Reducing Communication-Based Energy Consumption in HPC Memory Hierarchy

Meriem Ben-Salah and Gage Eads

Abstract—Excessive energy consumption is a limitation for High-Performance Computers (HPC), especially as we move towards exascale computing systems. In the 2008 ExaScale Computing Study by Yelick et al., current memory hierarchies are estimated to consume over 30% of single rack power. In this work, we model the energy consumption across the memory hierarchy (from L1 cache to main memory) of a 96-node Cray XT4 system using simulation and analytical models. Using hardware performance counters, we estimate the energy associated with MPI communication in the memory hierarchy for five NAS parallel benchmarks. We propose cache injection to efficiently integrate I/O data into ongoing computation, and show that this method results in DRAM energy reduction up to 47% for well-matched algorithms. We believe that the application of our proposed memory technology will help accomplish the implementation of energy efficient high performance computers.

I. Introduction

The energy problem is the "single most difficult and pervasive challenge" for building HPC systems, according to [1]. Power is consumed by functional units (CPUs and GPUs), DRAM main memory, interconnect (on-chip, between chips, between boards, and between racks), secondary storage, and cooling. The report’s “light node strawmen” - systems specially designed with customized processors for massive replication, such as the Blue Gene supercomputer series [2] - are an example of current systems and used as a baseline for comparison against new techniques and technology. The “light node strawmen” dissipate about 20% of total single rack power into main memory and over 10% into the cache hierarchy and DDR controller, resulting in over 30% of single rack power consumed by the memory hierarchy.

Ways to reduce memory-related energy include new low-power technology and reducing total memory accesses. One exciting new memory technology is 3D chip packaging with through-silicon vias. This technology removes off-chip signaling and drastically reduces the distance between CPU and main memory, thus removing power due to relatively high pin capacitance and long interconnect. One can modify their software to best exploit temporal and spatial locality, with techniques such as cache blocking, to reduce the data bouncing between levels of the memory hierarchy. Microarchitectural optimizations can also reduce energy consumption. Cache injection is a technique that efficiently transfers I/O data into ongoing computation by writing directly to a processor’s cache. If performed when data is needed, this technique can remove cache hierarchy traversal and two main memory accesses, and the energy cost associated with each access.

Communication-related accesses can contribute some amount to total memory accesses, and the standard data-transfer mechanisms leave much room for optimization.

Cache injection has been investigated previously as a means to reduce memory traversal processor stalls, with overall performance improvement.

In this work we explore cache injection’s effects on memory access behavior and, using energy models from Cacti and DRAMSim2, estimate the impact on the total memory hierarchy energy. We show that cache injection, which requires little architectural change, can greatly improve algorithms with frequent small (within the cache size) message communication.

II. Background

Thanks to the courtesy of Kathy Yelick, etc., we have access to the NERSC supercomputers to run our benchmark suite. We are using Silence, a single-cabinet 96-node software test platform that is predominantly unused. Thus, our results are not affected by other applications occupying shared resources (i.e. the inter-node network and routers).

A. Silence

Each of Silence’s Cray XT4 nodes contain a 65-nm quad-core 2.3 GHz AMD Opteron “Budapest” model. Each core has a private 64-KB L1 instruction cache, 64-KB L1 data cache, and an exclusive 512-KB L2 cache. All four cores share an exclusive 2-MB L3 cache. In addition, the chip contains one memory controller and one HyperTransport link which supports 8 GB memory (2 GB per core). Figure 1 depicts the Opteron architecture.

B. Seastar Router

Each node is connected to a Seastar router through the HyperTransport link. The router contains a DMA engine,
384 KB scratch memory, and a PowerPC 440 controller [4]. The DMA engine transports reads and writes to the memory controller across the hypertransport link. The 7-ported router provides one port to an AMD processor, and the other six to +x, -x, +y, -y, +z, and -z routing directions. In addition, each port provides four virtual channels to minimize packet blocking. A block diagram of the Seastar chip is shown in Figure 2.

C. Portals API

The Seastar chip is programmed through the Portals messaging interface. Portals provide one- and two-sided communication, which map coherently to MPI’s blocking and non-blocking send and receive. Portals was primarily designed for scalability, making it an attractive option for the 9,572-node Franklin supercomputer. It achieves scalability by requiring a minimal amount of state: a process is not required to establish a connection with another process, all message buffers are maintained in user space, and the target (receiving) process determines how to respond to incoming messages.

A Portals ‘put’ operation proceeds as follows: the initiator node fills a Memory Descripter (MD) with the data to be sent and constructs a put request [5], [6]. The data may be sent with the request, or in separate packets. The target node reads the request to determine the desired portals table (process ID), table index, and match bits. The table entry contains a pointer to a match list, which is sequentially searched until one is found that matches the match bits. If this procedure occurs without fault, the data is DMA’ed into the corresponding user-space buffer. If requested, the target MD’s corresponding Event Queue is updated with the details of the put event. Once the transaction is complete, an acknowledgment may optionally be returned to the initiator.

D. MPI on Portals

A number of parallel programming APIs, such as MPI, UPC, and Co-array Fortran, have been implemented on top of the Portals API. We initially considered UPC for our work, due to our familiarity with UPC, its open-source code, and recent interest in partitioned global address space (PGAS) languages. However, we soon learned that the CRAY’s robust performance analysis tool (CrayPAT) is not fully ported to the UPC API. In particular, we were unable to trace sent message statistics. With MPI, however, CrayPAT can trace message source, destination, size, and counts. This capability is crucial to our project, so we decided to use MPI. The downside to using MPI is that its Cray implementation is closed-source; as such, we are limited to technical reports and user guides to understand its implementation.

The Portals MPI implementation is based on MPICH, an open-source MPI implementation, which defines separate optimized behavior for short and long messages [7]. Each process-specific Portals table contains three entries: receive (for receiving messages), read (for unexpected long messages), ack (for receiving acknowledgments from synchronous sends), and some space for unexpected short messages. The match bits are configured to encode the send protocol, communicator, local rank (process ID, essentially), and MPI tag.

A short message is constructed in an initiator MD and set to respond to a single put operation. The message is immediately transferred to a matching buffer or (if unexpected) is buffered at the receiver. The send completes when the target EQ is updated. Synchronous sends receive an acknowledgment once the message reaches a matching receive MD. Long messages are handled in the same way as short, unless a matching receive isn’t posted. In this case, the data isn’t transferred and the sender must wait until the receiver requests the data.

When a process calls receive, the MPI library constructs a Memory Descriptor for the user buffer. Before posting a receive, you must search for the message in the unexpected message list.

E. Benchmarks

We chose the NAS Parallel Benchmark (NPB) suite to test the communication capabilities of Silence, because its applications exhibit a variety of different communication behavior.

F. Measurement Tools

The Silence system software packages include CrayPAT (Performance Analysis Tools), a package for instrumentation and tracing specific functions, including libraries such as MPI [8]. We are using CrayPAT specifically for its ability to profile message source, message destination, and message size per MPI send call (and each static send can be distinguished by its call stack). In addition, CrayPAT provides a convenient interface to AMD’s hardware performance counters. In order to measure cache accesses, we profiled L1 data cache reads, L2 reads, L3 reads, and L3 cache misses (to approximate main memory accesses). No memory controller performance counters are available for this platform.
III. Benchmark Analysis

In this section, we study the communication patterns and the memory hierarchy activity of 5 NAS benchmarks:

- MG: solves the poisson equation with a V-cycle multigrid method.
- IS: sorts small integers using the bucket sort.
- BT: solves a nonlinear partial differential equation (PDE) using a block tridiagonal solver kernel.
- LU: solves a nonlinear PDE using a lower-upper symmetric Gauss Seidel solver kernel.
- FT: solves a three-dimensional PDE using the Fast Fourier Transform.

These benchmarks exhibit memory, compute and network intensities. Using the performance analysis tool CrayPat, we explore the benchmark execution on Silence for different problem sizes and processor pools by accessing the hardware performance counters and the message traffic reports.

A. Communication Patterns

The benchmarks exhibit different communication patterns that are interesting and complete for our energy and architectural analysis. The IS benchmark exhibits a nearly balanced intra-chip communication structure. The LU and BT solvers reveal an unbalanced and complex inter-chip communication behavior (Figure 3).

B. Memory Hierarchy Activity

We present the memory hierarchy activity across the L1, L2, L3 caches and main memory for all benchmarks of small and large sizes parallelized with 4 and 64 processors respectively. Figure 4 reveals that MPI communication does not utilize more than 3% of the total memory accesses. Figure 5 shows that the memory accesses prescribed by the MPI communication are remarkable for large problems and large processor pools. DRAM accesses can reach up to 20% of the total benchmark DRAM accesses, as it is the case for FT. The dramatically increased L1 accesses due to MPI in BT, FT, and MG can be traced to a specific synchronizing barrier at the end of each benchmark. When processes finish computation earlier than the others, they spin on a synchronization in the L1 cache for billions of cycles. We believe that BT exhibits high L2 accesses due to cache thrashing. The IS benchmark L1 accesses are similarly inflated due to MPI synchronization functions, but not to the same extent. This study is a confirmation to our motivation thoughts.

IV. Memory Hierarchy Energy Analysis

In this section, we discuss our proposed method for analyzing the energy consumption across the memory hierarchy of Silence. We use the simulation tool CACTI 5.3 to model the energy consumption within each cache level. We use DRAMSim2 to model the energy consumption internally in the DRAM. Finally, we add an analytical model to our study to evaluate the energy consumption induced by the CPU to DRAM signaling.

A. Cache Energy

All cache levels of the machine Silence have a uniform cache access (UCA). They are manufactured using a 65 nm technology. They are built with 8 banks and a 64 bytes cache line. In addition, the L1 cache compromises 65536 bytes with 2-way associativity. The L2 cache is modeled with 512 Kbytes and a 16-way associativity. The L3 cache has the size of 2Mbytes and uses a 32-way associativity. The average energy consumption per read port for all cache levels are given in table I.

CACTI 5.3 assumes a single model of wires neglecting full-swing wires, which produce a significant power overhead. Figure 6 ([9]) shows that the energy produced by full-swing wires for a varying range of wire sizes does not exceed the pico Joules. The overall energy spent in the different cache levels is of the range of nano Joules. Therefore, in this case, the use of a single wire model does not
put/output termination signals (T). The energy consumed to DRAM can be categorized into input/output activation,  

$$E_{activation} = t_{activation} \times I_{active} \times V_{supply} \times f$$

and read/write operations. The DRAM operates at a frequency of 800 MHz, and the average termination power spent in the read pins is 2.665 mW. The energy consumed by these signals is given by:

- **$E_A$** = \(I_1 \times \frac{L}{t_2} \times V \times \frac{V^2}{2} \times l\), where \(I_1\) is the operating current, \(I_2\) is the active standby current, \(t_1\) is the row access strobe in memory cycles, \(t_2\) is the activation-to-activation minimal cycle, \(V_{supply}\) is the operating voltage, \(V\) is the maximum voltage supply of the device and \(l\) is the DRAM latency.

- **$E_W$** = \((I_3 - I_2) \times \frac{L}{t_2} \times V \times \frac{V^2}{2} \times l\), where \(I_3\) is the operating current for write operations, \(t_3\) is the life time of written data on the input/output pins in memory cycles, \(f\) is the processor frequency and \(I_{active}\) is the operating frequency.

- **$E_R$** = \((I_4 - I_2) \times \frac{L}{t_2} \times V \times \frac{V^2}{2} \times l\), where \(I_4\) is the operating current for read operations, \(t_4\) is the life time of read data on the input/output pins in memory cycles.

- **$E_T$** = \([p_1 \times (n_1 + 1) + p_2 \times (n_1) + \frac{L}{t_4} \times \frac{V}{2} \times 0.5 \times l]\), where \(p_1\) is the average termination power spent in a write pin, \(p_2\) is the average termination power spent in a read pin, \(n_1\) and \(n_2\) are the output/input pins.

Specifically, for the built-in DRAM:

- **$E_A$** = \([90mA \times \frac{1}{18}] \times 1.4V \times \frac{1.05^2 V^2}{1.4 V^2} \times 32.5 = 1.293825\) mJ,
- **$E_W$** = \([170mA \times 15mA \times \frac{1}{18}] \times 1.4V \times \frac{500MHz \times 1.05^2}{1.4 V^2} \times 32.5 = 2.665\) mJ,
- **$E_R$** = \([180mA \times 15mA \times \frac{1}{18}] \times 1.4V \times \frac{500MHz \times 1.05^2}{1.4 V^2} \times 32.5 = 2.9432\) mJ,
- **$E_T$** = \([8.2mW \times (214 + 1) \times \frac{1}{18}] + 1.1mW \times (214) \times \frac{18}{18} \times \frac{500MHz \times 1.05^2}{1.4 V^2} \times 0.5 \times 32.5 = 32.47\) mJ.

In total, the processor signals consumes an average of 36.5625 mJ between read and write signals.

C. DRAM Energy Model Sensitivity Analysis

For our energy analysis, we did not have access to the specification sheets of Cray XT4. We were missing the exact DRAM device width, the columns-to-rows partition and the life time of read and write data on the data pins.
Therefore, we have used reasonable guesses based on the machine geometry and implementation technology. In this section, we inspect the sensitivity of the DRAM energy model to variations of the estimated data.

- Device Width: We estimated the DRAM device width based on the machine size. We conduct our energy simulations for a range of device widths (x4, x8, x16, x32, etc.). Figure 7 shows that the DRAM average power consumption is very sensitive to DRAM device widths for small machines. Large device widths, and for large machines, the average DRAM power remains constant. Therefore, our DRAM energy model is insignificantly sensitive to the device width.

- Columns-to-rows partition: The columns-to-rows partition of the DRAM was undisclosed to us. We investigate the sensitivity of the DRAM energy model with varying partitions. Figure 8 reveals that the energy model is partition oblivious.

- Life time of read and write data on the pins: We decide to take conservative actions regarding modeling the life time of read and write data on the pins. We assume that the life time of data is equal to the activation-to-activation cycle from CPU to DRAM. Figure 9 shows that our assumption leads to the largest power consumption.

D. Benchmark Energy Activity

Using the memory energy models presented in sections A and B, we conduct an investigation into the energy consumption of the benchmarks, and specifically of the MPI communication, across the memory hierarchy. Figure 10 shows the energy consumption for a small problem parallelized with 4 processors. The energy consumption of the MPI functions across the memory hierarchy is negligible if compared with the overall energy consumption. This was predictable based on the memory hierarchy activity presented in section IV.

The total and MPI energy consumption of a large problem implemented with 64 processors are given in figure 11. In this case, the MPI functions consume a significant percentage of the total energy needed by the memory hierarchy. This class of problems is a good representative of the energy activity within the scope of high performance computers.

V. DRAM-AVOIDING MODIFICATIONS

We considered two methods to reduce the number of DRAM accesses due to inter-node communication: refill interception and cache injection. Refill interception exploits the fact that the core accesses main memory and router through a common interface, the System Request Interface (SRI). Cache injection expands the capability of I/O devices, allowing them to write directly into a core’s L1 data cache. This technique has higher energy reduction potential and requires fewer architectural modifications, so we restrict our analysis to only cache injection.

A. Cache Injection

Cache injection is the process by which an external device writes directly to a core’s cache. Standard coherence bus transactions do not typically support this behavior. However, most capabilities required for cache injection (bus snooping, address matching, cache modification) already exist in cache controllers. What is lacking is the ability to command a cache controller to listen for addresses besides those currently cached.
A.1 Implementation

We implement injection with a coherence bus command:

\[ \text{Ext} \langle \text{core} \rangle, \langle \text{cache line address} \rangle, \langle \text{data} \rangle \]

The “core” operand specifies the destination core L1 cache, “cache line address” is the address and “data” is the injected data. L1 caches snoop for a matching core ID when Ext_FILL goes on the bus; if found, they store the data and set the line to “modified” state, else they invalidate any cached copy. The required modifications to cache controller logic consist of one \( \log_2(n) \)-bit core ID comparator (where \( n \) is the number of on-chip hardware threads), one additional coherence command to decode, and the associated control logic.

The router needs to construct Ext_FILL coherence packets and pass them to the SRI. For simplicity, we propose sending the first full INJECTION_SIZE_LIMIT packets to the cache and DMA’ing remaining and partial lines to main memory. Caching a partial line may require one or more main memory accesses to locate an up-to-date copy of its unknown elements (either in cache or main memory) so we do not consider these for injection.

Injection also requires modifying the Portals API. The router needs to know the core ID for the receiving process to construct Ext_FILL packets. The per-process Portals table is an ideal structure to place a core ID, as long the ID is maintained across process migration. Also, the API must allow the user to set INJECTION_SIZE_LIMIT. We recommend an upper bound equal to the amount of cachable data, where any attempt to exceed this number will cause a non-fatal error.

A.2 Injection Size

What defines a “reasonable” message size? The injection of a reasonably-sized message will cause a net energy and/or performance benefit. Assume, for now, that message data is injected at the correct time - when it is immediately to be used in computation. A message greater than the total system cache size - L3 size (+ L2 size + L1 size, if...
exclusive) - will evict injected message data to main memory! Clearly a reasonable message size is within the total system cache size. If the L1 cache is a Harvard Architecture, restricting injections to the L1 size has the benefit of evicting data only (although instructions could be evicted in the L2 as a result of evicted L1 data), though the L1 (64 KB in our system) is typically quite smaller than the L2 and L3 caches. Allowing injections of L2 size and greater will cause instruction eviction, which should be avoided for performance reasons: an n-way pipeline fetches n instructions each cycle, while memory operations are unlikely to exceed 50% of all instructions. Standard programming constructs, such as loops and function calls, make instruc-

We profiled B class benchmarks compiled for 16 cores, shown in Figure 12. 64 KB messages fit within the L1, 576 KB messages fit within the L2 and L1, and 2624 KB messages fit within the cache hierarchy. Notice that all applications have at least 37.5% messages that fit within the L1. FT and IS have few overall messages, but many are extremely large. FT performs multiple all-to-all sends of (for B-class benchmarks) about 32 MB per message! LU and MG are much more likely to benefit from injection; each process sends many small messages, so there will be many communication-related main memory accesses and nearly all messages fit directly in the L1 (the rest in L2). For these benchmarks, we decided to use an INJECTION_SIZE_LIMIT of 1024, the number of L1 data cache lines. This is reflected in our analysis in section VI.

Cache injection has the potential to improve performance when invoked immediately or shortly before the data is used. This can alleviate the memory retrieval burden from the CPU, and reduced coherence bus occupancy caused by L2- and L3-miss-induced bus reads. Performance effects of cache injection were previously studied in [11], [12], [13]. This technique has higher energy reduction potential and requires fewer architectural modifications, so we restrict our analysis to only cache injection.

VI. Analysis

We performed our analysis as follows. Per dynamic receive call, the number of injected cache lines is defined as:

\[ CLI = \min(1024, \frac{\text{Receive Message Size (Bytes)}}{\text{Cache Line Size}}) \]

where 1024 is INJECTION_SIZE_LIMIT cache lines. We assume that injections are performed at the correct time, i.e. during MPI_RECV or MPI_WAIT, so that the effects of data displacement are equivalent to the baseline method of data retrieval. For larger messages and early injections, one must consider performance degradation due to displaced useful data/instructions.

Under these assumptions, the L1 data cache sees no change in access behavior. As with the baseline method, every data load results in a request sent to the L1 cache. The L2 and L3 caches receive fewer requests - every request for a cache line that is injected will not require a request to the L2 cache, and subsequently to the L3 cache. The
formula for change in L2 and L3 cache message requests is:

\[ \Delta \text{ accesses}_{L2/3} = -CLI \ast \left( \frac{1 \text{ Read}}{\text{Cache Line}} \right) \ast \text{numReceived} \]

The variable \text{numReceived} is the number of times this particular size of message is received by the process. For our benchmarks, nearly every call stack with \text{MPI\_SEND} at the head referred to a particular sized message, allowing us to equate \text{numReceived} with number of \text{MPI\_SEND}s. There were eight cases where a unique call stack had varying message sizes among processes - three in MG, one in BT, one in IS, and three in LU. In the worst case, the message size varied 7.6% from an average 7418.86 byte message. The effect is negligible - the maximum and minimum-sized messages differ by 2 cache lines. We assume the L1 cache is non-blocking with MSHRs that allow the controller to recognize multiple misses to the same block and submit a single request. Thus, we consider one read request per cache line.

The change in main memory accesses is defined similarly:

\[ \Delta \text{ accesses}_{mm} = -2 \ast CLI \ast \left( \frac{8 \text{ Accesses}}{\text{Cache Line}} \right) \ast \text{numReceived} \]

The 2 term is due to the write from I/O, then read from processor, of each cache line. With an assumed bus width of 64 bits, eight accesses are needed to read or write a full cache line.

VII. Results

In this section, we investigate the gain in energy consumption when using the cache injection memory technology. We use the cost analysis presented in section VI. While the L1 cache is heavily working, energy gains are expected to occur at the L2, L3 and DRAM levels. It is of interest to compare the energy consumption of the MPI functions across the memory hierarchy before and after the memory architectural modifications. For the benchmarks FT and IS, figures 13 and 14 show the energy consumption associated with the MPI calls with and without cache injection. The gain at the MPI level translates to a total energy gain. Larger problems benefit the most from cache injection.

Since the MPI workload is not substantial in the case of small problems, the total energy gain is bounded by 5 % (Figure 15). An energy gain of up to 47 % at the DRAM level and up to 7 % at the L3 cache level is recorded in the case of large problems (Figure 16). The LU benchmark benefits the most of the memory technology.

VIII. Related Work

Studying the power consumption of high performance computers was of interest for a number of researchers. Shalf and Kamil [14] have conducted in-situ power measurements at different machine subset levels for several high performance computational loads.

Cache injection is not a new technique to rapidly integrate data into ongoing computation. One approach to cache injection is an Injection Table [12]. The Injection Table extends the listened-for address set of its cache controller, at the user’s behest. The typical usage is producer-consumer code: the consumer process will fill the injection table with \textit{Open Window} commands, the producer process will broadcast data with a \textit{WriteSend} command, and the consumer process can close the address window with a \textit{CloseWindow} call. This work focuses on handling memory latency in shared memory multiprocessors for producer-consumer(s) relationships; similarly we explore a producer-consumer relationship, but focus on improving I/O data integration without adding significant cache controller logic or any ISA modifications (thus no application recompilations). This work does not consider the energy impact of the design.

A similar work [13] proposed producer-initiated remote-cache write mechanisms. Specifically, they propose \textit{WriteSend} for remote cache writes and \textit{WriteThrough} which writes through to memory (useful if the consumer is unknown). The authors found that remote writes reduced network traffic, reduced L2 cache miss rates, and produced lower cache miss latencies. The \textit{WriteSend} instruction is quite similar to \textit{ExtFill}, though \textit{WriteSend} is intended for core-to-core data transfer instead of I/O-to-core and does not consider the energy impact of the design.
IX. Future Work

Both the energy and cache access data were derived from simplified models of the physical computer system. As such, future work includes verifying our results through full system simulation, such as with VLSI CAD tools and program and memory traces for transistor switching activity. The experience of implementing our proposed modifications in a simulated system is necessary to detect complex memory and core interactions that our models do not account for.

While cache injection appears to be an attractive mechanism for energy reduction, there are a number of possible improvements available. In order to work with inclusive cache hierarchies (such as Intel processors), the Ext_Fill command must write data into inclusive caches as well as the L1. This will use more energy than the exclusive version, due to the L2 and L3 activity, but could still produce overall energy reductions by avoiding DRAM.

In order to support producer-consumers computation, Ext_Fill requires the ability to simultaneously write multiple L1 caches. This could be accomplished with a bit mask, assuming no more than 16-32 local cores, multiple ID specifiers, or with an injection table-style cache-resident configuration. This technique may benefit parallel computation on I/O data, such as a video or audio stream, by directly feeding it to each involved core’s cache. (FIXME) do this section

X. Conclusion

In order to solve the energy problem in HPC, new techniques and optimizations must be made at all levels of the computing abstraction. We predicted the energy behavior of the memory hierarchy using the open-source simulators CACTI and DRAMSim2, from which we derived estimates of energy per access of each memory component. Memory hierarchy activity was gathered from hardware performance counters on our test system, Silence, a single-cabinet 96-node machine.

We have discussed a microarchitectural optimization, cache injection, which can reduce DRAM energy consumption by up to 47% according to the energy model. This technique works best when programs send large numbers...
of small messages, which with standard I/O methods can cause many DRAM accesses for only a few cache lines. I/O data injection exploits the fact that we can expect received messages to be used soon after reception, so buffering them in main memory is likely not the best option.

This is one in a number of possible communication optimizations. Most of the future work remains in extending Ext_Fill, and exploring other energy-reducing communication hardware.

XI. Acknowledgments

We’d like to thank the NERSC scientists, especially John Shalf and David Skinner, who provided us with support, insight, and accounts for their supercomputers. We’d also like to thank our professor, John Kubiatowicz, who provided valuable insight and guidance.

REFERENCES


