MAESTRO: Dynamic Runtime Power and Concurrency Adaptation

Allan Porterfield
Renaissance Computing Institute
100 Europa Drive, Suite 540
Chapel Hill NC
akp@renci.org

Rob Fowler
Renaissance Computing Institute
100 Europa Drive, Suite 540
Chapel Hill NC
rjf@renci.org

Mark Neyer
Renaissance Computing Institute
100 Europa Drive, Suite 540
Chapel Hill NC
neyer@renci.org

ABSTRACT

Microprocessor core counts are on a trajectory such that for many applications the microprocessor will soon be over-provisioned for computation with respect to off-chip communication and memory system capabilities. In the absence of strategies to exploit intra-chip communication, performance of even simple benchmarks is noticeably impacted by co-scheduling multiple copies on the cores of current system and the problem will get worse.

We are developing the MAESTRO runtime system to experiment with the idea of dedicating excess computational resources to (1) to reduce power consumption of memory bound applications with little performance impact, (2) provide guidance to adaptive concurrency control to improve performance, and (3) isolate the performance monitoring and other system activities to reduce the effects of "jitter" on tightly-coupled applications. We describe initial experiments in which we dedicate one core to monitor chip-wide bottlenecks using hardware performance counters. When bottlenecks in the memory system are approached, the frequency (and power) of the cores are reduced. The MAESTRO prototype runs on an AMD Phenom as a Linux daemon pinned to one core while applications are pinned to the remaining cores. On a desktop system, running at low frequency can save up to 36% of the total power consumed with minor performance degredation. MAESTRO allows single core jobs to compute at full frequency and saves power when the bottlenecks exist during parallel execution.

We present motivating experiments, describe aspects of the working prototype of MAESTRO, and present some early results.

1. INTRODUCTION

Microprocessor designers have used 'Moore's Law' doubling of transistors on a chip to greatly increase application performance over the past several decades. This has been accomplished by increasing the clock rate and adding complexity to the computational cores to increasing instruction level parallelism. Recently, Intel, AMD, IBM, and Sun, among others, have started to replicate the computational cores on a chip without increasing the speed of the cores. Almost everyone agrees, the era of faster clock rates has passed and that we've entered a new age of dramatically increasing core counts.

To increase performance, applications will be required to exploit parallelism. Multiple instruction streams will be required, each concurrently accessing distinct memory locations. The cores are being replicated, but the memory subsystem is not, nor is the amount of parallelism within memory subsystems keeping pace with the increase in core counts. Although improvements in the memory system (bigger caches, faster clocks, DDR architecture improvements, etc.) are occurring, to take advantage of these improvements will require intense use of contiguous blocks of memory. For many applications the locality required by the memory subsystem will either not exist or will require major performance tuning to exploit. The likely effect is that, for most applications, the per core effective memory bandwidth will be dropping.

Another trend is the emergence of heterogeneous processor chips. The STI Cell Broadband Engine has two different core types within a single processor; this increases the pressure on the runtime to match application requirements to available hardware resources. Other vendors have announced plans for heterogeneous chips. Effective scheduling of work to a variety of cores within a system requires a new degree of interaction between the application, the OS and the hardware. As the computation is increasingly limited by memory performance, runtimes will be called on to do more to improve performance.

We are designing MAESTRO to take advantage of the trend in multi-core chips to over-provision computation by allocating dedicated resources to the runtime. The resources can be used to improve many features of program execution. By removing the OS and the runtime from the application computational resources, jitter effects should be minimized. Initially, the focus has been on decreasing power consumption without reducing performance. Using the same techniques

in the future, heterogeneous scheduling of FPGAs and GPGPUs can be handled, substantially reducing the complexity of writing applications that use those devices. Performance introspection can allow a system designer to monitor performance of a production system to design better systems for the future. An application can use introspection to monitor performance and adjust itself to decrease either execution time or execution cost.

The next section presents some preliminary performance benchmark results for a Intel and a AMD quad-core system. We explore the AMD system further by exploring its power utilization for several clock rates. The design of MAESTRO, a new runtime designed to aggressively use multi-core processors to optimize performance and operational costs, is then introduced. Finally, some very early power performance results using a MAESTRO prototype are presented.

2. QUAD-CORE PERFORMANCE

Current commodity microprocessors are expanding their core counts and, for some applications, are beginning to exhibit slowdowns because of memory bottlenecks. The performance of two quad-core systems running very simple memory-intensive benchmarks is examined. As the number of active threads/cores increases, the contention for shared resources visibly affects performance of these systems.

2.1 Intel Quad-Core Results

The first quad core system examined was a Dell 2950 server with dual sockets each containing a 2.33 GHz Intel 5300 (Clovertown) processor. It was running a Linux 2.6.9 kernel with the Perfmon 2 [6] extensions. The Clovertown has caches shared between pairs of cores and ships all memory references through a single front-side bus to an off-chip memory controller. All of the runs reported had processes pinned to particular hardware resources with the `taskset` command. Other than the benchmarks, the system was idle during the tests.

One of the best known memory benchmarks is STREAM[16], which measures effective bandwidth memory bandwidth on simple, optimizable loops. We used an unmodified copy of version 5.8 of the benchmark compiled with gcc (version 4.2.3) and options -O3 and -fopenmp and used the default array size of 2 million elements. All numbers reported are for the Triad operation.

As the placement and number of threads are varied, levels where the replication of resources helps performance and where the contention for shared resources hurts performance become noticeable. In Figure 1, it is clear that adding a second thread using a single cache has little effect on performance (2.83GB/sec to 2.88GB/sec, about 1.5% increase). However, having two threads use different caches on the same chip greatly increases effective bandwidth (4.62GB/sec, 60.4% improvement). Only an additional 1.6% increase occurs when all four threads on a single chip are used. If we instead use two threads with caches on different processors, contention (probably for the bus to memory) becomes obvious as performance falls to 3.18GB/sec with less than a 1% improvement for four threads and only two caches. Four threads, each with their own cache, improve performance to 5.46GB/sec (18% better than 1 chip 2 cache). Finally, having all eight threads active improve another 1.4% to 5.53GB/sec.

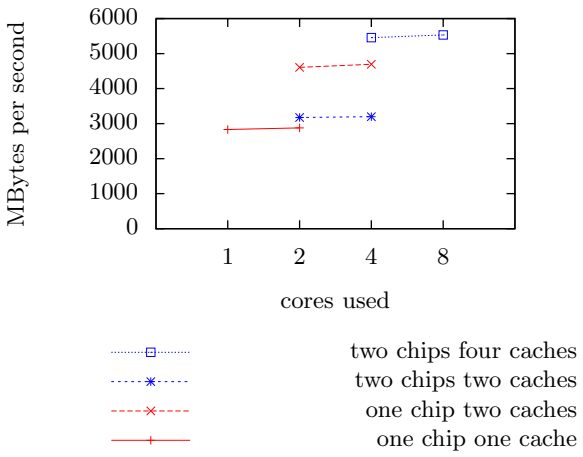

Figure 1: Intel Clovertown Stream Results

These results show that one thread can use all the bandwidth available from a cache. The memory can support twice the bandwidth offered by one thread, but various contention and concurrency issues require having all 4 caches busy to reach peak bandwidth. Minor contention exists within a chip between caches, but when arbitrating between chips contention becomes significant. This contention impacts core numbering and how the Linux scheduler assigns processes to cores.

Figure 2 shows the relative time to complete multiple copies of several of the NAS parallel benchmarks (version 3.2 class B). The fastest observed single core time for each benchmark is used as a baseline to compare the slowdown as more copies are introduced. Only one chip was used in each test and all combinations of the four cores within that chip were tested. The range shown for each test is the difference between the fastest and slowest copy in any of the tests. For 1 and 4 copies the times were relatively consistent. When running 2 and 3 copies, the results were bi-modal depending on whether a cache was being shared. Sharing a cache resulted in longer completion times. The penalty for sharing a cache ranged from 13% for BT to 77% for LU. When using four cores (sharing 2 caches) on one chip, performance was slightly slower than when one cache was being shared for the 2 and 3 core tests. The additional slowdown probably results from contention for memory/memory buses as load on the memory increases, increasing memory latency slightly.

For simple HPC and memory benchmarks, the memory system is not able to support full-speed execution of the 4 cores sharing various portions of the memory system. This slowdown results from various bottlenecks and contention along the path to the memory. As the core count increases with each microprocessor, this will become more visible to the application programmer.

2.2 AMD Quad-Core Results

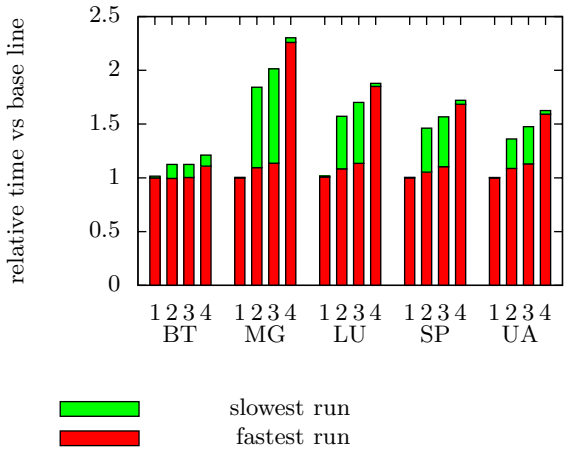

Figure 2: Selected Intel Clovertown NAS Parallel Benchmark Results

The same tests were run on a HP Pavilion system with a 2.3 AMD Phenom processor and 4GB of main memory consisting of 4 1GB DDR2-800 DIMMs. The AMD system was running Ubutu LINUX with kernel 2.6.23 with the perfmon2 extensions. The AMD part has an on-chip memory controller. In addition to per core L1 and L2 caches, the Phenom has a 2MB L3 cache shared by all four cores.

Results from running the STREAM benchmark are shown in Figure 3. Looking at the 2.3GHz (full frequency) numbers, going from 1 to 2 threads increased performance significantly (32%) and adding a third thread increased performance slightly (4.5%). Peak bandwidth occurred with only 3 threads and performance fell by 1.1% when the fourth thread and its memory streams were added. A memory bottleneck is, basicaly, being hit with only two threads, and the system becomes unstable in that adding threads eventurally hurts throughput.

Looking at the AMD `DRAM_ACCESSES` counter, the number of DIMM accesses varies from 3.60 billion for one stream to 3.75 billion for 4 streams. The increase in `PAGE_CONFLICTS`, the most expensive operation, is even greater; one thread had only 0.21 billion conflicts, two threads had 0.84 billion, three threads had 1.08 billion and four threads had 1.17 billion. When we look at the rate of `DRAM_ACCESSES`, we see that three threads averaged 104 million accesses a second, 30 million of which were conflicts. Four threads had slightly fewer accesses per second, 102 million accesses, but 32 million each second are conflicts. Without locality in the DIMM pages, the data rate supplied by memory effectively peaks well below the theoretic limit.

Examining the AMD Phenom performance on the NAS benchmarks class B Figure 4, per process performance decreases consistently as more copies of a benchmark are added. In the case of LU, the slowdown from 1 to 4 was 81%, MG 77% and SP 51%. Notice that the AMD results were consistent for 2 and 3 copies, since no partially shared cache exists.

The shared L3 cache results in more consistent performance

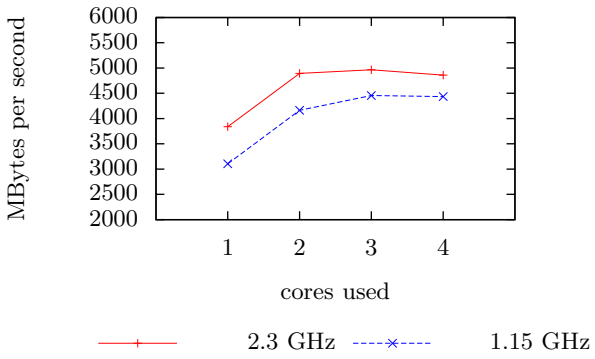

Figure 3: 1.15 GHz AMD Phenom Stream Results

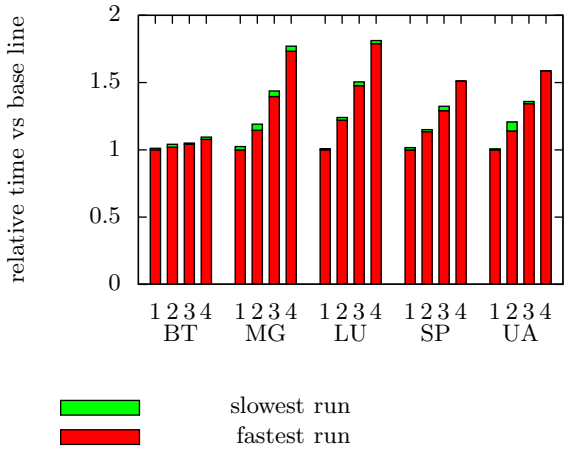

Figure 4: Selected AMD Phenom NAS Parallel Benchmark Results

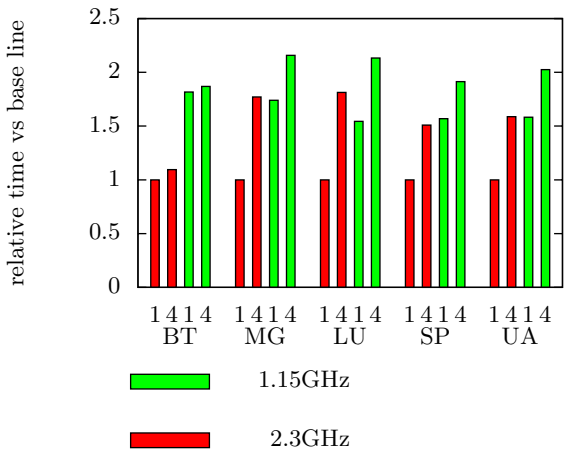

Figure 5: Selected 1.15 GHz AMD Phenom NAS Parallel Benchmark Results

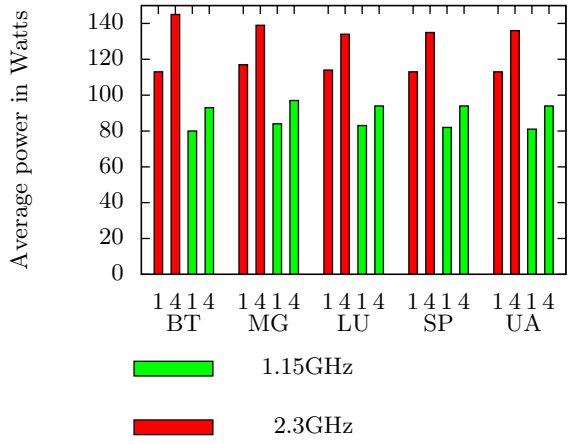

Figure 6: Power Consumption for AMD Phenom NAS Parallel Benchmark

versus the Intel part as problems are scaled. We can see performance effects from access to various shared resources in the memory system (either L3 access/capacity or memory bank contention). These effects are already significant for quad core processors. As core count rises faster than memory parallelism increases the effect will continue.

Since the performance of the AMD Phenom processor was tailing off as contention for memory rose, the tests were repeated with the processor running at 1/2 speed. The ability to reduce frequency and power usage is available on most current microprocessors as a power-saving option for battery-powered systems such as laptops. For this system (Phenom and BIOS), the only reduced frequency option was 1.15GHz; often more selections are possible. The question is: For HPC applications, how much performance is lost by reducing frequency (and voltage) and how much power is saved? Figures 3 and 5 compare the performance of half and full frequency for the previous benchmarks (only 1 and 4 core numbers given for the NAS Parallel Benchmarks).

2.3 AMD 1/2 Frequency Quad-Core Results

For single-threaded STREAM Triad, performance fell 20.3% but in the four thread case it fell by only 9.7%. For the NAS benchmarks performance of BT, LU and MG fell significantly. As copies were added, the performance of the 1/2-frequency version better matching the full speed version (BT 1 thread - 45.0% 4 threads - 41.9%, MG 1 - 42.5% 4 - 18.2%, LU 1 - 35.3% 4 - 15.0%, SP 1 - 36.3% 4 - 21.2%, UA 1 - 36.8% 4 - 21.0%). Extrapolating these numbers, the low-frequency versions for LU and MG could match the full-frequency version with as few as 8 or 10 cores.

As the number of active cores increased, the performance difference between 1/2 speed and full speed declined. When we look at the power consumed, Figure 6, shows that power savings increases as the chip becomes busier. For these measurements, we averaged over time the power entering the system. We included power for the processor, memories, disk drives, networking, power loss in the power supply, etc. By observing total system power including more than the microprocessor, we found that the power saved by the total system running at a lower frequency can be quantified.

When no jobs are running at 2.3GHz, the system uses 78 watts and at 1.115GHz, it falls to 70 watts, just over a 10% reduction. Running just one copy of the NPB averages 114 watts at full speed. This is reduced to 83 watts by halving the clock rate, a savings of 27%. When the processor load is increased to one copy per core, the savings increases to 42 watts or 31%. For individual benchmarks (BT), the savings were as high as 36%.

The power savings rise, absolutely and relatively, as core usage increases. The performance slowdown drops relatively as core usage increases. As core counts increase, there is an opportunity for the runtime to take advantage of the proximity of these two lines, to reduce power while maintaining performance.

3. RUNTIME ADAPTATION OPTIONS

As the computational power of microprocessors stresses and exceeds the memory system's ability to supply data, runtimes will be required to do more to match application requirements to hardware resources. Runtimes need not be limited to hardware scheduling, but they can also adjust the hardware resources to match application needs better. As the previous results in section 2.3 showed, different applications will use shared hardware resources differently, but many of them will have significant idle computational resources as core counts increase. The runtime can use the excess to monitor and control the hardware and the applications jointly.

In 2006, data centers used 1.5% of the total US electrical consumption[20] and this use is projected to double again in 5 years. As core counts increase, some applications will run as quickly at lower processor frequencies and voltage as at peak. Our results show that the power savings from a lower frequency can be significant. Just running at lower frequency may not reduce power, since a large fixed power cost exists. Increased application performance often reduces to-

tal power consumption even with the power cost of a higher frequency. The runtime needs to detects dynamic performance and then to adjust frequency and voltage according to the situation.

Often applications can run at lower power by running on less than all of the computational units. AMD Quad-core chips have a shared cache and Intel is expected to introduce a shared cache in the fourth quarter of 2008 on the eight-core Nehalem. As the effective cache per core shrinks (more threads using it), the cache hit rate may fall dramatically, causing longer latencies and running into a bottleneck getting data from off-chip memories. In such cases improved time to completion and optimal power use is acheived by running some of the cores at full speed while almost turning off the others. For this to be a viable strategy, the runtime must be prepared for the set of available hardware resources to change over a program's execution. This affects not only thread scheduling but any synchronization primitives that the program may be using.

When the runtime changes the power for some cores on a processor, it is making that system heterogeneous. Cores may progress at different rates even though they are identical hardware. Development of a runtime that supports heterogeneous speed cores is an early step towards runtimes that can support heterogeneous hardware resources such as moving some of the computation to special-purpose devices like GPGPUs, FPGAs, or the IBM CELL chip.

4. MAESTRO OVERVIEW

We are designing MAESTRO to take advantage of the current trends in hardware design to minimize operational costs while providing maximum application performance. In particular we are addressing the gap between power and parallelism in processors versus memory systems. Multiple parallel threads tend to access memory in patterns hard for a memory controller and DRAM to access with low latency and high throughput particularly when fairness is a concern[17].

Since we believe that future systems will be overprovesioned with computation for most applications, MAESTRO dedicates one core (R-core) to the runtime. As core counts increase, this will become a minimal overhead. The R-core is responsible for watching dynamic system performance and adjusting either the hardware or the application to use the available resources better. Hardware performance events for shared resources (AMD L3 cache, DRAM events, Intel's Front Side Bus, etc.) to determine when a hardware bottleneck is occurring. An interface to the Operating System can allow transmission to the runtime information about the whole system's performance, including load, I/O etc. to be transmitted. MAESTRO is designed to run within an applications address space and communicate directly with the application and thread scheduling. As this interface is developed, MAESTRO can make informed decisions about parallelism, performance and power levels.

When MAESTRO detects the memory is saturated, i.e., that the application is generating memory requests faster than the memory can handle (detected by high L3 miss rates or DRAM Page Accesses rate), it reduces the pressure on memory either by slowing all the cores or by turning off a subset of cores. Which is preferable is application dependent. If a slightly larger cache would result in a significantly higher cache hit rate, reducing the thread count actually increases application performance by reducing offered load to memory. A single copy of a simple program that repeatedly touches 500,000 values runs in about 23 seconds, two copies take 69 seconds and three copies 100 seconds. Reducing threads would improve perfomance and reduce power. When the threads are latency-bound, slowing their issue rate to match memory speed to the processor speed results in minimal performance loss. Both methods significantly reduce both processor and total system power uses for some applications.

Dynamically changing processor clock rates allows MAESTRO to turn a homogeneous system into a heterogeneous one. This requires applications to use dynamic threading package that can run on top of the changing hardware. The multiple cores can all share a single address space, work queues and other structures are shared. Before execution starts, a thread is a small "continuation" with a small data footprint. Once execution has started, a loose binding to a core is formed by data footprints in private caches, so the size and expense of moving a thread grows substantially. Synchronization methods between pairs and groups of threads are provided. The programming interface[19] provided by MAESTRO is intended to be a target for other tools and models. Rather than inventing yet another programming language/model, MAESTRO attempts to support other models efficiently and to provide a level of virtualization between the model and the actual hardware being used.

Light-weight threads with little state can be used to emulate heavier-weight threads such as those in OpenMP[18]. A simple implementation under MAESTRO is to run one persistent thread per core and to dispatch OpenMP work on demand. OpenMP synchronization calls can then be mapped to MAESTRO synchronization in a straightforward manner. An implementation where the number software threads needs to equal hardware thread count reduces MAESTRO's ability to shut down cores, since a missing thread could have significant impact on synchronization. An implementation where OpenMP threads can be multiplexed would be substantially more complicated but would allow the processor count to exceed the actual hardware resources, providing MAESTRO more opportunities to optimize hardware resources without sacrificing performance.

Distributed programming models such as MPI[7, 8] present different problems and opportunities. Since memory is shared between the R-core and each MPI process, most asynchronous communication can be quickly pushed to the R-core, removing most of the overhead of processing of the MPI call from the worker thread. For at least some applications, this additional overlap between computation and communication should improve overall performance.

Like OpenMP, most MPI applications assume a fixed number of threads throughout execution. Because of this, most implementations do not virtualize threads and cannot deal with dynamically variable hardware resources. AMPI[12] is the best known exception. It, however, places restric-

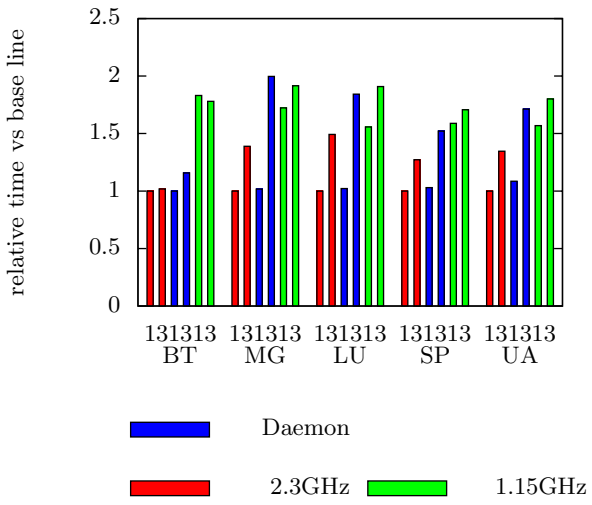

Figure 7: Selected NPB Performance with Power Daemon

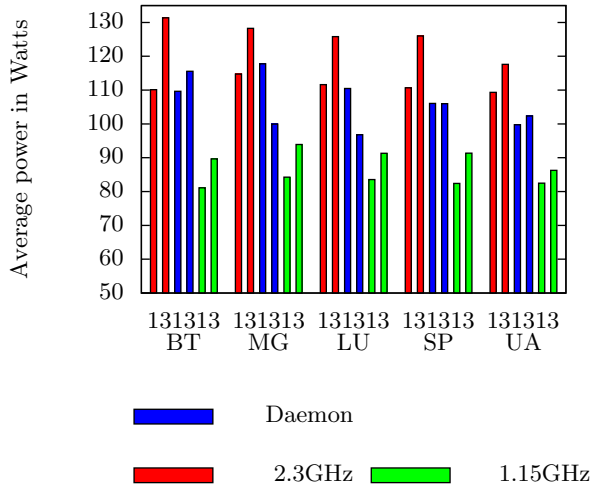

Figure 8: Power Used for Selected NPB Benchmarks

tions on address-space usage to allow threads to be migrated across processors. Because we will be running in a shared address space between processors, MAESTRO will be able to support AMPI-like virtualization without the memory allocation restrictions. By supporting aggressive communication offloading and virtualization, a MAESTRO framework should allow implementation of a fast, flexible platform for MPI applications.

5. MEASTRO INITIAL RESULTS

A MAESTRO API [19] threading interface has been created that incorporates facilities to incorporate performance introspection data and to dynamically adapt both hardware state and application scheduling strategies. It can be used directly, but normal use is expected to be via output from another threading tool, e.g. a compiler. The library is being prototyped and a simple program has executed using it. However, many scheduling and synchronization issues are still in the design evaluation phase.

We are also working with a prototype monitoring framework. Currently, the L3 cache-miss ratio and DRAM events are the metrics of choice on an AMD Phenom processor. The monitor thread uses the `libpfm` interface to acquire the counter values. We are considering counter multiplexing to overcome the limited number of counters. When the cache-miss ratio is high, the function uses the `libcpufreq` interface to change cache frequency. This currently requires that the thread be run with root permission.

Testing of the R-core thread before the library was complete required us to implement a standalone daemon that could run independently of any particular application. The daemon incorporates the R-core functionality, but has no direct interaction with the applications. The previous tests were rerun with one core executing the daemon (figures 7 and 8). In all tests, the applications were locked to cores 1-3. Core 0 was either idle or running the daemon and was running at 1.15GHz at all times, including during the high power tests, to provide better estimate actual power savings. Core 0 was chosen for the daemon because that is where most of the OS executes, and the Phenom processor seems to be symmetric over cores 1-3, but favors core 0. Removing core 0 reduces the likelihood that an application will need to overcome 'jitter' because of a fast or slow core.

For these initial experiments, the daemon was set to watch the L3 cache-miss rate. The number of cache misses was examined about every 10000 clock ticks of the 1.15GHz core 0. The processor frequency is raised when the misses exceed .055 for 5 periods, with no intervening period below .01. The periods do not need to be consecutive. The frequency is lowered if 5 periods have a rate below .01 but not greater than .55. This relatively simple mechanism is sufficient hysteresis to avoid overfrequent changes in frequency. The minimum time between switching of only about 50 millisecs allows the system to respond very quickly to changing performance and so far does not produce any noticable overhead (beside that of the dedicated core).

Even with this simplistic model, behavior acheived is close to that desired. When a single core is active, the power mostly stays high and the performance is close to the full-power version (within 2% except for UA at 8.5%). Power savings were minor (9.5% for UA and 4.3% for SP and -2.6% for MG). When 3 copies of the tests with the daemon are run, the performance is mostly between the fixed cases. The total power consumed is much lower than in the 2.3GHz test, but is consistently higher than the 1.115GHz test. The power draw is not consistent across the test. Although the average power is reduced, the instantaneous high power required is the same with the 2.3GHz test and the power daemon. The average power used when the daemon is present is roughly the same whether one or three jobs are being run. If this is not a function of the benchmarks being used, it could have some implications for the total power needed by large data centers deploying power features like this in the future.

6. RELATED WORK

Most previous power conservation work has been for embedded devices and portable computers. Those devices have strict conservation goals because of limited available battery life. Hardware performance counters have been used to estimate energy consumption of static and dynamic CPU, as well as memory energy estimation. Our goals differ, since we are only attempting to limit operational costs. If performance is high, high power usage is acceptable (i.e. cost efficient).

Bircher et al.[23] looked at the correlation between IPC and other counters and power dissipation for selected SPEC benchmarks. They found that IPC was a good predictor above .5 and could be improved with L1 miss rate and branch mispredictor rates when IPC was below .5. In [3] Bircher et al. built a power/energy model based on fetched micro-ops per cycle and further improved it with a two-input trace cache model. Li et al.[14] used IPC and simulation to model operating system power. Each OS routine requires its own model and will not be appropriate for user applications where functions used change frequently. Bellosa[2] used synthetic workloads to demonstrate a correlation between performance counters and power dissipation. Isci et al.[13] used 22 performance counters to monitor each region of the microprocessor. This produced results less accurate than ours and cannot be done dynamically on current systems. They do give the programmer and system architect a better understanding of where the power is going. Contreras and Martonosi[4] also use an off-line modeling technique driven with a large number of performance metrics. The focus of all of this work was to estimate power use in order to limit peak requirements for batteries and cooling. Our concern is to limit power use without causing performance degradation.

For embedded systems, Gurun and Kritz [9] use clock cycles and data stalls to model runtime power and continually refine it by checking against the battery-monitoring unit. Their goal of optimizing embedded system battery life differs from our goal of reducing operational costs of HPC programs by reducing energy usage. Their use of hardware to improve the fit dynamically often is not available on HPC systems.

The power work that is HPC-specific has dealt with mechanisms to use dynamic voltage and frequency-scaling to limit power without reducing application performance. Hsu and Feng[10] introduce the β-algorithm to determine phases in which power can be reduced. Earlier, Hsu and Kremer[11] looked into methods to incorporate profiling data into a compiler to locate a region where power could be reduced with minimal performance impact. MAESTRO can use phase detection software techniques like these to improve runtime frequency decisions. Since the R-core is separate from the application, there is no need to limit opportunities for checking utilization. As the application interface is completed, we will be examining the increased power optimization opportunities.

Performance prediction research has used performance counters to estimate and to decrease execution time. Most of that work has focused on off-line algorithms. For instance, Eeckhout et al.[5] made IPC predictions based on short code samples, and Yang et al.[24], partial execution-based prediction. Settle et al.[22] ADORE system and Lu et al.[15] are early examples of dynamic optimization based on performance counters, but neither system is directed towards power management.

7. CONCLUSIONS

Achieving good performance on a multi-core processor is going to be a challenge. Shared resources outside of the processor core but within the processor chip are going to have major performance implications as parallelization between cores becomes common. The major performance bottleneck will continue to move from the cores to the memory system. In a system over-provisioned with cores, there is an opportunity to separate the sources of OS jitter from application computations and to allow the runtime system to claim some computational resources directly rather than competing with the application on shared cores. We have designed MAESTRO to be a framework for exploring the options in this design space.

The initial MAESTRO implementation is focused on reducing power without reducing performance. We have a framework that allows us to use one core to monitor one shared resource and to change system frequency. MAESTRO needs extension to look at multiple shared resources and to make better use of the counters to detect profitable power frequency opportunities. Connecting the daemon thread with the application will allow us to modify not only the hardware but the application to utilize available resources better.

Although our initial focus for MAESTRO is adaptation to constraints induced by contention for external resources, the long-term intention is to turn our focus on introspection and adaptation for on chip bottlenecks. The combination of hardware performance counters and application knowledge will allow better thread scheduling and power management decisions to be made. As the MAESTRO framework is completed, exploration of issues relating to the presence of heterogeneity (either in performance or ISA) will be explored.

8. REFERENCES

[1] *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA*. IEEE Computer Society, 2005.

[2] Frank Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *Proceedings of the ACM SIGOPS European Workshop, Kolding, Denmark, September 17-20, 2000*, pages 37–42. ACM, 2000.

[3] W. L. Bircher, M. Valluri, J. Law, and L. K. John. Runtime identification of microprocessor energy saving opportunities. In Roy and Tiwari [21], pages 275–280.

[4] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale processors using performance monitoring unit events. In Roy and Tiwari [21], pages 221–226.

[5] Lieven Eeckhout, Sébastien Nussbaum, James E. Smith, and Koen De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[6] Stephane Eranian. Perfmon2: a flexible performance monitoring interface for linux. In *Linux Symposium 2006, Ottawa, Canada*, July 2006.

[7] Message Passing Forum. Mpi: A message passing interface,

1993.

[8] Message Passing Forum. Mpi-2: Extensions to the message-passing interface, 1996.

[9] Selim Gurun and Chandra Krintz. A run-time, feedback-based energy estimation model for embedded devices. In Reinaldo A. Bergamaschi and Kiyoung Choi, editors, *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2006, Seoul, Korea, October 22-25, 2006*, pages 28–33. ACM, 2006.

[10] Chung-Hsing Hsu and Wu chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA* [1], page 1.

[11] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 38–48. ACM, 2003.

[12] Chao Huang, Orion Sky Lawlor, and Laxmikant V. Kalé. Adaptive mpi. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing, 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003, Revised Papers*, volume 2958 of *Lecture Notes in Computer Science*, pages 306–322. Springer, 2003.

[13] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*, pages 93–104. ACM/IEEE, 2003.

[14] Tao Li and Lizy Kurian John. Run-time modeling and estimation of operating system power consumption. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2003, June 9-14, 2003, San Diego, CA, USA*, pages 160–171. ACM, 2003.

[15] J. Lu, H. Chen, P. Yew, and W. Hsu. Design and implementation of a lightweight dynamic optimization system. *The Journal of Instruction-Level Parallelism*, 6, 2004.

[16] John D. McCalphin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December, 1995.

[17] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of the 16th USENIX Security Symposium (USENIX SECURITY), Boston, MA*, pages 257–274, August 2007.

[18] OpenMP Architecture Review Board. OpenMP application program interface version 2.5, May 2005.

[19] Allan Porterfield. Maestro:program thread and sychronization interface, version 0.1. Technical Report RENCI Technical Report TR-08-01, Renaissance Computing Institute, 2008.

[20] Energy STAR Program. Epa report on server and data center energy efficiency, 2007.

[21] Kaushik Roy and Vivek Tiwari, editors. *Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005, San Diego, California, USA, August 8-10, 2005*. ACM, 2005.

[22] Alex Settle, Joshua L. Kihm, Andrew Janiszewski, and Daniel A. Connors. Architectural support for enhanced smt job scheduling. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*, pages 63–73. IEEE Computer Society, 2004.

[23] W. L. Bircher and Jason Law and Madhavi Valluri, and Lizy K. John. Effective Use of Performance Monitoring Counters for Run-Time Prediction of Power. Technical Report TR-041104-01, Electrical and Computer Engineering Dept., University of Texas, Austin TX, November 2004.

[24] Leo T. Yang, Xiaosong Ma, and Frank Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA* [1], page 40.