Effects of Multi-core Memory Concurrency Limits on Multi-threaded Applications

TR-10-03

Anirban Mandal, Min Yeol Lim, Allan Porterfield, Rob Fowler

September 2, 2010

RENCI Technical Report Series
http://www.renci.org/techreports

Effects of Multi-core Memory Concurrency Limits on Multi-threaded Applications

Anirban Mandal Min Yeol Lim Allan Porterfield
Rob Fowler
Renaissance Computing Institute
100 Europa Drive, Suite 540
Chapel Hill NC
akp,rjf,anirban,mylim@renci.org

September 2, 2010

Abstract

Memory access is becoming an increasingly significant impediment to extracting performance out of multi-core systems. More than ever, the effectiveness of memory system use by an application is becoming a critical determinant of performance. In previous work, we demonstrated how explicit consideration of memory concurrency provides a better model for memory performance on multi-socket, multi-core systems than just using best case latency and bandwidth. This paper investigates some of the implications of this on application structure and compiler optimization. We developed a methodology to use hardware performance counters in a performance reflection tool, *RCRTool*, to measure achieved memory concurrency. We applied this to several important memory-bound scientific applications and kernels compiled with varying levels of optimization. We convolve the observed application concurrency with available system memory concurrency to derive insights for compilers and application tuners. The models provide compilers and runtimes with information about how load on the memory sub-system changes the effectiveness of various optimizations. As the number of hardware cores/threads increases, and as off-chip memory bandwidth per core remains constant or decreases, these measurements and analysis can provide insights to compiler and application writers. For example, on highly-threaded systems the system can be saturated if each software thread offers only 2 or 3 concurrent memory references. The implication is that optimizations to improve cache usage are more important than ever, while program transformations designed to increase memory concurrency may lose their utility.

1 Introduction

Recent generations of computing systems have achieved high memory bandwidth through the mechanism of supporting highly concurrent, overlapping access to the memory sub-system. In prior work [8], we presented

a benchmarking methodology for measuring the concurrency that systems can actually deliver as well as bounds on that concurrency at the core, socket, and system level. In this study, we measure the concurrency that multi-threaded applications actually consume. We present a hardware counter-based measurement methodology and apply it to a set of kernel and application benchmarks. We conclude by discussing the implications on the future of compiler optimization and application tuning.

Scientific applications are known to be sensitive to memory sub-system performance. Williams *et al.* [24, 25] observed that while Intel quad-core chips (5300-series and 5400-series) had greater peak floating-point capacity than contemporary AMD Opteron NUMA systems, the AMD performed better on many benchmarks because memory was more scalable. Furthermore, adding memory hardware [25] on AMDs expanded memory-concurrency potential. With the 5500-series processors, Intel added a scalable memory system and both vendors have now moved to DDR3 memory parts.

Most characterization methods for memory performance use two measures, memory latency and bandwidth as measured using benchmarks like LMBench [11] and STREAM [10]. Models often treat these as fundamental scalar properties of the system. Common system memory performance characterizations, such as [14, 12, 4, 6] use these values as the basis of their models. In reality, both measures depend on a set of complex factors and can vary greatly depending on offered load within a system.

These measures are only part of the story for multi-socket multi-core architectures. There are multiple points at which requests saturate different levels of the memory hierarchy. Each saturation point occurs with different numbers of concurrent accesses from all the threads sharing that hardware resource. Accurate performance models must account for these constraints if they are to prove useful for guiding future programming and optimization strategies.

Because bandwidth is achieved through concurrency and because latencies vary due to queuing and scheduling, we argue that concurrency, rather than latency and bandwidth, is the fundamental quantity for modeling. The degree of concurrency to memory that a system can serve efficiently is largely a function of higher-level system design and depends on several factors, including the number of cache misses each core can tolerate, the number of memory controllers, the number of concurrent operations supported by each controller, memory communication channel design, and the number and design of each of the memory components. As more threads share the limited resources, it takes fewer memory references from each thread to saturate the shared components.

Constrained available memory concurrency per thread affects the utility of many loop based optimizations. Optimizations such as unrolling, which increases the number of memory references that can be processed in parallel, are greatly reduced in value. In a world of severe bandwidth and concurrency constraints, prefetching, both software and hardware, may be counter-productive. Optimizations such as tiling to increase data reuse will be more important than ever. We show that, for some programs, lower optimization levels (-O2) perform as well or better than higher levels.

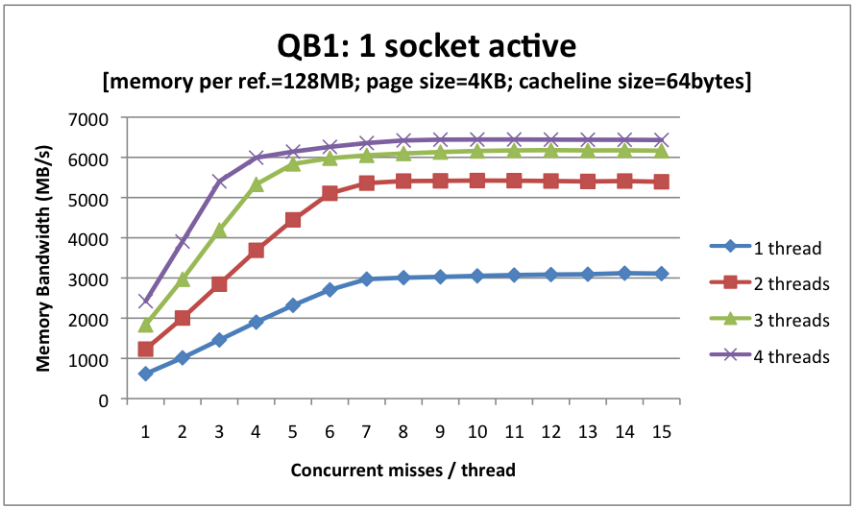


Figure 1: Memory Bandwidth vs. Concurrent Misses (QB1)

2 Memory Concurrency

The offered concurrency among various memory operations (loads and stores) by the application and the ability of the system to handle that concurrency are fundamental determinants of memory performance for multi-core systems. Non-blocking caches are crucial to attaining high memory bandwidth because programs can issue multiple loads and stores that are overlapped with each other and with other classes of instruction. The *achieved concurrency* depends on both the ability of a program to generate an *offered concurrency* load and on *concurrency capacity* at various levels (core, socket, board) levels of a system.

The offered concurrency is the aggregate of the offered concurrency of all of the threads running on the core, socket, or board. On real hardware, achieved concurrency may match the offered concurrency at low levels of load. At high loads, though, the achieved concurrency is determined by the capacity constraints. In cases in which no one thread, or small set of threads, is capable of saturating the system, the aggregate load of many threads may be needed to fully utilize the resources. Compiler optimizations, program tuning, and runtime scheduling activities that focus solely on single-thread performance will not address this issue. It is vital that we understand system-wide properties, both of the hardware and of applications.

In previous work [8], we used pChase [13, 7] to identify system capacity limits for memory concurrency on multi-socket, multi-core systems. pChase is a benchmark designed to test memory throughput at different levels of the memory hierarchy under carefully controlled degrees of concurrent access. There are multiple threads, each of which executes a loop with a controllable number of independent "pointer chasing" operations per iteration. We used pChase in a mode in which the sequence of pointer addresses is pseudo-random and designed to defeat hardware prefetching while limiting TLB misses. A wrapper script around pChase is used to iterate over different numbers of memory reference chains (thereby controlling memory concurrency) and threads for each pChase run. For example, the results from running pChase on one socket of a quad-socket AMD Opteron system (QB1) are shown in Figure 1. With one core active and a single outstanding cache miss,

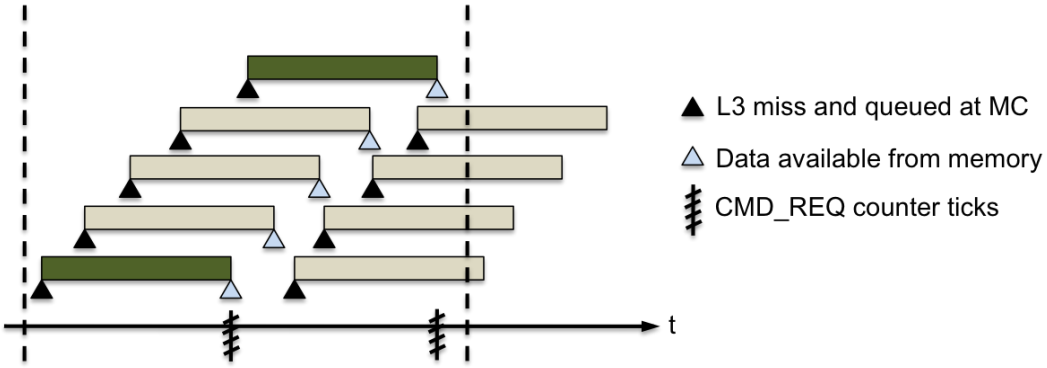


Figure 2: Example: Tracking of memory requests using counters

memory bandwidth performance is about 618MB/sec. Each additional concurrent miss increases bandwidth by about 400MB/sec until a peak bandwidth of 3.0GB/sec with 7 outstanding references is reached. Any number of additional concurrent misses increases bandwidth by a total of less than 100MB/sec. For small numbers of concurrent misses per core (1-3), increasing the number of cores raised aggregate bandwidth nearly linearly (3.92 maximum speedup for 4 cores). As the number of concurrent misses per core increases (> 6) the speedup falls to between 2.06 and 2.13. This second bottleneck is a chip-wide limit that is more restrictive than the per-core limit. This chip limitation only appears when multiple cores are on a chip and should be manifest in models of memory used for tuning and optimization.

3 Methodology

In this section, we describe how we estimate the the dynamic achieved memory concurrency of an application running on a given system.

3.1 Deriving Achieved Memory Concurrency

AMD Opteron systems have a hardware performance monitor (HPM) device that can be used to measure hundreds of different architectural events and their costs. We use the following three events that are shared across all of the cores on a particular processor chip:

- `L3_CACHE_MISSES:ALL`
- `CPU_READ_COMMAND_REQUESTS_TO_TARGET_NODE_0_3:ALL (or CMD_REQ)`
- `CPU_READ_COMMAND_LATENCY_TO_TARGET_NODE_0_3:ALL (or CMD_LAT)`

L3_CACHE_MISSES:ALL counts the total number of level 3 cache misses from all cores in a particular socket. This is a major component of the total set of operations that go to DRAM. The CMD_LAT and and CMD_REQ events are designed to work together to compute the average latency of memory read requests. CMD_LAT event aggregates the number of clocks between issuing and satisfying memory requests. Only one request is monitored at a time, so the CMD_REQ event is used to count the number of events that were monitored. Other memory requests that are concurrent with a monitored request are not counted. CMD_REQ is incremented only when the memory request that is being tracked by the CMD_LAT counter gets satisfied. It ticks again when the next tracked memory request gets satisfied and so on. Another way of thinking about CMD_REQ is that it is both the number of requests that were served in non-overlapping intervals and it is the number of time intervals in which one or more requests were outstanding.

For example, in Figure 2, if the first memory request (the left-most slab in dark shading) is monitored, the other three requests that overlap with it won't be tracked, but the next non-overlapping request will be. In this example, the CMD_REQ counter will tick twice in the shown observation period.

The L3_CACHE_MISSES:ALL event is a measure of the total number of memory requests. The mean number of requests in each of the non-overlapping intervals over an observation period is thus the ratio of L3_CACHE_MISSES:ALL to CMD_REQ events. Thus, in Figure 2 the achieved concurrency in the observation period would be calculated as 4.5. In section 5, we demonstrate good agreement between results using this estimate of achieved concurrency and results of pChase runs. Specifically, this experimental model matches the pChase results in both the low load (achieved concurrency equals offered concurrency) and high load (achieved concurrency equals system constraints) regimes.[1]

3.2 Resource Centric Performance Reflection

Given the potential importance of bottlenecks induced by the full utilization of shared resources in multi-core, multi-socket systems, we are developing performance measurement and analysis tools that focus on these resources rather than the traditional "first-person" approach which measures each thread's activity independently. It is our intent to make this information available in real time to help applications and system code introspectively adapt to resource constraints. We use "Resource-Centric Reflection" (RCR) to describe this approach and have implemented a prototype RCRTool for experimental use.

Figure 3 presents the structure of RCRTool. Major components are an RCR daemon, RCR meters, and an RCR viewer. The RCRTool daemon accesses hardware and OS performance counters, applies a variety of models such as the achieved concurrency model described above, and posts model outputs in a set of performance meters. Hardware performance monitoring is done via the Perf_Events interface available in recent Linux kernels. The RCR meters module implements an interface to make model outputs available to the OS and applications. Using the Linux debugfs, the information is kept in a hierarchical structure corresponding to the underlying hardware (thread, core, socket, system) topology. The RCR viewer is a user interface for on- and off-line graphical display.

[1]For the Intel Core i7 systems, the counters for the shared memory controllers are kept in the shared "uncore" portion of the system. Currently the Linux perf_events HPM driver does not support access to these counters and events. When "uncore" counters become available, we will explore suitable methods for the Intel chips.

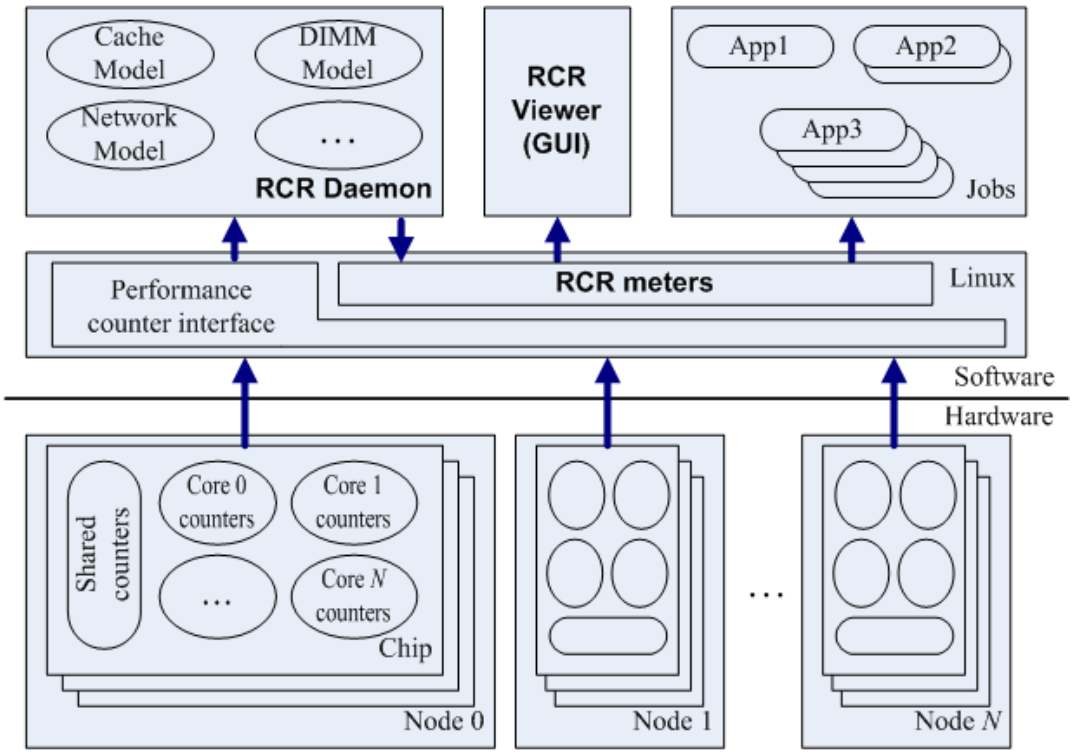


Figure 3: RCRTool Overview

In addition to being used to evaluate applications and systems, real time feedback from RCRTool on achieved concurrency levels can be used to improve both application and system performance. If the runtime detects that memory concurrency is saturated, performance may be improved by reducing the number of threads and giving each thread a larger fraction of the shared resources (cache). The OS by co-scheduling applications that have different memory characteristics can complete a series of jobs quicker. Inside a parallel region, compiler optimizations to increase memory concurrency become much less important and optimizations to increase memory reuse and increase ALU utilization become more important.

4 Benchmarks

We examined the achieved concurrency for a variety of kernel and application benchmarks.

pChase and STREAM: pChase [13, 7] is described above. STREAM [9] is the well-known benchmark. They are used to validate the methodology.

NAS Parallel Benchmarks - BT and CG: We used two codes, each with a significant number of memory requests, from the NAS Parallel Benchmarks suite (version 3.2.1). BT solves a synthetic system of nonlinear partial differential equations with a block tridiagonal solver. CG estimates the smallest eigenvalue of a large, sparse, symmetric positive definite matrix using inverse iteration with a conjugate gradient method [1]. Both used class B size.

FermiLab Quantum Chromodynamics (QCD) Benchmark: The FermiLab QCD benchmark is a suite of several application benchmarks used to evaluate new systems for their suitability for running Lattice Quantum Chromodynamics codes [17]. It is made up of the following benchmarks: (a) STREAM (OpenMP version, with restricted compiler options), (b) a MILC test, (c) a Clover inverter using Chroma, and (d) a domain wall Fermion (DWF) test using Chroma. The MILC, Clover, and DWF applications are built with mpich2-1.2. The suite supports multi-core systems with 8, 12, 16, 24, 32, and 48 cores. Each of the lattice QCD tests is run with a test problem size appropriate to the core counts. The memory required per core is approximately 500 MB.

Lattice-Boltzmann Magnetohydrodynamics (LBMHD): LBMHD is a Lattice Boltzmann Magnetohydrodynamics benchmark that models homogeneous isotropic turbulence in dissipative magnetohydrodynamics (MHD). We used a version of the LBMHD code obtained from Lawrence Berkeley National Lab and as described by Williams *et al.* in [23]. The LBMHD code is known to have tremendous memory capacity requirements and moves large amounts of data for each point in the lattice space.

5 Experiments: Validation and Benchmarks

Several AMD multi-socket systems were used to measure achieved concurrency of the memory bound kernels and applications described in section 4. **WS1** is a dual-socket Dell PowerEdge T605 with two 2.1GHz AMD Opteron (Barcelona) processors, *i.e.,* eight cores in all. It runs Ubuntu Linux with the 2.6.33.3 kernel. The memory slots are fully populated with eight 2GB (16GB total) dual-rank DDR2/667 memory sticks. **QB1** is a quad-socket Dell PowerEdge M905 blade with four 2.2GHz AMD Opteron (Barcelona) processors, a total of 16 cores. It runs CentOS Linux with a 2.6.35-rc3 kernel. QB1 has 24 dual-rank 2GB DDR2/667 memory sticks for a total of 48GB.

5.1 Model Validation using pChase and Stream:

We used pChase and Stream runs to validate the HPM counter-based model. Figure 4 shows the results of a pChase experiment on QB1 in which a script launched a sequence of 15 separate jobs, with the number of reference chains per thread increasing from 1 to 15. The left graph shows typical results for one thread, on one core of Socket 3 in this case. The right graph shows results for a run with 4 threads on (all on Socket 0). The X-axis is the wall clock time and the Y-axis is the memory concurrency as estimated using the RCRTool memory concurrency model. In both figures, the other sockets are mostly idle and this is reflected in the values reported for them.

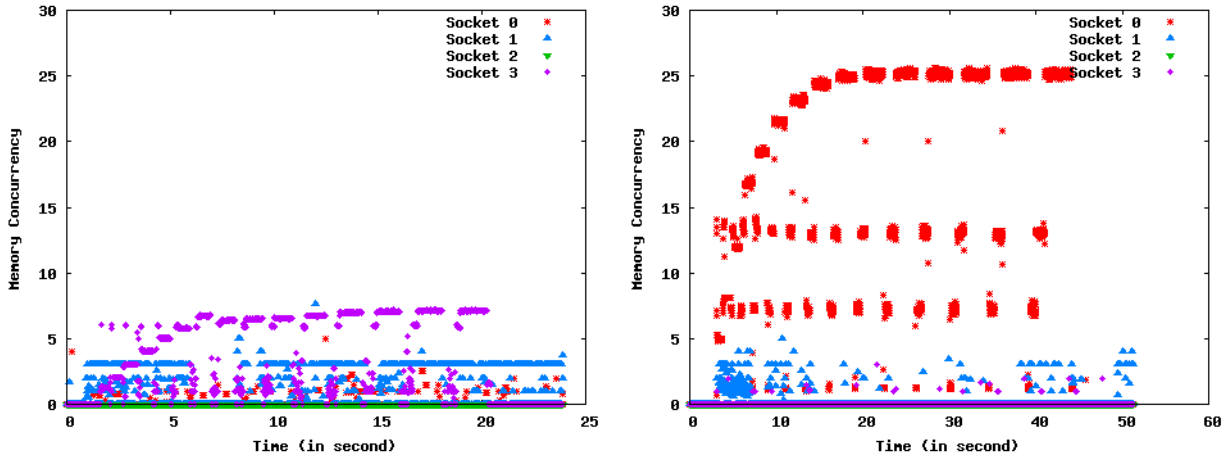


Figure 4: Memory Concurrency for pChase benchmark run (1 thread and 4 threads)

For each test in the sequence there is a short initialization phase followed by a bandwidth-intensive "pointer chasing" phase. Memory concurrency is moderate in the initialization phases, serving as "punctuation" separating the actual tests.

The maximum estimated memory concurrency attained by one thread is 7 and this asymptote is first reached by the pChase run that generates 7 concurrent references. This agrees with the the pChase results illustrated in Figure 1 in section 2. The asymptotic memory concurrency value when executing pChase with 4 threads is around 25. The asymptotic value and the placement of the inflection point is in close agreement with Figure 1. Given the close agreement between runtime measurement and model results with the carefully controlled workload generated by pChase , we have enough confidence in the methodology to use it to examine application benchmarks. We are currently exploring methods for characterizing estimation error and looking at "corner cases".

A further validation test was to run the OpenMP version of Stream on QB1. This achieved an achieved memory concurrency with values between 23 and 25 and bandwidths consistent with the pChase results.[2]

5.2 NAS Parallel Benchmarks - BT and CG:

Figure 5 shows results from a run of the NAS BT benchmark on all cores of Socket 0 on QB1. The scatter plot on the left shows achieved memory concurrency for the entire duration of the application. Each iteration has several phases with distinct levels of concurrency. These manifest themselves as bands in the scatter plot. The data is presented as a histogram in the right sub-figure. Although the achieved memory concurrency is

[2]There is a related counter, CPU_REQUESTS_TO_TARGET_NODE_0_3_4_7, which is supposed to count both reads and writes. However, that counter always returns zero. We believe that the CMD_REQ counter counts both reads and writes, as is evident from results for STREAM, which issues both reads and writes but still has a maximum achieved concurrency bound of 25.

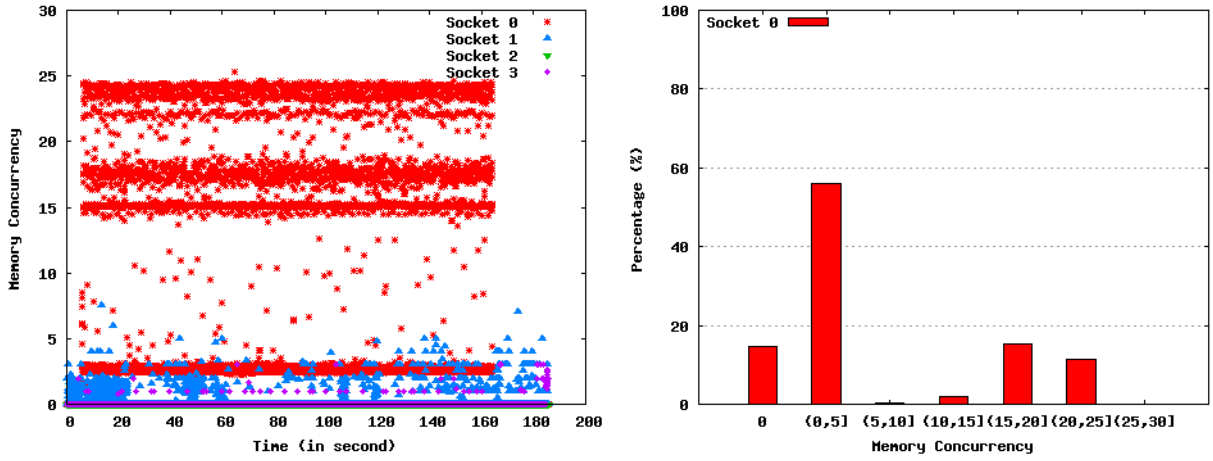


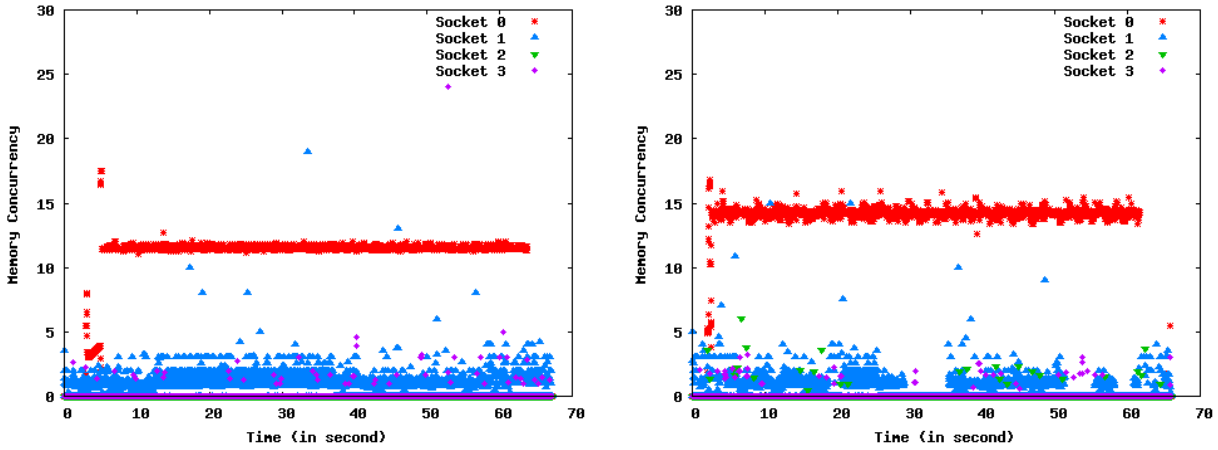
Figure 5: Memory Concurrency for NAS BT Benchmark

relatively low (<5) about 75% of the time, it reaches the socket-wide limit of 25 about 10% of the time and is above 15 about 24% of the time. Even when the average memory bandwidth/concurrency is moderate over an entire application, variability during the run can result in a memory bottleneck for a significant fraction of the time.

For this version of NAS BT, more aggressive optimization to increase the offered memory load in the regions of high concurrency would not decrease execution time. It would be beneficial to explore transformations that level out the load by moving memory references to unsaturated phases, or by "diluting" the memory load by interleaving arithmetic operations with the high-intensity regions. These goals might be achieved by applying whole-program optimizations, or by co-scheduling arithmetic-intensive threads with memory intensive ones.

We performed similar experiments with the NAS CG benchmark on all cores of one socket of the QB1 system. For the OpenMP version in figure 6(a), memory concurrency is very close to 11 for the entire duration. On a full socket, CG uses less than half the available bandwidth. The memory constraint is latency and lack of concurrency. This is a case where more concurrency should be exposed either though more aggressive optimization, either at the source-level, in the compiler or through additional hardware threads.

We recompiled the OpenMP version of CG using the Intel (11.1, -O3), PGI (10.4, -O3), and with GCC (4.5.0, -O3) compilers. There were no substantive memory behavior differences between those versions and the experiment described above. We also compiled the MPI version of CG using all three compilers. For GCC and PGI, the results are both qualitatively and quantitatively similar to the OpenMP case. The Intel compiler (Figure 6(b)), however, was able to push the average concurrency level up to 14.2 and achieved an overall 13% speedup over the other cases.



(a) gcc OpenMP version

(b) icc MPI version

Figure 6: Memory Concurrency for NAS CG Benchmark

5.3 Lattice-Boltzmann Magnetohydrodynamics (LBMHD):

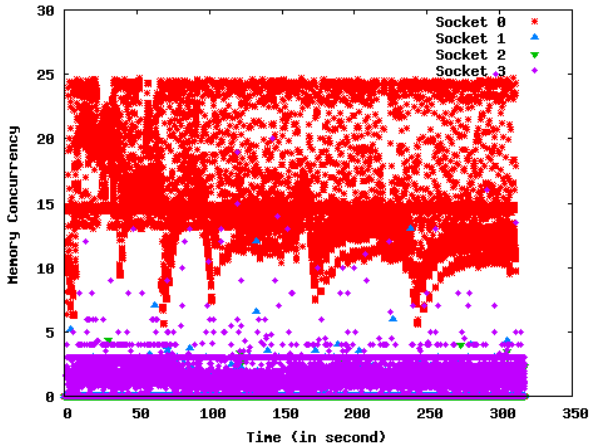
Figure 7 presents results from running the LBMHD benchmark on running all cores of Socket 0 of QB1 compiled with different levels of optimization using gcc. The first is with aggressive optimization, including AMD Opteron architecture- and model-specific flags plus loop unrolling and -O3. The second version is compiled using -O2 (no system specific or explicit unrolling). The highly optimized version actually takes about 2% longer to run.[3] The overall memory concurrency patterns are similar in both cases. The -O2 version generates more memory references in the same time and therefore has visibly more memory concurrency points between 15 and 20 than the -O3 version. Several intermediate sets of compile options were tried; the visible difference occurs when the -march and -mtune options are added. The reduction in the number of references that reach memory can be explained by better use of the various cache levels in the higher optimization levels. However, -O3 also reduces the overlap of the remaining memory references resulting in a slight increase in execution time. For larger numbers of cores, the reduction in offered memory concurrency may allow continued scaling and be an overall win.

5.4 FermiLab QCD:

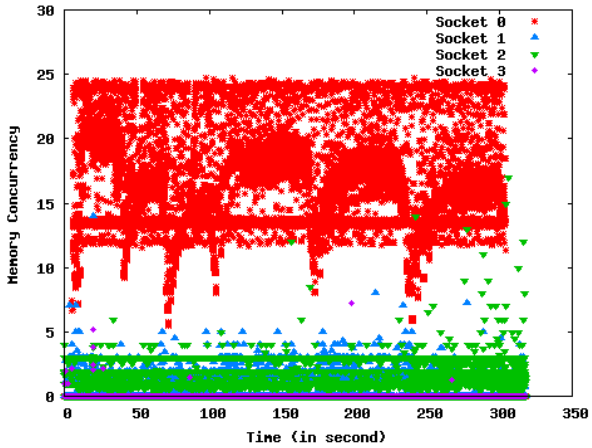
The Fermi QCD suite runs three benchmarks, the first is STREAM, discussed earlier. We examine individual iterations of each of the other applications. The Fermi QCD suite was run on all cores of the WS1 system.

Figure 8 shows several iterations of the MILC QCD code, "su3_rmd", during the time interval from 1000 sec. to 1100 sec. The histogram shows the memory concurrency distribution for one of the iterations (1040 sec. to 1080 sec.). The memory concurrency varies evenly between 5 and 20 throughout the iteration.

[3]RCRTool graphs are started and stopped independently of a particular execution, so the origin of the time scale is not the start of the experiment.



(a) O3+ optimization (307 sec)



(b) O2 optimization (298 sec)

Figure 7: Memory Concurrency for LBMHD Benchmark

Although, the memory concurrency is below the maximum available, there is limited headroom. Further optimization (software or hardware) to increase memory concurrency have limited room for improvement.

Several phases of the "chroma" QCD code are shown in Figure 9. They span the time interval from 2500 sec. to 2600 sec. in the scatter diagram. Over 90% of the time slices, as shown by the histogram for one phase (2535 sec. to 2590 sec.), have memory concurrency below 15, with the average well below 10. Even with the overall low memory concurrency shown, a number of time slices are at or near the system limit. Memory concurrency currently does not limit performance, but scaling could be limited by the high concurrency bands.

5.5 Sampling Effects

In RCRTool, the measurement interval of each metric can be dynamically changed to adapt to application patterns and overhead constraints. The choice of sampling interval depends on how the memory access pattern of an application fluctuates over time. If the fluctuations are minimal, longer intervals are sufficient, but many programs need to use shorter intervals to show the true dynamic nature of memory usage.

All of the experimental results presented above used a 5 millisecond sampling interval. Figure 10 shows the effect of changing sampling intervals on memory concurrency measurements. Two experiments were run using the same version of LBMHD, one with 100ms sampling and one with 5ms sampling. If one uses the larger sample size on the left, achieved memory concurrency falls in narrow bands and it never approaches the system limit. When a smaller sample size, 5ms, is used, it becomes clear that many short code segments within an iteration are hitting the system limit.

While using the finer sampling interval helps to identify phases, it may also hurt performance. The

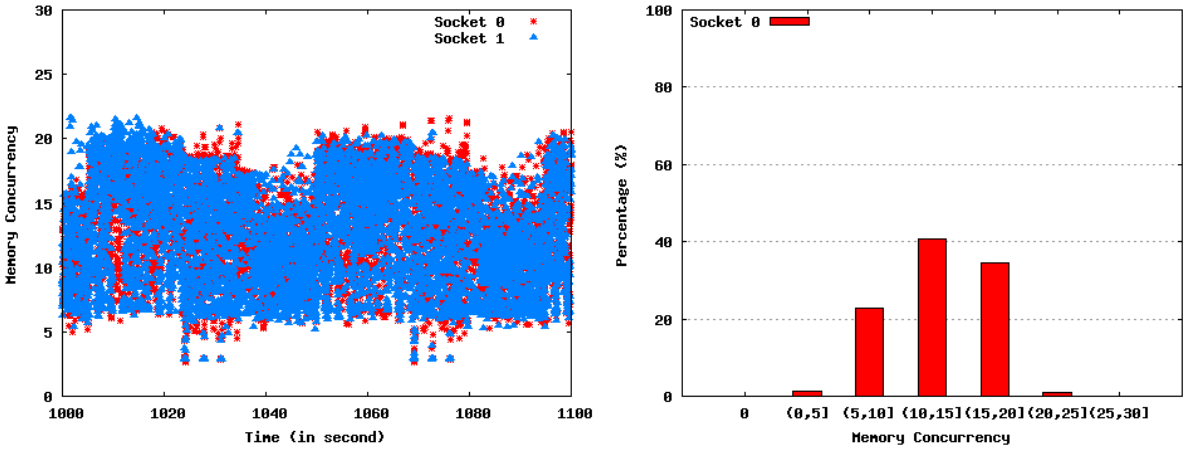


Figure 8: Memory Concurrency for "su3_rmd" / MILC

current prototype implementation of RCRTool uses a daemon to do the sampling as well as to pass the data on to the RCRViewer in an on-line visualization mode. There is no performance penalty if the system has an otherwise idle core on which to run the daemon. If the application uses all the cores, running the RCRTool daemon slows that application down by between 1% (100ms) and 20% (5ms), depending on the sampling interval and application workload.

6 Related Work

Snavely *et al.* [19, 3] use linear models for system capabilities based on unit-stride and "random" memory bandwidths numbers and characterizes applications in terms of demand for each. They compute predicted performance by taking an inner product of the two vectors. This works well only in a domain where linear relationships hold. Multi-core systems however exhibit non-linear characteristics with bottlenecks at various system levels. Hence, as we discussed in [8], explicitly introducing a memory model based on concurrency results in modeling a unified continuum of memory behavior. This work focuses on deriving application's observed memory concurrency and convolving it with the system concurrency models to infer about memory performance.

Williams [22] has pointed out that with the addition of cores to a chip or system the offered memory concurrency will increase rapidly, an insight which we have used. Tikir *et al.* [20] have used a genetic algorithm approach to predict memory bandwidth of HPC applications as a function of cache-hit rates by "learning" the bandwidth using memory bandwidth benchmarks.

When comparing the performance of UPC versus Co-array Fortran versions of an application, Coarfa *et al.* [5] showed the importance of exposing memory concurrency to the compiler. In one case, recoding to expose memory concurrency in a UPC sparse-matrix program increased performance by a factor of 2.3 even with the same memory-access pattern.

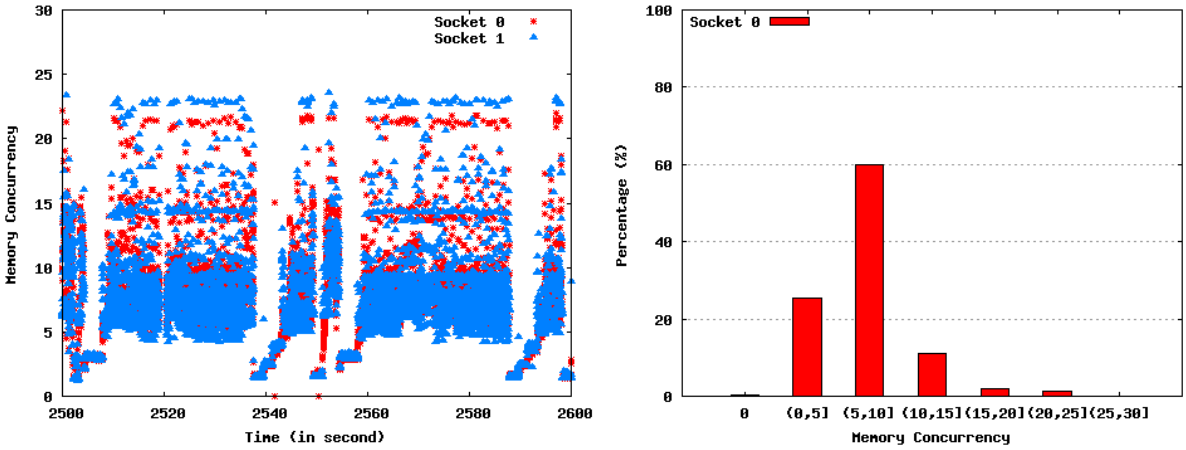


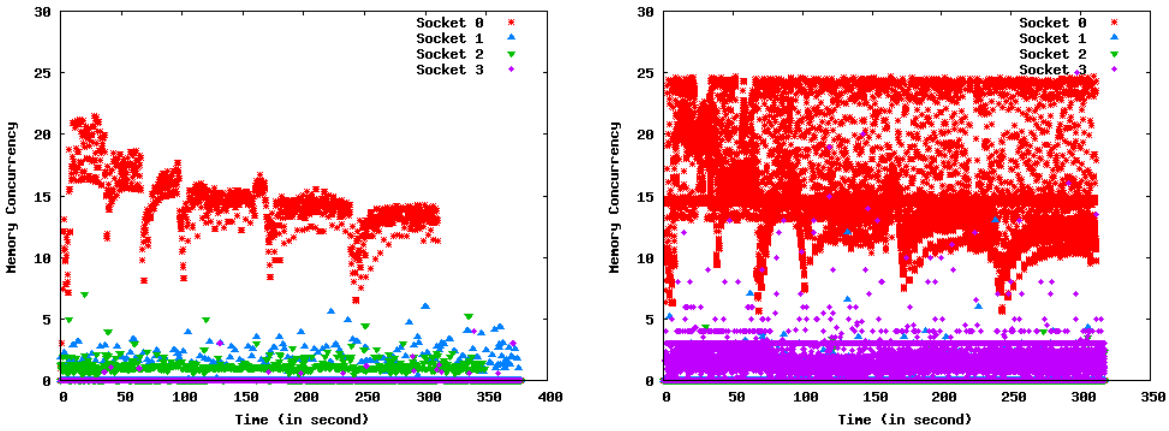
Figure 9: Memory Concurrency for "chroma"

Most existing performance tools [21, 2, 18, 16] are based on a "first person" view of performance. HPM counters are virtualized on a per-thread or process basis to attribute architectural events and their resulting costs to individual threads of control and fragments of application code executed by those threads. This model makes sense when applications are single-threaded, when the goal is to change an application's source code, and when the programmer did not control, or even know the details of, the other processes running on the hardware. In current multi-core, multi-socket environments, especially for high performance computing applications, a node of a system runs one application at a time and that overall performance depends upon how the processes and threads of that application cooperate or compete with one another. In a single-threaded environment, a thread that consumes 30% of the available memory bandwidth will not stress the system, but any attempt to run more than 3 threads will be constrained. RCRTool focuses on a "resource centric" approach to performance measurement and modeling and on making that information available for runtime introspection ("reflection") to guide adaptation at all levels, from the choice of application methods to cooperative runtime thread scheduling.

7 Conclusions

The use of multi-core systems as the primary means of increased system performance is here to stay. Increases in application performance will result from increased parallelism and increased parallel efficiency, not from increased clock rate. This will radically increase the number of threads and outstanding memory references. Memory subsystems, while improving, are not keeping up.

We have shown that scientific codes already generate enough memory concurrency in some execution phases to saturate the memory capacity with a small number of threads. As the number of cores and of hardware thread contexts increases, the number of outstanding references per thread needed to induce saturation has already decreased to the range of one to three. This trend is reducing the effectiveness of high-level optimizations designed to achieve high effective bandwidth by overlapping memory operations,



(a) LBMHD 100ms

(b) LBMHD 5ms

Figure 10: Comparison of sampling intervals

but this will continue to increase the importance of cache performance for those programs that can use it. One effect that we see already is that reduced levels of optimization are becoming competitive.

The pChase benchmark is designed to defeat the effectiveness hardware prefetchers as well as of compiler optimizations. As per-thread concurrency and bandwidth become increasingly precious resources, sharing those resources with prefetchers may be ill-advised. We are planning on investigating this.

As memory and other shared resources begin to dominate performance, a new generation of tools that focus on these resources will be necessary. Our efforts with RCRTool are one step in that direction. As stated above, RCRTool is intended to provide feedback to runtime schedulers. If the achieved memory concurrency load is high, the scheduler may choose to run at a lower level of concurrency, thus simultaneously decreasing the number of threads generating memory load while increasing the per-thread shares of caches. We are exploring this strategy in the MAESTRO project [15].

Individual application tuners, who want the best performance out of their codes, need to look not only at single thread performance, but also at how jobs are laid out on the system and at co-scheduling within and between applications. Applications with different shared resource usage patterns may be able to run without interference. Learning how to maximize shared resources within and between applications is required. This will require a coordinated program of compiler and runtime system research.

Acknowledgments This work was funded by DoD and by DOE SciDAC PERI (DE-FC02-06ER25764) and USQCD(DE-FC02-06ER41445) projects.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[2] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing Conference*, 2000.

[3] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22:3, Feb. 2006.

[4] J. Carter, Y. He, J. Shalf, H. Shan, and H. Wasserman. The performance effect of multi-core on scientific applications. In *Cray User Group 2007 conference (CUG 2007)*, May 2007.

[5] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel C. In *PPOPP*, pages 36–47, 2005.

[6] G. Hager, T. Zeiser, and G. Wellein. Data access optimizations for highly threaded multi-core CPUs with multiple memory controllers. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing, 2008 (IPDPS 2008)*, pages 1–7, April 2008.

[7] R. L. Knapp, R. L. Pase, and K. L. Karavanic. ARUM: application resource usage monitor. In *9th Linux Clusters Institute International Conference on High-Performance Clustered Computing*, April 2008.

[8] A. Mandal, R. Fowler, and A. Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *2010 IEEE International Symposium on Performance Analysis of Systems and Software*, White Plains, New York USA, March 2010.

[9] J. McCalpin. The Stream Benchmark Page. http://www.cs.virgnia.edu/stream/.

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

[11] L. McVoy and C. Staelin. LMBench: portable tools for performance analysis. In *Proceedings of USENIX Annual Technical Conference*, 1996.

[12] T. Moscibroda and O. 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.

[13] D. Pase. Linpack HPL performance on IBM eServer 326 and xSeries 336 servers. Technical report, IBM, July 2005.

[14] L. Peng, J. Peir, T. Prakash, C. Staelin, Y. Chen, and D. Koppelman. Memory hierarchy performance measurement of commercial dual-core desktop processors. *Journal of Systems Architecture*, 54:8:816–828, August 2008.

[15] A. Porterfield, N. Nassar, and R. Fowler. Multi-threaded library for many-core systems. In *Workshop on Multi-Threaded Architectures and Applications*, Rome, Italy, May 2009.

[16] D. Reed, R. Aydt, R. Noe, P. Roth, K. Shields, B. Schwartz, and L. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proceedings of the Scalable parallel libraries conference*. IEEE Computer Society, 1993.

[17] SciDAC QCD. US Lattice Quantum Chromodynamics. http://www.usqcd.org/.

[18] S. Shende and A. Malony. The Tau parallel performance system. *IJHPCA*, 20(2):287–311, 2006.

[19] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, 2002.

[20] M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely. A genetic algorithms approach to modeling the performance of memory-bound computations. In Becky Verastegui, editor, *SC*, page 47. ACM Press, 2007.

[21] Rice University. Hpctoolkit. http://www.hpctoolkit.org/.

[22] S. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[23] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*, 2008.

[24] S. Williams, K. Datta, J. Carter, L. Oliker, J. Shalf, K. Yelick, and D. Bailey. PERI: auto-tuning memory intensive kernels for multicore. *Journal of Physics: Conference Series*, 125 012001, 2008.

[25] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, November 2007.