Introducing the Cray XMT

Petr Konecny
Cray Inc.
411 First Avenue South
Seattle, Washington 98104
pekon@cray.com

May 5th 2007

Abstract

Cray recently introduced the Cray XMT system, which builds on the parallel architecture of the Cray XT3 system and the multithreaded architecture of the Cray MTA systems with a Cray-designed multithreaded processor. This paper highlights not only the details of the architecture, but also the application areas it is best suited for.

1 Introduction

Cray XMT is the third generation multithreaded architecture built by Cray Inc. The two previous generations, the MTA-1 and MTA-2 systems [2], were fully custom systems that were expensive to manufacture and support. Developed under code name Eldorado [4], the Cray XMT uses many parts built for other commercial system, thereby, significantly lowering system costs while maintaining the simple programming model of the two previous generations. Reusing commodity components significantly improves price/performance ratio over the two previous generations.

Cray XMT leverages the development of the Red Storm [1] system Cray has built for Sandia National Laboratory and brought to market as the Cray XT3 [3]. The Cray XT3 is a large-scale distributed memory system with custom infrastructure and a network that can sustain high bandwidth with any packet size. Cray XMT replaces Opteron processors in the compute partition of the XT3 with a new implementation of MTA processor. This enables efficient injection of small messages into the network and transforms a distributed memory system into a shared-memory Cray XMT.

The components of early computer systems ran at the same speed. Processors ran at about the same speed as the memory systems. However over the past two decades the processor speeds have increased several orders of magnitude while memory speeds increased only slightly. Despite this disparity the performance of most applications continued to grow. This has been achieved in large part by exploiting locality in memory access patterns and by introducing hierarchical memory systems. The benefit of these multilevel caches is twofold: they lower average latency of memory operations and they amortize communication overheads by transferring larger blocks of data.

The presence of multiple levels of caches forces programmers to write code that exploits them, if they want to achieve good performance of their application. The problem is exacerbated on large-scale systems where most memory is remote and the latency of data access becomes much larger. The applications are forced to transfer larger and larger blocks of data in order to hide the latency and to overcome inefficiencies in handling small messages by conventional interconnection networks. This constrains the space of scalable algorithms and increases application complexity and development cost.

The Cray XMT employs a fundamentally different approach. Instead of relying on large message sizes to amortize overheads, it has efficient support for small message sizes. And instead of sending large blocks of data, it relies on thread level fine-grained parallelism to hide latencies and keep the system

components busy. In the next section we highlight architectural details of Cray XMT. In the third section we highlight features of the programming environment that enable the massive parallelism needed.

2 Hardware Architecture

Cray XMT is a shared memory system that efficiently exploits fine-grain parallelism in order to hide latencies and utilize available resources. It is largely based on Cray XT infrastructure and the software developed for Cray MTA-2 project. The system consists of interconnected cabinets housing compute and service modules. All modules have four Seastar2 chips which provide network interface and routing functionality. Each compute module has four custom multithreaded processors. A service module consist of two AMD Opteron processors and four PCI-X interfaces. All processors in the system use commodity DIMM memory.

2.1 Interconnection Network

The Seastar2 chips are connected into a 3D torus network. Despite inefficiencies in the suboptimal packet format, the network bandwidth is reasonably well matched to other paths in the system. Besides routing the network traffic, the Seastar2 translates remote memory access (RMA) requests and responses between the network packet format and the HyperTransport protocol. The communication is protected from error on a per link basis.

Because the most interesting mode of operation assumes uniformly distributed traffic, the network performance is expected to be dominated by the bisection bandwidth. This is sub-linear in the system size. It depends not only on the number of processors, but also on the exact topology of the system.

2.2 Multithreaded processor

Each multithreaded processor consists of four major components:

1. instruction execution logic
2. DDR memory controller, data cache
3. HyperTransport logic and physical interface
4. a switch that connects these three components

2.2.1 Instruction logic

The processor has 128 hardware streams and a 64 KB, 4-way set associative instruction cache divided equally among them. A hardware stream consists of 31 general purpose 64-bit registers, 8 target registers and a status word that includes the program counter. The processor schedules instructions onto three functional units, M,A and C. It stalls, if no stream has an instruction ready. The instruction word contains one operation (possibly a NOP) for each functional unit. On every cycle, the M unit can issue a memory operation, the A unit can execute an arithmetic operation and the C unit can execute a control operation or a simple arithmetic operation.

In order to eliminate dependencies and simplify the implementation, no stream can have more than one instruction in the pipeline at any given time. After issuing a memory operation, a stream can issue up to seven other instructions (which may include memory operations), before it has to wait for the completion of the first memory operation. Therefore the processor can keep 1024 memory operations in flight. The cycle speed is 500 MHz and it can execute three floating point operations per cycle. However we will see that the peak performance of 1.5 Gflop/s is largely theoretical as most workloads will be limited by bandwidth.

2.2.2 Memory interface

The processor provides an interface to commodity DIMM memory. Like MTA-2, the Cray XMT implements extended memory semantics. Each word of memory consists of 64 data bits and two state bits that are used for synchronization.

The logic of the memory controller handles not only reads and writes, but also a variety of atomic memory operations that manipulate the state and data bits. The requests for the memory operations can originate in any M-unit in the system, not only the one in the same processor. The responses are sent back to the originating M-unit.

The memory controller encodes in 288 bit blocks. It combines four words (64+2 bits each) with error correcting code in a block. The ECC code protects the block from single bit errors. The blocks are always transferred in pairs and the eight word cache lines are cached in a 128KB, 4-way set associative buffer. Note that this buffer never stores remote data and hence the system is trivially cache coher-

ent. All memory requests allocate space in the cache. While cache hits do have lower latency than cache misses, this benefit is not necessary to achieve peak bandwidth.

Assuming random access pattern the theoretical peak of the memory controller is 66 million cache lines per second, or 4.224 GB/s. The interface to the memory buffer supports 500 million single word operations per second, or 4 GB/s.

2.2.3 Network Interface

A HyperTransport link connects the processor to a Seastar2 chip. The link is used to transfer RMA requests and responses in and out of the network. The processor can also use it to perform DMA operations to transfer data between its nearby memory and the memory of a service processor. The theoretical peak bandwidth depends on the type of the operation performed. For instance the link can perform 140 million loads per second.

2.3 Other system components

The Cray XMT system reuses all of Cray XT infrastructure. This includes I/O, hardware supervisory system (HSS), power and cooling.

3 Software

Like Cray XT3 and MTA-2 system, the XMT utilizes multiple operating systems. The compute nodes form a partition that runs a single instance of MTK - a multithreaded operating system developed for the MTA-2 project. Each service node runs an instance of the Linux operating system.

3.1 Programming environment

The applications for the Cray XMT can be developed in C and C++ programming languages. Cray provides a compiler that is capable of parallelizing many common loops. The compiler itself runs on a login node and the applications are then launched on the compute partition. Besides parallelizing loops, the compiler supports language extensions for lightweight thread creation and synchronization. The goal of an XMT programmer is to expose enough parallelism to allow many operations to proceed concurrently, thereby hiding latencies.

Performance tools are an indispensable part of the development process on the XMT. They allow the programmer to view compiler annotations: which loop are parallelized, what mix of operations is in their kernels and how many streams are necessary to execute them efficiently. They also provide an insight into runtime application performance by analyzing hardware performance counters and profiling information.

3.2 HPCC Random Access

We present an application kernel and steps in its implementation on the Cray XMT. RandomAccess is one of the HPC Challenge Benchmarks [5]. It is a variant of the famous GUPS benchmark. It repeatedly performs a commutative update operator on data at randomly selected locations in a large table. In this case the operator is bitwise XOR. The serial version of the code is quite simple:

```
u64Int ran;
ran = 1;
for (i=0; i<NUPDATE; i++) {
  ran = (ran << 1) ^
        (((s64Int) ran < 0) ? 7 : 0);
  table[ran & (TableSize-1)] ^= ran;
}
```

It generates a sequence of random numbers using a linear feedback shift register and uses them to locate and update a value in a table. As written this code has data dependency between iterations and is not suitable for any parallel computer. To parallelize the code we employ function `HPCC_starts` which, given an integer `i`, returns the `i`-th value in the random number sequence. We rewrite the loop using this function:

```
for (i=0; i<NUPDATE; i++) {
  u64Int ran = HPCC_starts(i);
  ran = (ran << 1) ^
        (((s64Int) ran < 0) ? 7 : 0);
  table[ran & (TableSize-1)] ^= ran;
}
```

The XMT compiler can parallelize this loop without user intervention. It recognizes that the only dependency between iterations is through the updates of the table. Because XOR is commutative, the updates can be performed in any order as long as each update is atomic. The compiler generates a code that uses the state bits in each word to protect

it from unsafe concurrent modifications by multiple threads.

While this was a successful parallelization of the code it is not an efficient one. The function `HPCC_starts` is quite expensive. In order to amortize its cost we can manually block the parallel loop:

```
for (i=0; i<NUPDATE; i+=bigstep) {
  u64Int ran = HPCC_starts(i);
  for(j=0; j<bigstep; j++) {
    ran = (ran << 1) ^
          (((s64Int) ran < 0) ? 7 : 0);
    table[ran & (TableSize-1)] ^= ran;
  }
}
```

Now the inner loop is same as the one in the serial version of the benchmark and the Apprentice2 performance tool reveals that each iteration consists of five instructions with two memory operations. This includes all random number generation, address computation and the synchronized update to the table. Increasing the block size helps amortize the cost of `HPCC_starts`, while decreasing available parallelism.

3.2.1 RandomAccess Performance

The two memory accesses (load and store) in the kernel are the bottleneck in its performance. Each involves RMA request and response messages being sent across the network. Since the stream of addresses is pseudo-random, we can expect the loads to miss cache and hope that stores will hit it. In the best case the hardware will perform about two cache line transfers per update operation (upto a small fraction of dirty cache lines that stay in the cache at the end). Thus the peak is 33 million updates per second per processor. The bisection bandwidth does not become a limiting factor until the system size exceeds about 1000 processors.

Cray has built a prototype 64 processor XMT system. While this system is functional at the time of writing this paper, there are multiple known hardware bugs in the multithreaded processor. Avoiding them requires workarounds that negatively impact performance. Cray plans to re-spin the processor in the second half of 2007 to address the problems. Despite the shortcomings the system is capable of executing 1.28 billion updates per second, or 20 million updates per second per processor. This significantly exceeds the per processor performance of any published results. It is noteworthy that it exceeds even the performance of the AMD Opteron using the same directly attached DIMM memory.

3.3 Conclusion

We have highlighted features of the Cray XMT system that enable efficient implementation of the shared memory programming model:

- efficient support for small messages
- ability to hide latencies through increased concurrency
- parallelizing compiler
- performance tools

While performance of real applications remains to be determined, the Cray XMT system promises a solution for problems that are too difficult to solve on distributed memory systems. The common traits of strong Cray XMT applications are:

1. use lots of memory
2. lots of parallelism
3. fine granularity of data access
4. data is hard to partition
5. load balancing problems

Application areas that share these traits include:

- Problems on large graphs (intelligence, protein folding, bio-informatics)
- Optimization problems (branch-and-bound, linear programming)
- Computational geometry (graphics, scene recognition and tracking)

References

[1] Cray Red Storm Project [Online]. http://www.cray.com/products/programs/red_storm/

[2] Cray MTA-2 Project [Online]. http://www.cray.com/products/programs/mta_2/

[3] Cray XT3 system [Online]. http://www.cray.com/products/xt3/

[4] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In Proceedings of the Second Conference on Computing Frontiers, 2005, Ischia, Italy, May 4-6, 2005.

[5] HPC Challenge Benchmarks [Online]. http://icl.cs.utk.edu/hpcc/