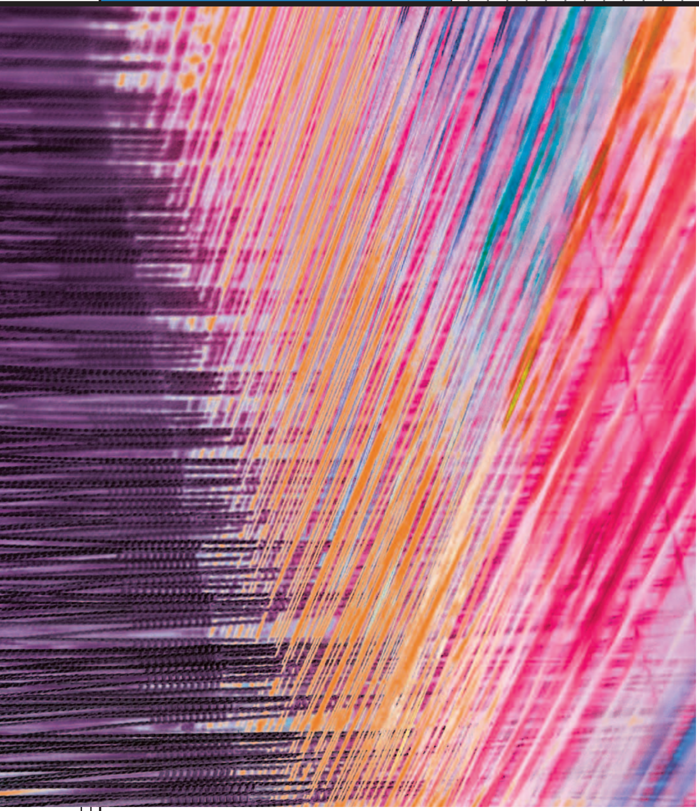

Designing Next-Generation Massively Multithreaded Architectures for Irregular Applications

Antonino Tumeo, Simone Secchi, and Oreste Villa, *Pacific Northwest National Laboratory*

Massively multithreaded architectures like the Cray XMT address the needs of irregular data-intensive applications better than commodity clusters. A proposed evolution of the XMT integrates multicore processors and next-generation interconnects, along with memory reference aggregation to optimize network utilization.

Current high-performance computing systems are designed to efficiently execute floating-point-intensive workloads.[1] HPC systems are mainly built for scientific simulations, which are characterized by high computational density and locality, and regular, partitionable data structures. These application requirements are driving processor designs toward fast SIMD (single instruction, multiple data) arithmetic units, and deep cache hierarchies to reduce access latencies.

At the system level, memory and interconnection bandwidths are increasing at much slower rates than peak computational performance, but regularity and locality lessen the impact of this problem. At the same time, emerging processor architectures are pushing application development toward implementations that can exploit their features.

However, applications from emerging fields such as bioinformatics, community detection, complex networks, semantic databases, knowledge discovery, natural language processing, pattern recognition, and social network analysis have an irregular nature. They typically use pointer-based data structures such as unbalanced trees, unstructured grids, and graphs that are massively parallel but have poor spatial and temporal locality. Efficiently partitioning these data structures is challenging. Furthermore, data structures often change dynamically during the application execution—for example, adding and removing connections in graphs.

Complex cache hierarchies are ineffective with such irregular applications. The off-chip bandwidth that is available for the system memory to access local data and for the network to access data on other nodes mainly determines the system's performance. Under these conditions, a single control thread usually does not offer enough concurrency to utilize all of the available bandwidth. Therefore, multithreaded architectures usually try to tolerate rather than reduce memory access latencies by switching among multiple threads, continuously generating memory references and maximizing bandwidth utilization.

RESEARCH MOTIVATION

The Cray XMT is a multinode supercomputer specifically designed for developing and executing irregular applications.[2] Its architecture is based on three “pillars”: a global address space, fine-grained synchronization, and multithreading.

The XMT is a distributed shared memory (DSM) system in which the globally shared address space is uniformly scrambled at very fine granularity on the different node memories. Each node integrates a ThreadStorm custom processor that switches, on a cycle-by-cycle basis, among numerous hardware threads. This allows toleration of both system latency for accessing the memory local to a node and network latency for accessing memory in remote nodes.

In contrast to the latest HPC systems, the XMT provides a system-wide programming model, which eases the implementation of applications with large memory footprints without requiring optimization for locality. Even if modern HPC systems integrate multithreaded architectures such as graphics processing units (GPUs), they appear better suited to regular applications. To date, they are not designed to tolerate latencies for accessing memories on different nodes. In many cases, they cannot even tolerate latencies for accessing memories of other processors in the same node. Furthermore, coordinating memory accesses and maximizing bandwidth utilization requires data partitioning and significant programming efforts.

Cache-based processors normally experience a large number of cache misses in irregular applications due to unpredictable memory accesses.

Pacific Northwest National Laboratory’s CASS-MT project (http://cass-mt.pnnl.gov) is currently exploring massively multithreaded architectures for irregular applications. Here, we present a classification of multithreaded architectures and discuss them in relation to the Cray XMT. We then propose ways to evolve these architectures and evaluate a possible future XMT design integrating multiple cores per node and a next-generation network interconnect. Finally, we show how integration of a hardware mechanism for remote reference aggregation can optimize network utilization.

MULTITHREADED ARCHITECTURES

A multithreaded processor concurrently executes instructions from different threads of control within a single pipeline. There are two basic types of multithreaded processors: those that issue instructions only from a single thread in a cycle, and those that issue instructions from multiple threads in the same cycle.

Many advanced out-of-order superscalar processors such as the IBM Power6 and Power7 or the latest Intel architectures, Nehalem and Sandy Bridge, support the simultaneous multithreading (SMT) technique. SMT keeps multiple threads active in each core—the processor identifies independent instructions and simultaneously issues them to the core’s various execution units, thereby maintaining high utilization of the processor resources.

Multithreaded processors that issue instructions from a single thread every clock cycle, known as *temporal multithreaded processors*, alternate between different threads to keep the (usually in-order) pipeline filled and avoid stalls. Temporal multithreading can be coarse-grained (block multithreading) or fine-grained (instruction/cycle interleaved).

Block multithreading switches from one thread to another only when an instruction generates a long latency stall, such as a cache miss that requires access to the off-chip memory. Intel Montecito uses block multithreading.

Interleaved multithreading switches from one thread to another on a cycle-by-cycle basis. The ThreadStorm processors in the Cray XMT, as well as their predecessors in the Tera MTA and Cray MTA-2, use interleaved multithreading. The SPARC cores in the Sun UltraSPARC T1 and T2 and SPARC T3 also employ a form of interleaved multithreading.

The UltraSPARC T1 consists of eight cores, each with four threads. The T2 also has eight cores but doubles the execution units and the number of threads, enabling it to co-issue instructions from two threads out of a pool of eight in each clock cycle. The SPARC T3 doubles the number of cores with respect to the UltraSPARC T2. These cores issue an instruction from a different thread every cycle. When a long latency event occurs, however, they take the thread that generated it out of the scheduling list until that event completes.[3]

GPUs also integrate multiple thread scheduling blocks (warps or wavefronts) that can be efficiently switched on the SIMD execution units to tolerate long latency memory operations.[4] GPUs feature hundreds of floating-point units and massive memory bandwidths, and have on-chip memories to optimize accesses to frequently used data. However, GPUs are currently designed as accelerators with their own private memory and are more amenable to regular workloads.

Cache-based processors normally experience a large number of cache misses in irregular applications due to unpredictable memory accesses. Temporal multithreaded architectures are generally better suited to these applications because they can tolerate long latency memory accesses by switching to other ready threads while the memory subsystem loads or writes the data, thereby not necessarily requiring caches to reduce access latencies.

CRAY XMT ARCHITECTURE

The Cray XMT consists of dual-socket Opteron AMD service nodes and custom-designed multithreaded compute nodes with one ThreadStorm processor per node. The system can scale to 8,192 compute nodes with 128 terabytes of shared memory. However, the largest system built to date has 512 compute nodes.

Each ThreadStorm is a 64-bit VLIW (very long instruction word) processor containing a memory unit, an arithmetic unit, and a control unit. It switches on a cycle-by-cycle basis among 128 fine-grained hardware streams to tolerate the stalls that memory accesses generate.

At runtime, the system maps a software thread to a hardware stream, which includes a program counter, a status word, a set of target and exception registers, and 32 general-purpose registers. The pipeline is 21 stages long for all instructions. By design, the processor does not issue a new instruction from the same stream until the previous instruction has exited the pipeline.

Because memory operations take longer than 21 cycles, the processor supports look-ahead, allowing independent instructions of the same stream to issue every 21 cycles. The compiler identifies the look-ahead for every instruction, which can be up to eight instructions. Each hardware stream can thus have up to eight pending memory operations at the same time, resulting in a maximum of 1,024 memory operations pending for the entire processor. Memory operations can complete out of order.

The ThreadStorm has a 64-Kbyte, four-way associative instruction cache for exploiting code locality and runs at a nominal 500-MHz frequency.

The processor is linked to the interconnection network through a point-to-point HyperTransport channel. The network interface controller (NIC) does not perform any aggregation, and a single network packet encapsulates each memory operation. The network subsytem is based on the Cray SeaStar2 interconnect,[5] and the topology is a 3D torus.

Each ThreadStorm's memory controller (MC) manages up to 8 Gbytes of 128-bit wide DDR (double data rate) RAM and has a 128-Kbyte, four-way associative access cache that helps reduce access latencies. The application accesses the system's memory via a shared memory abstraction: the system can direct load and store operations to any physical memory location from any ThreadStorm processor connected to the network.

The system scrambles memory with 64-byte granularity to allocate logically sequential data to physical memories attached to different processors. Data is distributed, almost uniformly, among the memories on the full system's nodes, regardless of the number of processors the application uses.

Associated with each 64-bit memory word are

- a *full-empty bit* that works as a lock;
- a *pointer-forwarding bit* that signals memory locations containing pointers rather than data, allowing automatic generation of a new memory reference; and
- two *trap bits* that can generate a signal when the location is stored or loaded.

The ThreadStorm generates up to 500 million memory references per second (Mref/s), that is, one reference per clock cycle. However, the MC can sustain up to 100 Mref/s to the DDR RAM, while the SeaStar2 NIC reaches 140 Mref/sec. The latency for memory operations ranges from ~68 cycles (hit in the local memory controller's cache) to ~1,200 cycles (miss in the farthest remote memory controller's cache) for a 128-node system.

The processor architecture is unchanged in the recently introduced Cray XMT 2. The only significant modification is higher local memory bandwidth, obtained using DDR2 memory and an additional memory channel, a feature that imposes higher latencies (approximately double) for local memory accesses.

Compared to other multithreaded architectures, the XMT is more flexible for developing and executing irregular applications with large memory footprints.

COMPARISON WITH OTHER ARCHITECTURES

With respect to other multithreaded architectures such as GPUs and UltraSPARC processors, the XMT is more flexible for developing and executing irregular applications with large memory footprints, typical in data analytics.

GPUs and UltraSPARC processors

In addition to running tens of thousands of threads, GPUs have massive amounts of memory bandwidth. However, to maximize memory utilization, applications should access memory with regular patterns. To allow the MC to coalesce many memory operations in a single large memory transaction, threads running on the same SIMD units should, in fact, access sequential memory locations or, at least, locations residing in the same memory segment. If memory operations access different memory segments, they require multiple transactions (in the worst case, one for each operation), wasting memory bandwidth. In other words, address bandwidth is significantly lower than data bandwidth.

In some cases, it is possible to exploit GPUs' texture units—memory units that load color data for pixels during graphic rendering—to load data for general-purpose computation. This approach can reduce the penalties associated with unaligned memory operations. However, due to its graphics origin, texture memory is mostly effective when data has good spatial locality and the application can better exploit connected texture caches.

GPUs also include fast on-chip scratchpad memories, which can reduce access latencies to small chunks of data with high locality. Developers of irregular applications can sometimes use the scratchpads to implement programming techniques for accumulating data and making off-chip memory accesses more regular. However, when applicable, this approach requires significant effort.

The latest-generation GPUs also include data caches, but if applications have very irregular access patterns they might not be effective due to the high number of misses. When the applications' irregularity is in the control flow and threads running on the same SIMD unit of the GPU diverge, there is a performance penalty.

The XMT's shared memory abstraction, flat memory hierarchy, and fine-grained synchronization provide a simple whole-machine programming model.

For irregular applications with big datasets, scalability on multiple GPUs still represents a challenge. Currently, the largest memory size for a GPU board is 6 Gbytes. Since there is no shared memory abstraction among different GPUs, application developers must manually partition data, and communication across the PCI Express bus could be overkill for algorithms that maintain a shared state.

Even with peer-to-peer and direct access available in the latest GPUs, it is still necessary to partition data across GPUs. At the cluster level, when it becomes necessary to also cross the network, partitioning becomes even more complex, and communication overheads increase.

UltraSPARC processors present similar challenges in scalability over nodes. Inside a node, they expose a shared memory abstraction. Across nodes, however, they still exhibit the usual distributed memory setup and require distributed memory programming approaches. UltraSPARC processors are designed to hide latency only at the node level, and do not make any effort to tolerate cluster-wide network latencies.

XMT advantages

The XMT is specifically designed to scale across nodes. Its shared memory abstraction, flat memory hierarchy, and fine-grained synchronization provide a simple whole-machine programming model. A program with a large memory footprint can use all of the XMT's available memory without requiring the programmer to restructure its code.

The ThreadStorm processor handles local and remote (to other nodes) memory operations at single-word granularity. Memory scrambling aims to distribute memory and network access patterns, reducing hot spots. Because of its large number of threads with respect to its limited resources, the ThreadStorm can tolerate interconnection network latency. These features reduce the usable data bandwidth but also remove penalties in terms of performance and programming effort when accessing memory with highly irregular patterns.

In general, in addition to a simpler machine-wide programming model, the XMT is more efficient than other architectures when data does not fit in the memory of a single node—or of a single board in the case of GPUs—and when the highly irregular access patterns make caches or coordinated memory accesses ineffective. For example, a programmer can implement a simple breadth-first search (BFS) algorithm without concern about data movement and memory pattern optimization, and obtain good performance and scalability.

Another example is Aho-Corasick string matching with very large dictionaries.[6] On cache-based processors, if patterns are in the cache, the matching procedure is fast. If they are not, the procedure slows down because the processor needs to retrieve the pattern from memory. Accesses to the pattern structure are unpredictable, thus coalescing them is difficult.

When the algorithm matches a dictionary with itself (worst case), it thoroughly explores the pattern data, and architectures optimized for regular accesses experience performance degradations. On the other hand, when the algorithm matches only a few patterns and can exploit caches (best case), the XMT does not necessarily reach the same peak performances. However, the system's flat memory hierarchy ensures performance stability, as best-case performances are not much different from the worst-case ones.

SIMULATING THE XMT

To support our research on massively multithreaded systems, we developed a novel parallel full-system simulator that can replicate a 128-processor XMT on a 48-core host (quad-socket Opteron 6176SE) with speeds up to 250 kilocycles per second and an accuracy error rate under 10 percent for a large set of typical irregular applications.[7] The simulator runs unmodified XMT applications, supporting all the system's architectural features, and integrates a parametric model that quickly but accurately describes network and memory communication events, including contention effects.[8] We used this simulator to identify bottlenecks in the current XMT architecture and to propose a possible multicore-based evolution that overcomes these limits.

We initially estimated the impact of memory operations on the overall execution time of a 128-processor XMT by instrumenting the simulator with counters updated every time the pipeline stalls due to pending memory requests. Figure 1a shows the simulated execution times of the Aho-Corasick string-matching algorithm[6] on a dictionary comprising the most common 20,000 English words and an input set of random symbols with a uniform distribution from the ASCII alphabet and an average length of 8.5 bytes.

Even when the system does not use all of the processors for memory operations, it still scrambles the address space on all the nodes. As Figure 1b shows, the percentage of time the system spends waiting for memory access remains stable at around 70 percent until 16 processors become active. When more than 16 processors are active, however, the overall execution time stops decreasing and the time the system spends waiting for memory access increases, exceeding 90 percent for 128 active processors.

To identify the cause of this behavior, we used the simulator to monitor memory reference traffic on the NICs and MCs. Figure 2 shows the incoming traffic to one of the network's NICs when all 128 processors are active. The solid line represents the number of incoming references that are requesting access to the NIC, while the dashed line represents the maximum rate of references per second that the NIC can serve (140 Mref/s bidirectional). The figure shows that in many application segments the requested injection rate far exceeds the available NIC bandwidth, stalling remote memory references and delaying overall execution.

Regarding the MC interfaces, we discovered that bandwidth saturation effects are less severe than those on the NIC. Although this might appear to contrast with the sustained reference rates, it is dictated by the uniform address scrambling. For example, on a 128-node configuration, due to the scrambling, a processor generates on average one local reference and 127 remote references out of 128 possible destination nodes. Only the local reference will use the MC channel, while the remote ones will use the NIC channel. However, incoming references from other nodes flow through both the NIC and MC channels. Combining incoming and outgoing references, the NIC channel is subject to higher traffic than the MC channel.

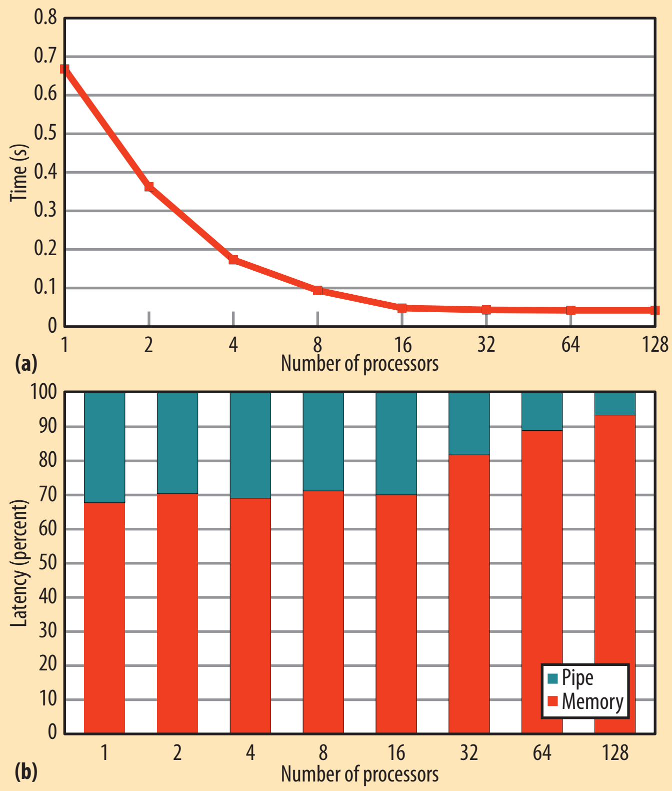

Figure 1. Impact of memory operations on overall execution times of the Aho-Corasick string-matching algorithm versus number of active processors, on a simulated 128-node Cray XMT: (a) overall application execution times and (b) percentage of time spent on memory operations. The globally shared address space is always uniformly scrambled over 128 physically distributed memories.

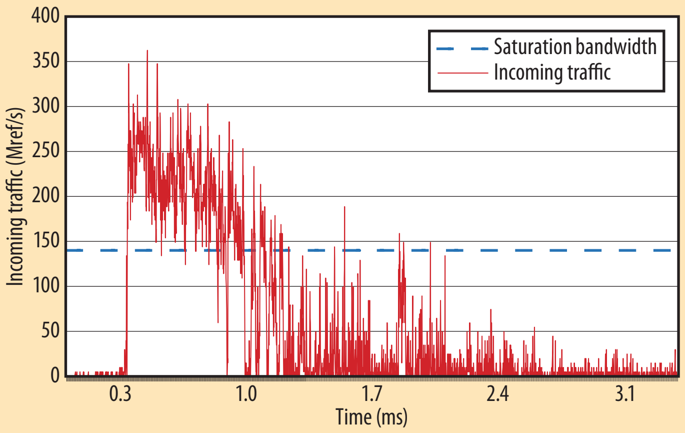

Figure 2. Bandwidth saturation patterns of one network interface channel (NIC) for the Aho-Corasick string-matching algorithm on a 128-node XMT-like system, when all processors are active. The solid line represents the number of incoming references requesting access to the NIC, while the dashed line represents the maximum rate of references per second that the NIC can serve (140 Mref/s bidirectional). In many application segments, the requested injection rate far exceeds the available NIC bandwidth, stalling remote memory references and delaying overall execution.

MULTICORE DESIGN EVOLUTION

Based on these results, we designed a possible XMT-based multicore system architecture. We used the simulator to determine the best tradeoff in number of cores and MCs that improves overall system performance.

System design

As Figure 3 shows, the design includes multiple ThreadStorm-like cores in which each hardware thread has a dedicated register file and a load/store queue (LSQ). The LSQs all interface to a scrambling module, which uniformly distributes memory operations across all of the system's memory modules. Every processor also includes several independent MCs, each one interfaced to a different off-chip memory module. All of the components are connected through a packet-switched network on chip (NoC), configured as a 2D mesh, and each has its own dedicated router. An integrated NIC connects the processor to the other nodes in the system. The NoC's latency accounts for less than 2 percent of the system-wide average memory reference latency.

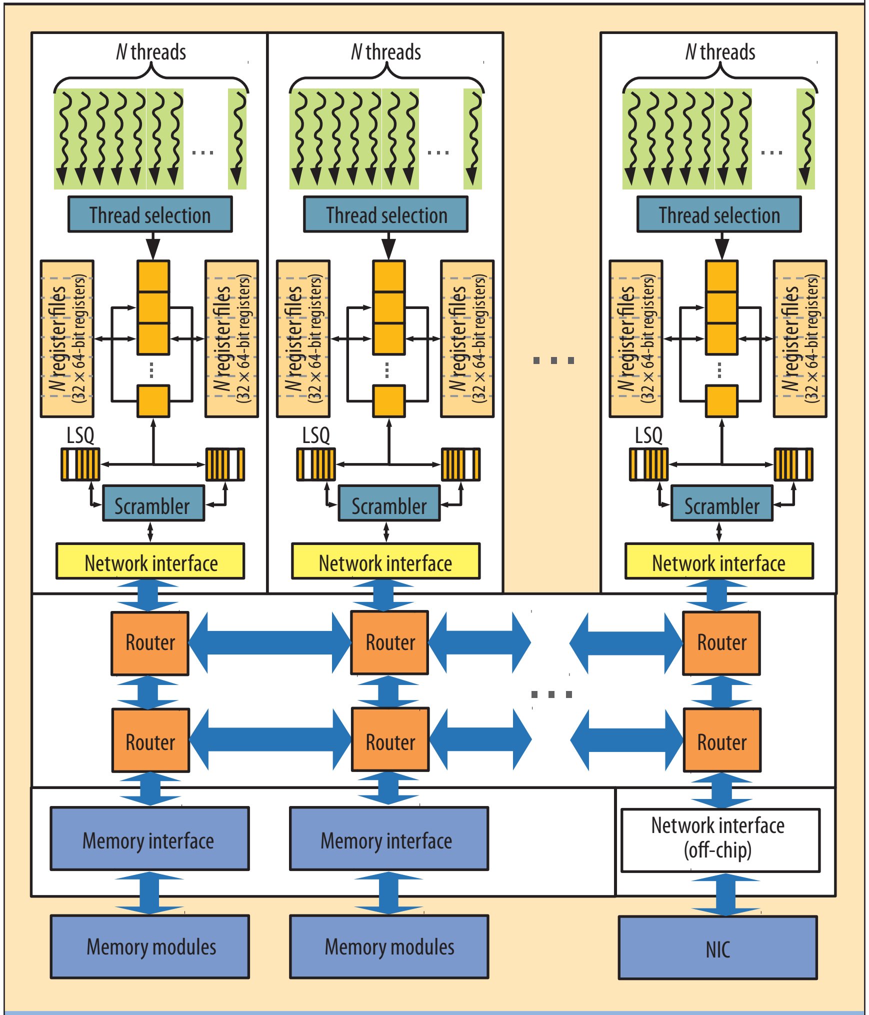

Figure 3. XMT-based multicore processor architecture.

Performance evaluation

We used the simulator to evaluate this design in an XMT-like system comprising 32 nodes when increasing the number of cores per processor from 1 to 16 and the number of independent MCs per processor from 1 to 8. We set the bandwidth for the NIC at 670 Mref/s (five times that of the current XMT SeaStar2 network[5] and similar to the Cray Gemini interconnect) and the bandwidth for each MC at 200 Mref/s (twice that of the current XMT DDR memory channels). Both latency and bandwidth parameters used in the simulations aligned with network road maps[1] and DDR3 memory specifications.

Figure 4 shows the results of the evaluation using three benchmarks: the same Aho-Corasick string-matching application we used in previous experiments; BFS, a common kernel in applications that use graph-based data structures; and matrix multiplication, a typical regular kernel that we included to assess the impact of our design decisions on more common and less-constrained applications. The solid "ideal" line in the figures represents the performance obtained when network and memory interfaces are contention free and saturation never occurs. These lines demonstrate that the benchmarks have enough parallelism to scale as the number of cores per processor increases.

As Figure 4a shows, in string matching, the number of MCs per processor does not substantially affect overall performance. This is a consequence of the memory organization and of application optimizations to reduce hot-spotting. The speedup is almost linear up to four cores per processor, and thereafter reaches a plateau due to NIC saturation.

The results are quite different for BFS, as Figure 4b shows. BFS is a synchronization-intensive benchmark (it uses a lock for every vertex visited), and the NICs are already saturated by retry operations in the basic configuration. Thus, the application does not scale.

In matrix multiplication, as Figure 4c shows, using more MCs (up to two) improves performance because it reduces hot-spotting on some of the memory interfaces. With more than two cores and two MCs per processor, the bottleneck again becomes the NICs, and application performance stabilizes regardless of the number of cores or MCs.

These experiments suggest that a multicore approach would allow exploiting next-generation interconnects for irregular applications. In fact, with a network that can sustain a bandwidth (with small messages) almost five times higher than the current SeaStar2, such a system would increase performance up to four cores per node, due to the higher injection rate. Regular applications also appear to benefit from the higher local memory bandwidth provided by the multiple MCs per processor.

MEMORY REFERENCE AGGREGATION

In the XMT and our multicore design, a small network packet encapsulates each remote memory operation, generating fine-grained traffic. Small messages generally incur high overhead due to the reduced header/payload ratio in all the networks not optimized for such traffic.

To mitigate this effect, some interconnect systems aggregate small packets. They usually do this either in hardware with custom router designs, as in the IBM Power7 hub module,[9] or in software through specific drivers and the API. However, in XMT-like machines where every memory load/store is directly mapped onto a network reference without any interposed software layer, the latter approach is difficult to realize.

To minimize the network overhead for small messages without designing an expensive customized network, we integrated a modification to our multicore design that implements network reference aggregation at the processor level.

Hardware implementation

The aggregation mechanism operates by buffering up to *D* memory references for the same destination node (DST) in a time window of *W* cycles. When *D* memory references are reached, or the time window has elapsed, the mechanism sends all the memory references in the buffer toward the DST using a single network packet, sharing the header.

Figure 5 shows the hardware design implementing the proposed mechanism. The LSQs of the various cores interface with a selection logic that identifies the DST for each memory reference. The selection logic uses the DST to address one of the FIFO (first-in, first-out) buffers that store the memory reference's data—address, value, memory operation codes, and control bits—for the aggregation.

There is a FIFO buffer for each DST in the system, and all buffers have a size of *D* references. When *D* references are

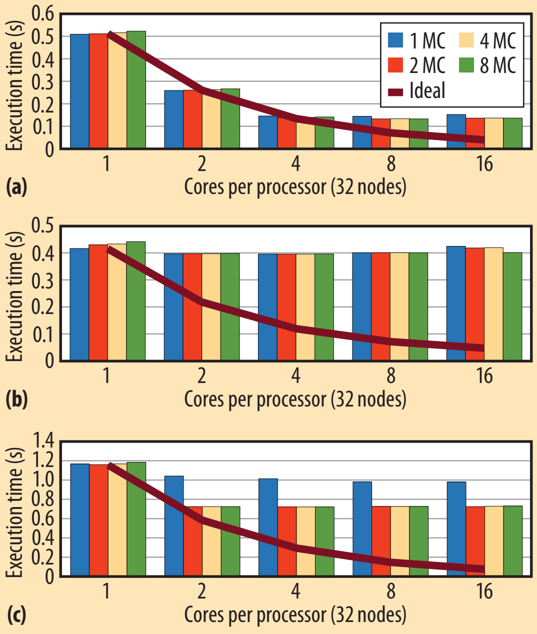

Figure 4. Execution times of three benchmark applications versus number of cores per processor in a 32-node XMT-like system using the proposed multicore design: (a) Aho-Corasick string matching, (b) breadth-first search (400,000 vertices, 400 neighbors per vertex), and (c) matrix multiplication (matrices of 2,000 × 2,000 elements). For each application, results are shown for different numbers of memory interfaces per processor.

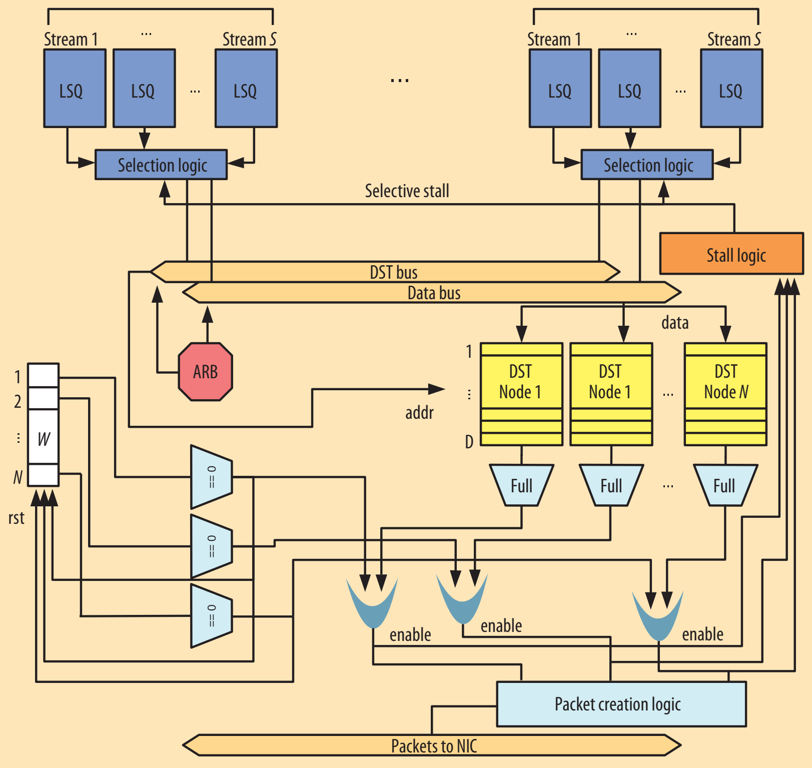

Figure 5. Hardware implementation of the aggregation logic.

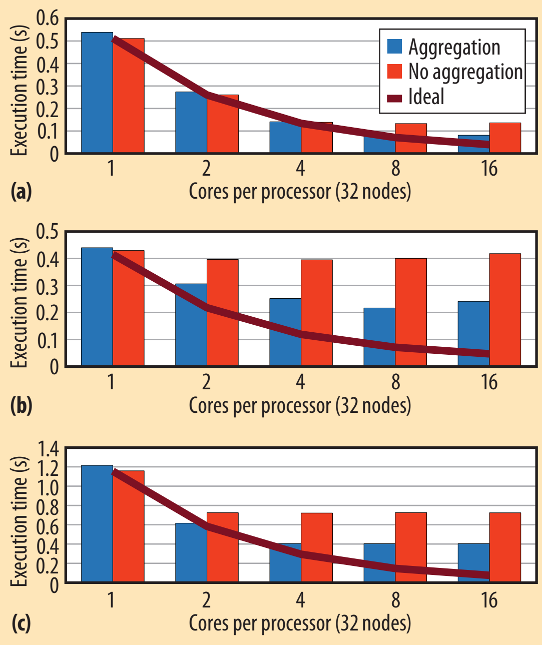

Figure 6. Execution times of three benchmark applications versus number of cores per processor in a 32-node XMT-like system using the proposed multicore design with an integrated aggregation logic mechanism: (a) Aho-Corasick string matching, (b) breadth-first search, and (c) matrix multiplication. For each application, results are shown with aggregation logic enabled and disabled. The processors have two MCs each.

in the buffer, the packet creation logic creates a network packet with all the content of the buffer and a single header. Each FIFO buffer is also associated with a downcounter that starts from cycle *W*. If the downcounter reaches 0, the packet creation logic is triggered even if the buffer is not full. If the buffer gets full before the downcounter reaches 0, the module sends the packet immediately and resets the downcounter to *W*. A selective stall logic module prevents the processor from generating further references with the same DST when a buffer is full but the network packet cannot be generated, such as when the aggregation logic is currently creating another packet.

In a system that supports a high number of nodes, implementing a FIFO buffer for each DST can require a large amount of on-chip memory. To reduce this requirement, it is possible to exploit multistage and hierarchical aggregation schemes, where every buffer collects memory references directed to groups of nodes. A node in the group then distributes references to their specific destinations. The tradeoffs of this approach are slightly higher aggregation logic complexity and delay. However, further increasing the number of threads per processor can mitigate the increased delay.

Scalability analysis

We integrated our aggregation mechanism in a simulated model of a 32-node XMT and initially evaluated the tradeoffs between buffer size *D* and time window length *W*. We found the best parameters to be $D = 16$ and $W = 32$. This result is obviously dependent on the number of nodes in the machine, as the time necessary to fill the buffers and the average network latency increase with the number of nodes.

We reexecuted the scalability analysis of the multicore configuration shown previously in Figure 4 using the same three benchmarks. Figure 6 compares the results with and without aggregation logic; in this case, each processor had two MCs, which we found to be the best tradeoff in the previous plots. The performance improvements are significant in all of the benchmarks, getting closer to ideal scaling. Although the aggregation mechanism adds more latency to remote memory operations (the time windows plus the overheads for packet generation), the cores have a sufficient number of threads to tolerate the additional overhead.

The current trend in high-performance computing is toward systems that employ processors with advanced cache architectures targeted at floating-point-intensive computations. These solutions are amenable to applications with high computational density, high locality, and regular data structures. However, they do not cope well with the requirements of emerging data-intensive irregular applications, which present large unstructured datasets with poor locality, high synchronization intensity, and memory footprints well beyond the size available on common compute nodes. These applications justify a renewed interest in architectures that can tolerate rather than reduce latency through massive multithreading, and have simpler whole-machine programming models.

Novel approaches such as data aggregation and compression will be required to enable better utilization of future interconnection infrastructures and significant increases in performance. Validating these approaches will also require new tools such as optimized scalable simulators and prototyping platforms. C

References

1. P. Kogge et al., "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," DARPA Information Processing Techniques Office, 2008; www.cse.nd.edu/Reports/2008/TR-2008-13.pdf.
2. J. Feo et al., "ELDORADO," *Proc. 2nd ACM Int'l Conf. Computing Frontiers* (CF 05), ACM, 2005, pp. 28-34.
3. J.L. Shin et al., "A 40 nm 16-core 128-Thread CMT SPARC SoC Processor," *Proc. 2010 IEEE Int'l Solid-State Circuits Conf.* (ISSCC 10), IEEE, 2010, pp. 98-99.

4. "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," white paper, Nvidia, 2009; www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.
5. R. Brightwell et al., "SeaStar Interconnect: Balanced Bandwidth for Scalable Performance," *IEEE Micro*, May 2006, pp. 41-57.
6. A. Tumeo, O. Villa, and D.G. Chavarría-Miranda, "Aho-Corasick String Matching on Shared and Distributed-Memory Parallel Architectures," *IEEE Trans. Parallel and Distributed Systems*, Mar. 2012, pp. 436-443.
7. O. Villa et al., "Fast and Accurate Simulation of the Cray XMT Multithreaded Supercomputer," preprint, *IEEE Trans. Parallel and Distributed Systems*, 2012; http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.70.
8. S. Secchi, A. Tumeo, and O. Villa, "Contention Modeling for Multithreaded Distributed Shared Memory Machines: The Cray XMT," *Proc. 11th IEEE/ACM Int'l Symp. Cluster, Cloud and Grid Computing* (CCGRID 11), ACM, 2011, pp. 275-284.
9. B. Arimilli et al., "The PERCS High-Performance Interconnect," *Proc. 18th IEEE Symp. High-Performance Interconnects* (HOTI 10), IEEE, 2010, pp. 75-82.

Antonino Tumeo *is a research scientist at Pacific Northwest National Laboratory. His research interests include modeling and simulation of high-performance architectures, hardware-software codesign, FPGA prototyping, and GPGPU computing. Tumeo received a PhD in computer engineering from Politecnico di Milano, Italy. Contact him at antonino.tumeo@pnnl.gov.*

Simone Secchi *is a postdoctoral research associate at Pacific Northwest National Laboratory. His research interests include modeling and parallel software simulation of high-performance computing architectures, FPGA-based energy-aware emulation of multiprocessor systems, and advanced network-on-chip architectures. Secchi received a PhD in electronic and computer engineering from the University of Cagliari, Italy. Contact him at simone.secchi@pnnl.gov.*

Oreste Villa *is a research scientist at Pacific Northwest National Laboratory. His research interests include computer architectures and simulation, accelerators for scientific computing, GPGPU, and irregular applications. Villa received a PhD in computer engineering from Politecnico di Milano. Contact him at oreste.villa@pnnl.gov.*