only 1-2 cycles of latency. Having a source-responsible network protocol also makes network failures and network congestion appear identical to the sending node; this simplifies network interface implementation without sacrificing fault tolerance. All clocks are synchronous to a single frequency source (with redundant backup) on the Q-Machine, and skew is tolerated by the use of a mesochronous timing scheme between nodes. The network implementation of the Q-Machine is based heavily on the METRO[6] network.

Conclusion

The Q-Machine is a spatially aware decentralized computer architecture that is well-suited to the process technology scenario that designers will be facing even after Moore's law runs out of steam. Its virtual machine interface abstracts details of the machine implementation away from the programmer in a manner that enables the efficient migration of data to reduce latency. This abstraction also gives the Q-Machine good scalability, fault tolerance, and serviceability across semiconductor process generations. The virtual machine abstraction is also capable of readily extracting parallelism out of languages similar to conventional object-oriented languages such as Java.

References

[1] McFarland, Grant W. *CMOS Technology Scaling and Its Impact on Cache Delay*. PhD Dissertation submitted to Stanford University, June 1997.

[2] *Xtensa Application Specific Microprocessor Solutions: Overview Handbook, a Summary of the Xtensa Data Sheet*. Tensilica, Inc. Issue date: 2/2000.

[3] Fillo, M. Keckler, S. W. Dally, W. J. Carter, N. P. Chang, A. Gurevich, Y. Lee, W. S. "The M-Machine Multicomputer." *Proceedings of the 28th Annual International Symposium on Microarchitecture*. IEEE, 1995. Pp. 146-156.

[4] Nuth, P. R. Dally, W. J. "The Named-State Register File: Implementation and Performance." *Proceeding of the First IEEE Symposium on High-Performance Computer Architecture*. 1995. Pp. 4-13.

[5] *TSMC 0.13u Process Fast 1-T SRAM Summary*. Available off the Mosys web page at www.mosys.com. Registration required to access design materials.

[6] DeHon, André. *Robust, High-Speed Network Design for Large-Scale Multiprocessing*. AI Technical Report Number 1445. September 1993. MIT Artificial Intelligence Laboratory.