Overview of the Next Generation Cray XMT

Andrew Kopser *and* **Dennis Vollrath**, *Cray Inc.*

ABSTRACT: *The next generation of Cray's multithreaded supercomputer is architecturally very similar to the Cray XMT, but the memory system has been improved significantly and other features have been added to improve reliability, productivity, and performance. This paper begins with a review of the Cray XMT followed by an overview of the next generation Cray XMT implementation. Finally, the new features of the next generation XMT are described in more detail and preliminary performance results are given.*

KEYWORDS: Cray, XMT, multithreaded, multithreading, architecture

1. Introduction

Cray XMT

The Cray XMT is a scalable multithreaded computer built with the Cray Threadstorm processor. This custom processor is designed to exploit parallelism that is only available through its unique ability to rapidly context-switch among many independent hardware execution *streams*. The Cray XMT excels at large graph problems that are poorly served by conventional cache-based microprocessors.

The Cray XMT is a shared memory machine. The programming model assumes a flat, globally accessible, shared address space. The memory distribution hardware deliberately scatters data structures across the machine so that the application developer is neither required nor encouraged to worry about data placement in order to achieve good results. As such, the primary goal of the hardware design team is to provide the best possible bandwidth when all processors randomly access global shared memory. Each potential bandwidth limiter is evaluated within this context. The most significant of these are the processor injection bandwidth to the network, the network bisection bandwidth, and the DIMM bandwidth. The current Cray XMT is built within the Cray XT3 infrastructure and uses DDR1 technology [2]. For random access applications, performance is limited by the DIMM bandwidth.

The primary objective when designing the next generation Cray XMT was to increase the DIMM bandwidth and capacity. It is built using the Cray XT5 infrastructure with DDR2 technology. This change increases each node's memory capacity by a factor of eight and DIMM bandwidth by a factor of three.

Reliability and productivity were important goals for the next generation Cray XMT as well. A source of frustration, especially for new users of the current Cray XMT, is the issue of hot spots. Since all memory is globally accessible and the application has access to approximately one hundred hardware streams on each processor, it is relatively simple to construct an application that oversubscribes a particular location in memory. When this happens, the memory controller and network back up, resulting in performance degradation. In extreme cases, system services can be disrupted. In the next generation Cray XMT, this issue is addressed through the addition of hot spot avoidance logic in the processor.

2. The Cray XMT

As the disparity between processor and memory speeds has grown, processor architectures have adapted to deal with increasing latencies to memory. Vector computers attempt to *amortize* the latency by accessing

data in large blocks. Cache-based architectures attempt to *reduce* the latency by keeping the working set of the application in the processor's cache. However, when spatial and temporal locality of data is minimal, such as in many graph applications, these methods are not as effective. Multithreaded processors *tolerate* memory latency by working on many independent threads in parallel [1].

On the Cray XMT, threads are very lightweight software objects. Threads are mapped onto hardware *streams*. A stream stores the thread state and executes its instructions. Stream creation requires a single instruction and may be executed from user space. Typically, the compiler generates many more threads than the number of streams in the machine. These threads are multiplexed onto the hardware streams.

To facilitate coordination among many streams, the Cray XMT supports *extended memory semantics*. Each 64-bit word of memory in the Cray XMT is tagged with several state bits, the most important of which is the *full/empty* bit. Each memory operation can interact with the full/empty bit. For example, the *readfe* operation waits for a memory location to be full, loads it, and atomically sets it empty. This functionality allows for very fine-grained synchronization among streams.

2.1 The Cray XMT Blade

A logical view of the Cray XMT blade is shown in Figure 1. Each of four Cray Threadstorm3 processors is connected to four DDR1 DIMMs. Each processor is connected to a Cray Seastar network chip with a HyperTransport link. The remaining Cray Seastar network links are not shown here.

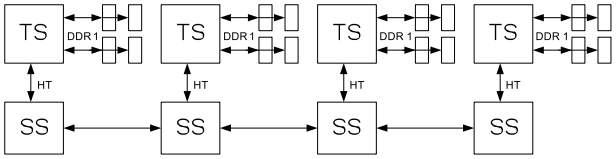

Figure 1: Cray XMT Blade Logical View

2.2 The Cray Threadstorm3

A high level view of the Cray Threadstorm3 used by the Cray XMT is shown in Figure 2. All memory references pass through a crossbar switch. Remote memory traffic is sent through a HyperTransport link to the network.

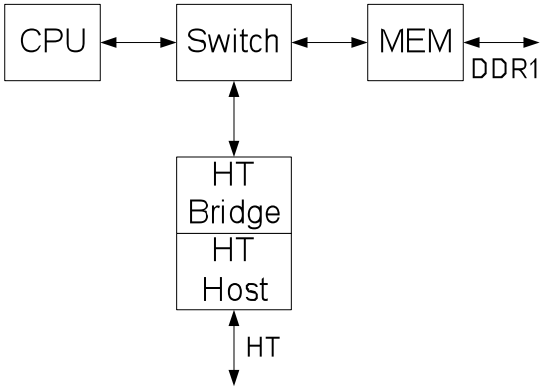

Figure 2: Cray Threadstorm3

2.3 Cray Threadstorm Processor

The Cray Threadstorm processor contains 128 independent instruction *streams*. Each stream includes its own complete set of register state. On each clock cycle, the instruction issue unit selects a stream to execute from among the pool of ready streams. Each instruction consists of up to three *operations*, one from each of three execution units. The *A-unit* performs arithmetic operations as well as a rich set of bit and bit matrix operations. The *C-unit* is responsible for branching but also includes arithmetic and bit operations. The *M-unit* handles all memory references. When an *M-operation* is issued, the associated *lookahead* field in the instruction indicates how many more instructions may be executed by this stream before the M-operation completes. This allows each stream to keep running while up to eight memory references are being serviced on its behalf by the M-unit.

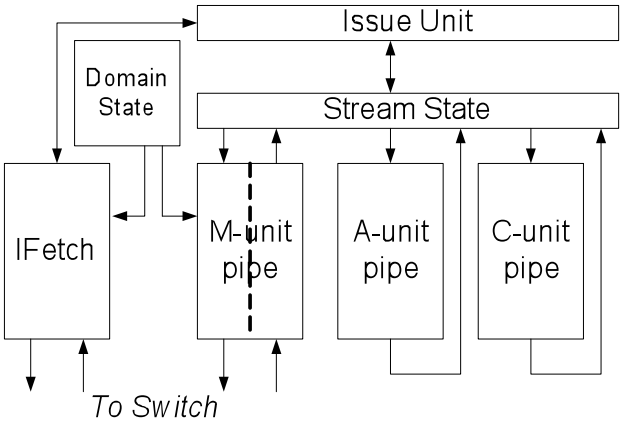

Figure 3: Cray Threadstorm CPU

Each application runs in one of sixteen protection domains. Each protection domain contains program and data state descriptors that define the privileges and

mappings for streams executing within that domain. Each protection domain also includes a variety of performance counters.

2.4 M-unit

When the M-unit receives an M-operation from the instruction issue unit, it first stores the information needed to retry the operation in the Data Control and Data Value Registers. Assuming the virtual address is valid, the data address translation proceeds as follows. First, the segment number of the virtual address is used to select a data map entry. Data map entries are stored in memory and cached by the data map translation lookaside buffer (DTLB) in the M-unit. The data map entry is used to relocate the virtual address into a logical unit number and logical unit offset. In addition, the data map entry specifies whether the address is to be *scrambled* and/or *distributed*. Certain local data structures, such as the program text and the data map entries themselves, are neither scrambled nor distributed. Most addresses, however, are both scrambled and distributed. The purpose of scrambling and distribution is to guarantee that memory references of any access pattern larger than a cache line are evenly spread across all memory controllers and memory banks. Scrambling and distribution does not affect the lower six bits (the cache line offset) of the address. During the scrambling step, the logical unit offset is multiplied by a constant bit matrix in order to hash the address. To distribute, the logical unit number and the scrambled offset are appended and then divided by the system size. The quotient is the physical unit offset and the residue is the physical unit number.

When data address translation has completed successfully, if the M-unit has the appropriate network credits, the M-unit releases the packet to the Switch. If the address is local, it is directed to the on-chip memory controller. Otherwise, it is sent over the HyperTransport link to the network. When a normal response arrives at the M-unit, if the operation was a load, the M-unit updates the register file with the data result. It also informs the instruction issue logic that it has completed so that the barrier created by the operation's lookahead value may be lifted. If an M-operation was a synchronized operation such as a *readfe* operation, the M-unit may receive a busy response instead of a normal response. In this case, the M-unit puts the operation in its Retry Queue. When traffic permits, the M-unit retries the operation. These retries do not use instruction issue slots but they do use network and memory bandwidth. The M-unit keeps track of how many times each operation has been retried. Eventually, the operation either completes successfully or exceeds its retry limit as specified by the data state descriptor. In this case, an exception is raised and the trap handler takes over.

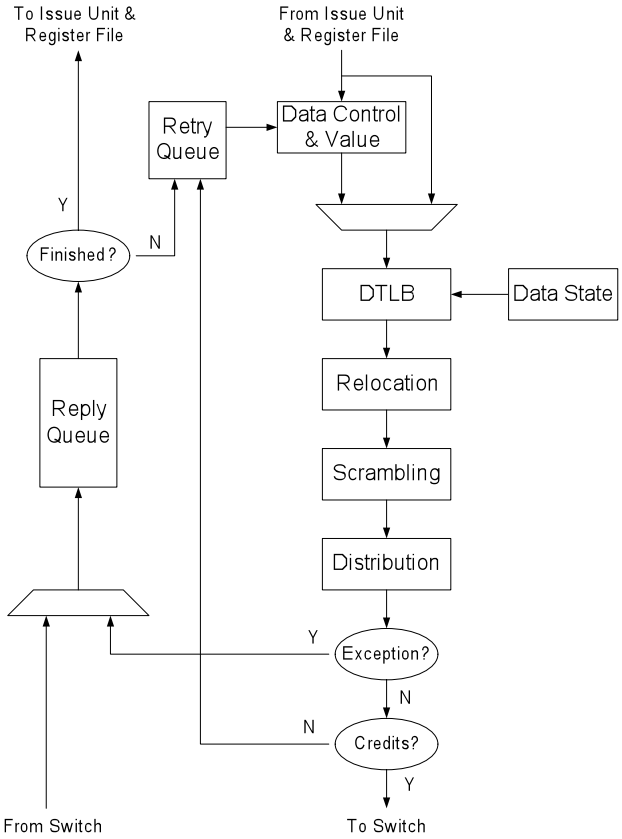

Figure 4: M-unit

2.5 Remote Memory Access

Both the Cray Threadstorm processor and the Cray Seastar network ASIC contain a Remote Memory Access (RMA) block. The RMA blocks reside on either end of the HyperTransport link and serve three important purposes. First, the RMA blocks allow up to 512TB of memory to be referenced directly without the need for messaging, windowing, or remote translation. Second, support is provided for the Cray XMT's extended memory semantics, which require a richer set of function, control, and response codes than may be achieved natively with HyperTransport. Finally, multiple remote references are *encapsulated* into each HyperTransport packet to allow more efficient use of the interface.

All RMA traffic is sent across the HyperTransport link in the form of posted writes. As requests are received in the HT Bridge on the Cray Threadstorm, the RMA block encapsulates these requests into the 64-byte payload of a posted write packet. The eight-byte HyperTransport header is used primarily to identify the packet as an RMA packet. On the Cray Seastar, the HyperTransport packet is unpacked and each request is sent to the appropriate

node. When a request packet arrives at its destination at the remote node, it is encapsulated on the Seastar with other requests, sent over the HyperTransport link, and once again unpacked on the Cray Threadstorm. This process is repeated in reverse for response packets. For symmetric traffic (each node making and servicing requests simultaneously), this method allows the Cray Threadstorm to sustain 100 million single word memory references per second.

2.6 Memory System.

The Cray XMT memory system processes requests from local and remote CPUs as well as the Seastar DMA engines. Memory operations can succeed or fail based upon the access control bits in the request used in conjunction with Extended Memory Semantics (EMS) bits contained in the memory word. These four EMS bits consist of:

- Full/Empty bit (F/E)
- Forwarding Bit
- Two Trap bits

All memory words have the F/E bit defined, but only pointers can have the remaining three bits. A separate control bit in each memory word indicates whether or not the location has forward and trap bits defined.

System memory is constructed with up to four slots of PC3200 (200MHz) DDR1 DIMMs. These are arranged in ganged pairs of two DIMMs providing a 16B data interface. The DIMMs are read in bursts of four, resulting in a complete cache line of 64 bytes read or written for each operation. While multiple words from the cache line are used in some situations, the principal mode of memory access is single word requests to random memory locations. Using 2MB DIMMs, the memory capacity is 8GB per node.

The memory system has two clock domains, 500MHz and 200MHz. Operations cross between the two domains in a synchronous manner.

A block diagram of the Cray XMT memory system is shown in Figure 5.

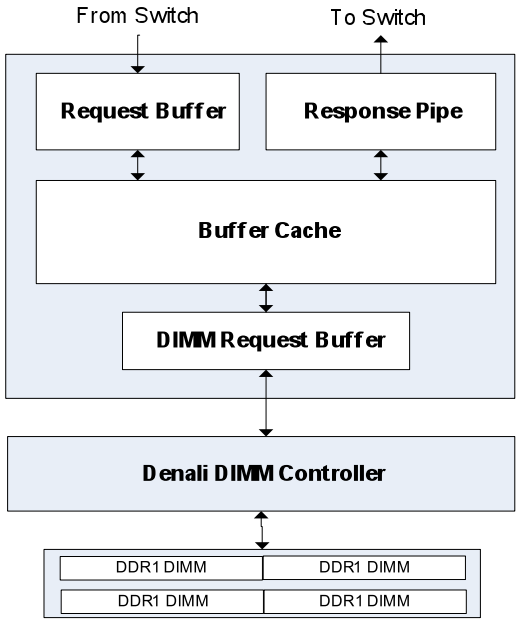

Figure 5: Cray XMT Memory Subsystem

2.6.1 Mem_top

The Mem_top block handles traffic to and from the switch at 500MHz. It is composed of four major functional sub-blocks.

The Request Buffer sub-block manages up to 32 data requests and eight instruction requests at a time. These are stored in a buffer and are prioritized for processing based upon age, state of the buffer cache, and a configurable instruction/data priority setting.

The Buffer Cache sub-block contains 128kB of storage and is the hub of the memory system through which all operations pass. This storage is organized as a four-way set associative cache with 512 indices. Each entry stores a complete 64B cache line and associated tag state.

If the request misses in the Buffer Cache, a DIMM read is issued to the DIMM Request Buffer. This sub-block has storage for 24 reads and 24 writes. It issues read and write requests to the Denali Software DIMM controller, handles responses, and maintains storage for pending DIMM operations. The DIMM Request Buffer sub-block straddles the 500MHz / 200MHz boundary.

When a memory operation has been satisfied, it is released to the Response Pipe for processing and returned

to the requestor. This logic takes the current value of the memory location (now in the Buffer Cache) as well as the request information and emits a response to the Switch and potentially a modified value for memory. The AMO logic in the response pipe handles Fetch & Add, partial word stores, and pass-through functions. The returned result code is based upon the memory location's EMS bits and the requesting operation's access control bits.

2.6.2 DIMM Controller

Reads that are not found in the Buffer Cache and modified cache locations that are evicted are sent to the DIMM controller. This 200MHz logic IP from Denali Software consists of RTL synthesizable Verilog code that integrates the controller and PHY functions. It schedules the DIMM operations and controls the initialization and refresh functions. Operations are scheduled to optimize the utilization of the control and data signals, thus maximizing bandwidth. A highly boot-time configurable Command and Status Register set accommodates DIMMs with different capacities and timing characteristics.

3. The Next Generation Cray XMT

The next generation Cray XMT is the latest multithreaded offering from Cray. It is architecturally very similar to the original Cray XMT, but it is built with the new Cray Threadstorm4 processor in the Cray XT5 infrastructure. This infrastructure uses DDR2 technology and provides twice the number of DIMM slots per node as the Cray XMT. The Cray Threadstorm4 takes advantage of the upgraded memory system and provides important new hot spot avoidance techniques.

3.1 The Next Generation Cray XMT Blade

The next generation Cray XMT Blade is shown in Figure 6. Each node includes two Cray Threadstorm4 ASICs. Those connected to the network function as processors whereas those labelled TS_{mc} operate as memory controllers only. One processor per node is sufficient to saturate the network bandwidth. The two Cray Threadstorm4 ASICs within a node are connected by a custom interface called the Node Pair Link (NPL).

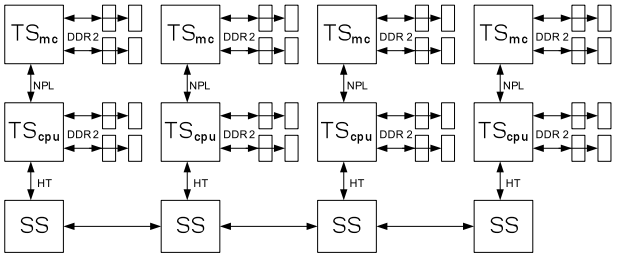

Figure 6: Next Generation Cray XMT Blade

3.2 The Cray Threadstorm4

A block diagram of the Cray Threadstorm4 is shown in Figure 7. There are two significant differences as compared to the Cray Threadstorm3. First, the memory controller has been upgraded to support DDR2. Second, the NPL link has been added as another port on the Switch. Note that for the TS_{mc} sockets in Figure 6 above, the CPU and HT interfaces are disabled.

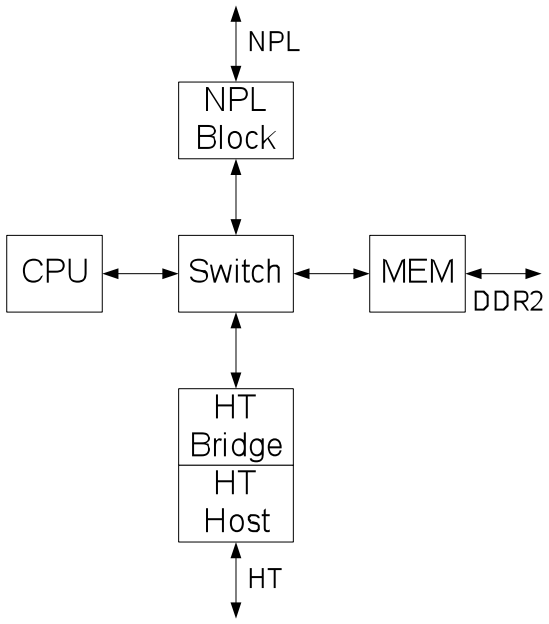

Figure 7: Cray Threadstorm4

4. Next Generation XMT Memory System

In the next generation Cray XMT, the memory system was changed to support DDR2. This is shown in Figure 8. The DIMM control block is now IP from Northwest Logic. It is internally organized into a 144-bit wide controller and two 72-bit wide PHY blocks.

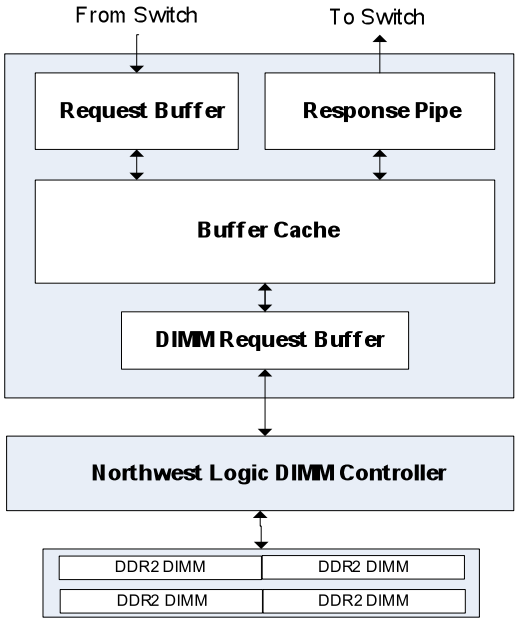

Figure 8: Next Generation XMT Memory System

4.1 Why DDR2?

With a short 18-month design cycle, keeping discretionary risks to a minimum was crucial. Towards this end, we decided to leverage existing Cray infrastructure as much as possible. By adopting proven mechanical, power, and cooling implementations, the design team could focus more on the processor design. The production-deployed Cray XT5 platform offered this opportunity. It supports registered DDR2 memory technology. In addition to risk mitigation, DDR2 also offers performance advantages to the next generation Cray XMT. When accessing memory in a Double Data Rate device, a minimum number of words must be read out in a single operation. This parameter, called the burst size, is four for DDR2. With a 16B wide memory channel and burst length of four, each access reads or writes 64 bytes of data. Since the dominant memory access mode for the Cray XMT is to single word random addresses, 56 of these bytes are often not used. We also considered DDR3, which has a minimum burst size of eight. This leads to a minimum access of 128 bytes of data, only eight of which may be used. While a DDR3 channel could run faster (twice as fast would provide equivalent single word random access bandwidth), DDR2's smaller burst size allows us to run at slower speeds with lower power and larger timing margins.

4.2 Capacity

The next generation Cray XMT supports eight DDR2 slots per node, twice that of its predecessor. With the larger capacity DIMMs available in DDR2 as compared to DDR1, the next generation Cray XMT supports up to 64GB per node, or 32TB of flat shared memory for a 512-processor system. This is an 8x improvement over the current Cray XMT.

4.3 Bandwidth

Memory bandwidth was improved as well in the next generation Cray XMT. The DIMM clocks run at 300MHz, compared to the current rate of 200MHz. In addition, there are twice as many memory channels per node. This leads to a three-fold improvement in overall memory bandwidth and ensures it is not the critical system resource.

4.4 RandomAccess Performance

The RandomAccess benchmark is a good representation of the sort of application that excels on the Cray XMT. It measures the global memory bandwidth of the system when all processors make random accesses to memory. It reports billions of updates per second. Each update includes two single-word memory operations--one load and one store. On the current Cray XMT, this application is limited by DDR1 bandwidth. On the next generation Cray XMT, however, the increased memory bandwidth allows the network to be saturated as shown in Figure 9. According to the optimized results of the 2010 HPC Challenge [3], these results are only exceeded by other machines with at least one thousand processing nodes.

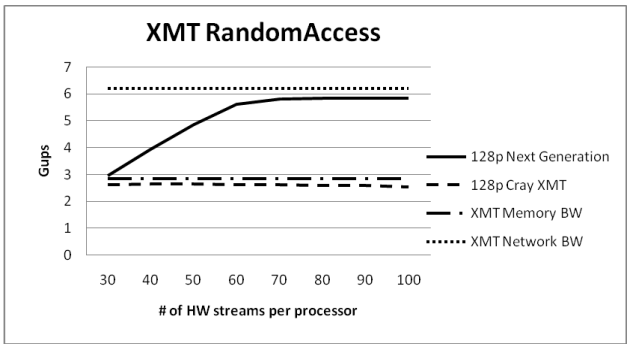

Figure 9: XMT RandomAccess performance

5. Hot Spot Avoidance

One challenge for application writers on the current Cray XMT is the inadvertent generation of hot spots. The Cray XMT provides useful synchronization primitives and atomic operations, but if all streams on the machine try to reference the same location in memory, a hot spot can be created. This causes a backup in the affected memory module that extends into the network. The typical result is poor performance for the application, but in the worst case, system services can be affected as well. For new users in particular, these sorts of problems can be difficult to diagnose. The solution often requires distributing data structures and making a simple application more complicated. In the next generation Cray XMT, this problem was addressed by adding content addressable memories (CAMs) in the processor to reduce unnecessary traffic on the network.

5.1 Synchronized References

Perhaps the simplest way to generate a hot spot on the current Cray XMT is to have many streams make synchronized references to the same memory location. A *readfe* operation waits for a memory location to become full and then atomically reads the location while setting it empty. Likewise, the *writeef* operation waits for a location to become empty and then atomically writes the location while setting it full. If exclusive access to a commonly accessed data structure is maintained through these primitives, a hot spot can occur.

Consider the situation in which one stream has emptied the synchronized location and many others are trying to read it. In each Cray XMT processor, perhaps 100 streams make the same *readfe* request. The M-unit sends these *readfe* requests out onto the network. Eventually, a response is received for each of these requests. At most one response contains the requested data; the others are guaranteed to receive a busy response. Those streams that receive a busy response are retried. For an extended period of time, each processor always has about 100 requests outstanding in the network.

Since at most one of these *readfe* may succeed, it is counterproductive to release the others to the network. In the next generation Cray XMT, the *SynchRef CAM* has been added to the M-unit. When the *readfe* would be injected into the Switch, it checks in the SynchRef CAM first. If a matching address is found, the *readfe* operation is returned to the Retry Queue. Otherwise, the address is added to the CAM and the operation proceeds normally. Thus, each processor never has more than one *readfe* operation outstanding to the same location.

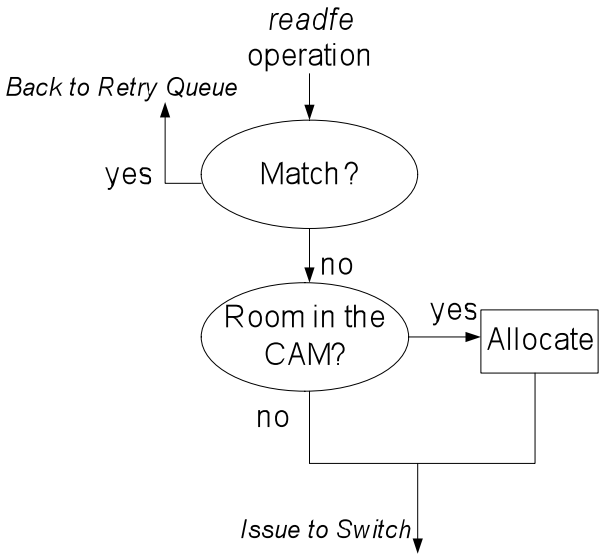

Figure 10: SynchRef CAM operation

5.2 readfe Reduction

To test the effectiveness of the SynchRef CAM, an extremely degenerate program was tested on 128-processor versions of the current and next generation Cray XMT. This program performs a million-element reduction on a single memory location using *readfe* and *writeef* to guarantee atomicity. As a result, only one stream at a time ever does any useful work and performance degrades as processors are added. 100 streams are requested on each processor. The kernel of the reduction is shown here:

```
for (int i=0; i < N; i++){
    int local_sum;
    local_sum = readfe(&sum);
    local_sum += rand_array[i];
    writeef(&sum, local_sum);
}
```

As shown in Figure 11, on the current Cray XMT, the application generates billions of retries and millions of retry limit traps; performance suffers as a result. It was not feasible to run the application on more than 64 processors. On the next generation Cray XMT, however, a negligible number of traps occur. At 96 and 128 processors, we begin to see more retries and traps as the network injection limit at the busy memory node becomes an issue. While this code still reflects a very poor way to perform a reduction, the next generation Cray XMT is much more forgiving than its predecessor.

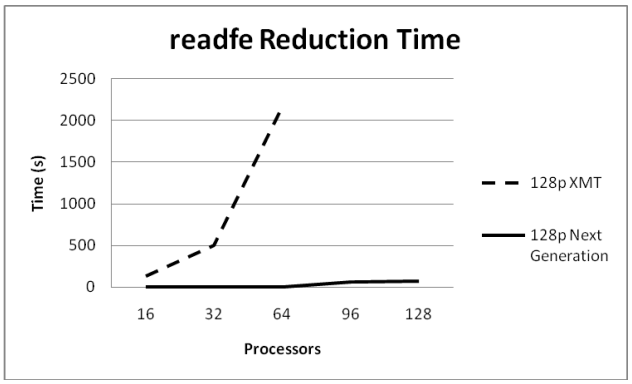

Figure 11(a): readfe Reduction execution time

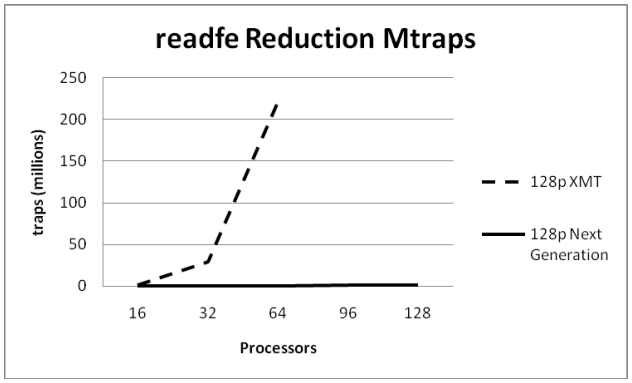

Figure 11(b): readfe Reduction traps (millions)

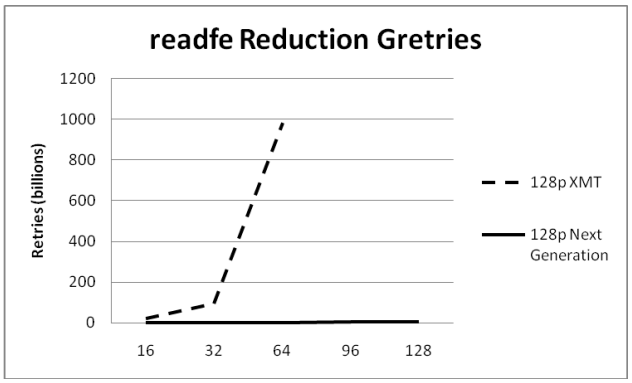

Figure 11(c): readfe Reduction retries (billions)

5.3 Fetch & Add Combining

Another commonly used method for coordinating among many streams or for updating a shared accumulator is to use the Cray XMT's atomic Fetch & Add operation. Unlike the *readfe* operation, the Fetch & Add operation is satisfied without the need for retrying. However, if many streams on all processors try to access the same memory location, the affected memory controller is still oversubscribed by a factor that grows with the number of processors in the system.

In the next generation Cray XMT, Fetch & Add operations are combined in the M-unit in order to reduce network and memory bandwidth requirements. The Fetch & Add Combining logic includes a small, four-entry, *Fetch & Add Combining CAM* (FACC), a 128-entry *Fetch & Add Linked List* (FALL), and a sixteen-entry *Fetch & Add Retirement CAM* (FARC).

When a Fetch & Add operation would be issued to the Switch, it checks for a match in the FACC. If a match is not found and there is an available entry, the packet allocates in the FACC and waits for a specified period of time to permit combining. The FACC includes an accumulator for the data and a pointer to the first element (initially null) of a linked list of its dependents. If a match is found and there is room in the FALL, the packet allocates in the FALL, adds itself to the FACC entry's dependent list, and updates the accumulator in the FACC with its data. If a packet cannot make the desired allocation (either a match is found but there is no room in the FALL, or a match is not found but there is no room in the FACC), the Fetch & Add operation is immediately released to the Switch.

When an entry in the FACC has waited for the specified period of time to allow combining, it allocates an entry in the FARC, removes itself from the FACC, and a network request is created. This Fetch & Add request includes the address and identifiers associated with the Fetch & Add operation that first allocated in the FACC, but specifies the accumulated data of all Fetch & Add operations that were combined in this entry. The FARC contains a pointer to the first element of the FALL. When the response to the Fetch & Add request is received by the M-unit, the FARC is consulted and the linked list is traversed. Each response to the register file is constructed by re-accumulating the data from memory with the data from the elements in the linked list. The FARC and FALL entries are deallocated as they are referenced. Figure 12 gives an example of three Fetch & Add requests that are combined in the M-unit.

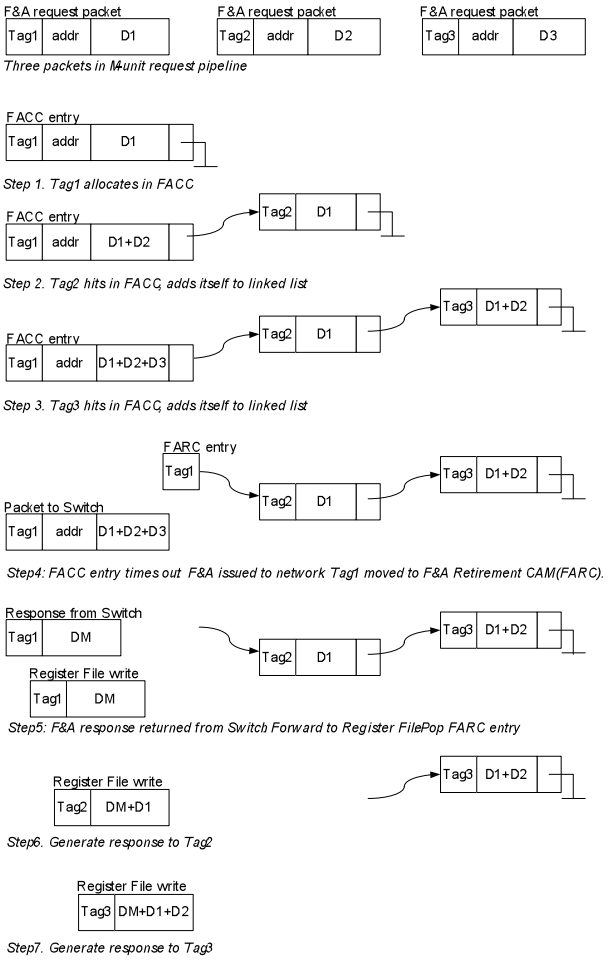

Figure 12: Fetch & Add Combining Example

What about that wait time? Since Fetch & Add operations are required to wait to allow combining, we must consider applications that do not benefit from combining. For example, when generating a histogram of a large table, very little combining may be expected. Consider an application with 100 streams building a histogram with Fetch & Add operations. If each stream always has only one Fetch & Add operation outstanding at once, then four (the size of the FACC) out of 100 Fetch & Add operations are delayed. Early experience has shown that adding latency to 4% of these Fetch & Add operations has little to no effect on performance. As long as there is sufficient concurrency in the application, the additional latency is completely hidden. The benefits of this feature as demonstrated below far exceed the potential drawback.

5.4 Fetch & Add Reduction Performance

On the current Cray XMT, when an accumulator is frequently updated in memory using the Fetch & Add operator, it is common practice to replicate the accumulator to spread out the updates. When an update is to be made, one of the copies is randomly selected. We wrote a simple program to compare the usefulness of this optimization on the current and next generation Cray XMTs. The kernel of this program is shown here:

```
for(int i = 0; i < num_updates; i++) {
    int copy = MTA_CLOCK(0) & (num_copies-1);
    int_fetch_add(&array[copy*CACHE_LINE], 1);
}
```

The MTA_CLOCK intrinsic returns the free-running processor clock and is used here as an inexpensive way to generate a somewhat random value. The number of copies (num_copies) is required to be a power of two. Note that each copy must reside in its own cache line so that the copies are mapped to different memory controllers.

This program was run with 32 billion updates on 512-processor machines. As shown in Figure 13, simply updating a single memory location on the current Cray XMT creates a significant hot spot and results in very poor performance. Performance is improved as copies are added. On the next generation Cray XMT, however, best performance is achieved with a single accumulator. As copies are added, the FACC becomes polluted and less effective as a result. When many copies are added, the FACC becomes less of a factor and performance more closely resembles that of the original Cray XMT. In all cases, however, the FACC improves performance. By optimizing the simplest implementation, new users on the next generation Cray XMT should not have to learn about this ugly coding trick.

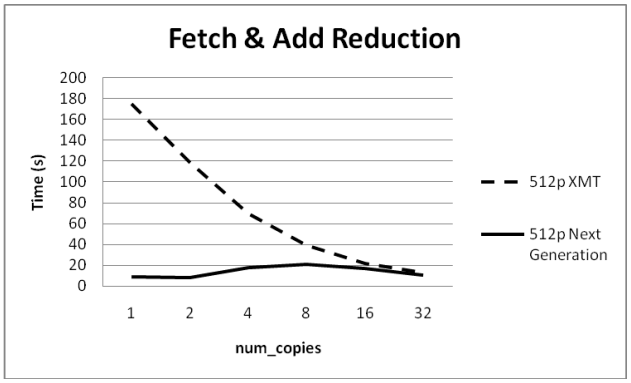

Figure 13: Fetch & Add Reduction

6. Conclusion

The Cray XMT has proven to be very effective at solving applications with very large data sets that are

randomly accessed. Many of these applications are currently limited by the DDR1 bandwidth. The next generation Cray XMT eliminates this bottleneck by providing three times the bandwidth using DDR2 technology. The memory capacity is also greatly increased, allowing much larger problems to be solved.

Hot spots have been a very difficult issue for many programmers of the current Cray XMT. The next generation Cray XMT addresses this by adding hot spot avoidance techniques for synchronized references and Fetch & Add operations. These techniques prevent services from being disrupted in all cases, and often allow the simplest implementation to perform most effectively. This should translate to improved productivity for application developers.

References

[1] Alverson, R. et al., "The Tera Computer System", *Proceedings of the 4th International Conference on Supercomputing*, p1-6, June 11-15, 1990, Amsterdam, The Netherlands.

[2] Feo, J. et al, "Eldorado", *Proceedings of the 2nd Conference on Computing Frontiers*, p 28-34, May 4-6, 2005, Ischia, Italy.

[3] "HPC Challenge Benchmark Results – Condensed Results – Optimized Runs", accessed April 21, 2002, http://icl.cs.utk.edu/hpcc/hpcc_results.cgi?display=opt

Acknowledgments

The authors would like to thank the entire Cray XMT team for making it work and evaluating its performance.

About the Authors

Andrew Kopser is a senior principal engineer at Cray Inc. Email: kopser@cray.com. Dennis Vollrath is a principal engineer at Cray Inc. Email: dennis@cray.com. Both authors have worked on this multithreaded architecture since its origins at Tera Computer Company. Both are located at Cray Inc., 901 Fifth Avenue, Suite 1000, Seattle, WA 98164.