Empirical Evaluation of Multi-Core Memory Concurrency Initial Version

TR-09-01

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

January 23, 2009
(Revised February 4, 2009)

Empirical Evaluation of Multi-Core Memory Concurrency
Initial Version

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

January 23, 2009
(Revised February 4, 2009)

Abstract

Multi-socket, multi-core computers are becoming ubiquitous, especially as nodes in compute clusters of all sizes. Common memory benchmarks and memory performance models treat memory as characterized by well-defined maximum bandwidth and average latency parameters. In contrast, current and future systems are based on deep hierarchies and NUMA memory systems, which are not easily described this simply. Memory performance characterization of multi-socket, multi-core systems require measurements and models more sophisticated than than simple peak bandwidth/minimum latency models.

To investigate this issue, we performed a detailed experimental study of the memory performance of a variety of AMD multi-socket quad-core systems. We used the pChase benchmark to generate memory system loads with a variable number of concurrent memory operations in the system across a variable number of threads pinned to specific chips in the system. While processor differences had minor but measurable impact on bandwidth, the make-up and structure of the memory has major impact on achievable bandwidth. Our experiments exposed 3 different bottlenecks at different levels of the hardware architecture: limits on the number of references outstanding per thread; limits to the memory requests serviced by a single memory channel; and limits on the total global memory references outstanding were observed. We discuss the impact of these limits on constraints in tuning code for these systems, the impact on compilers and operating systems, and on future system implementation decisions.[1][2]

[1]This work was supported in part by the DoD, in part by the DOE Office of Science SciDAC PERI (DE-FC02-06ER25764) and in part by NSF: Collaborative Research: The NSF Cyberinfrastructure Evaluation Center (SCI-0510267)

[2]This paper has been submitted to SIGMETRICS/Performance 09. Please limit distrubution.

1 Introduction

Moore's Law, the observation that the number of devices possible in an integrated circuit has been following an exponential growth trajectory, has had two very different effects in processor and memory chips. In processor chips, performance has increased dramatically because shrinking feature sizes allow higher clock rates and more logic gates allow higher concurrency. For many years, the higher concurrency manifested itself as instruction-level parallelism (ILP) realized through the use of multiple deep pipelines and complex, out-of-order control. More recently, designs use explicit parallelism through multiple cores running multiple threads. In contrast, DRAM memory parts have seen a dramatic increase in capacity, but more modest increases in speed and support for concurrency (buffering) within the chips.

This dichotomy between computational power and concurrency on one side versus capacity on the other is starting to manifest itself in the results of studies of multi-core chips for scientific computing. In empirical studies of tuning for a variety of multi-core architectures, Williams *et al.* [23, 24] observed that while the Intel quad-core chips (5300-series (Clovertown), 5400-series (Harpertown)) have greater peak floating point capacity than AMD Opteron systems, the latter outperform the former in practice on many multi-thread, multi-socket benchmarks. With a memory controller and a set of memory DIMMs per processor chip, the AMD architecture adds scalable memory throughput. The observation that fully populating the DIMM slots in an AMD system improves performance dramatically [24] reinforces the observation that adding memory hardware, thus potential memory concurrency, directly addresses a performance problem. The newest generation of Intel quad-core systems (Core i7), have addressed this problem with an entirely new memory subsystem using more memory channels and faster memory DIMMs leap-frogging current AMD performance.[3]

As the number of cores increase within a system node, the effective memory bandwidth per core is dropping. The amount of data per clock cycle that each core can access from off-chip is well below the amount historically considered necessary for a balanced system [1]. As multi-socket systems become increasingly memory-bound, application optimization will become less concerned by the number of floating point operations and more focused on minimizing off-chip data traffic though maximal reuse of data on the chip.

In this paper, we report on experimental studies to quantify memory performance differences among a set of current high performance system design variationss. This includes various memory design options for multi-socket AMD Opeteron systems, as well as a preliminary architectural comparison between Intel Core i7 and AMD Phenom systems. The goal is to measure effects due to implementation and configuration differences. This empirical study is the first step towards developing performance prediction models that can be used for application tuning, guiding optimizing compilers, and to help guide the design and configuration of new systems, at both the hardware and software levels.

[3]the Intel Core i7 is currently available only in single socket systems, we intend to test multi-socket systems as soon as they become available.

1.1 Memory Performance Benchmarks

Memory performance is typically stated using two simple measures. Memory read *latency* is the time between issuing a a memory request and when when the data is available in the CPU. Memory *bandwidth* is the rate, usually expressed in bytes per second, with which a sequence or reads, writes, or some mix can transfer data between memory and the CPU. These are often treated as scalar parameters that are fundamental properties of the system.

Measuring these quantities is problematic. Deep cache hierarchies reduce the number of operations that go all the way to memory, thus decreasing effective latency. Hardware prefetchers detect access patterns and start fetches before they are actually issued by the CPU. Compilers schedule fetches or insert software prefetch instructions to mask memory latency with computation. Each of these mechanisms effectively hides the actual memory latency from the user. Beyond what the processor does to hide latency, the memory subsystems themselves have no single latency number. The latency depends on what work the memory has to complete. A TLB miss requires the equivalent of a memory access, which may or may not be satisfied in cache, before starting the requested memory reference. A DDR buffer hit completes quickly because it does not require operating on the dynamic memory cells. A DDR buffer miss requires that a buffer be refilled from the dynamic memory cells. Because of these phenomena, the latency of each individual operation may vary over a wide range.

On the bandwidth side, there is a substantial gap between the theoretical bandwidth that can be delivered by DDR memory parts and what can be achieved by “real” programs. In the single-threaded case, the gap is especially sensitive to code generation issues such as whether “wide” transfers are used, the scheduling of instructions and the interleaving of reads and writes to different locales in memory. In a multi-threaded environment, the memory operation streams are also interleaved in real time; this affects performance by masking the locality present in each stream, thus changing the DDR buffer contents each stream encounters. Other threads can either push out data a thread needs (a cache conflict miss variant) or bring in data that a different thread will use (false sharing as a benefit).

Application performance models can use latency and bandwidth metrics as parameters. For example, Snavely *et al.* [22, 3] use a linear model in which expected performance is computed as an inner product of a pair of vectors, one of which characterizes the system using a set of rates for various resources, include unit-stride and “random” memory bandwidths, and the other characterizes applications in terms of demand for each resource. This works well in a domain in which the linear relationships hold. As we discuss below, multi-socket, multi-core systems have bottlenecks at several levels and memory limitations do not necessarily fit linear models. There will be a point at which memory requests saturate the memory and bandwidth no longer increases. The saturation can happen at several spots in the memory hierarchy, each potentially needing a different set of parameters.

Given the general emphasis on latency and bandwidth, most characterizations[20, 15, 4, 7] use variations of the LMBench [14] and STREAM [13] benchmarks to measure these system characteristics. We believe these approaches only tell part of the story for multi-socket multi-core systems.

1.2 Multi-Socket, Multi-Core Performance

Given the trend towards multi-socket, multi-core systems, memory models need to incorporate some notion of concurrency. This is not entirely orthogonal to the use of latency and bandwidth measures. Little's law [10], a fundamental result of queuing theory, states that the average number of requests in a system is equal to the product of the arrival rate and of the average time each request resides in the system. In communication, including memory communication, Little's Law can be restated to be that the number of bits "in flight" is equal to the latency times the bandwidth. In terms of memory operations, the number of concurrent operations in the system is equal to latency times bandwidth, with the latter expressed in terms of memory blocks per second. In this paper, we focus on operations that go all the way to memory, so the concurrency measure is the number of concurrent cache misses plus the number of concurrent writebacks. Any of the concurrency, latency, and bandwidth measures can be computed from the other two. Thus, we control concurrency, measure bandwidth, and compute effective latency.

Because of the importance of concurrency in the systems of interest, we are advocating using concurrency and latency as the fundamental quantities for modeling. Latencies are determined largely by the underlying technology. The degree of concurrency to memory that a system can provide is largely a function of higher-level system design and can depend on several factors including: the number of outstanding cache misses each core can tolerate, the number of memory controllers, and the number of concurrent operations supported by each controller, memory communication channel design, and the number and design of each of the memory components.

The amount of memory concurrency in an application is determined by the effectiveness of the hardware (out-of-order execution, etc.) at detecting independent memory operations and by the quality of the compiled code. High performance can be achieved either by offering a low memory access load (low miss rates), or by effectively overlapping memory operations with computation and with other memory operations. Placement of threads and data can greatly impact performance. Access latency can be reduced if a thread accesses data contained in memory local to its processor. If too many threads access data attached to a single memory controller, that memory bank will saturate and access latencies will increase.

1.3 Our Approach

The pChase suite of memory benchmarks [16, 9] is measures effective latency and bandwidth for a variety of access patterns. For this study, we use pChase specifically in a mode to test memory throughput under a carefully controlled degree of concurrent access. This is done by having each thread execute a loop with a controllable number of independent pointer chasing operations per iteration. Each sequence of pointer addresses is pseudo-random and designed to defeat hardware prefetching while limiting TLB misses.

We added wrapper scripts around pChase to iterate over different numbers of memory reference chains and threads for each pChase run. We also modified it to control thread placement.

We use pChase to explore the memory performance for variety of systems including a set of similar AMD Opteron multi-socket systems, a 'gamer' AMD Phenom system and a single socket Intel Core i7 system.

Variations in various hardware resources result in differences not only in the bandwidth limits but relative performance when running in sub-optimal performance areas. The systems use the same processor family, but vary in processor speed, number of processors, the size and speed of the memory parts, the number of memory parts installed and system mainboard design. Some of the systems were retested with alternate memory provisioning. This allows us to measure the impact of changing the number and type of memory parts on the bandwidth.

The STREAM benchmark [13] measures effective bandwidth for a set of simple stride-one access patterns. While Stream results are proportional to and are constrained by the architectural capabilities, the sensitivity of its results to the quality of generated code make it a good test of compiler optimization for such simple loops. Performance differences of almost 2x, have been observed between compiler generated and hand-tuned assembler versions [12]. The standard Stream benchmark can also be run in parallel on an SMP using OpenMP. This mode is additionally affected by thread (and data) placement, as well as by the ability of the system to handle multiple interleaved memory references streams. We use Stream results to help validate and interpret the pChase numbers.

2 pChase Benchmark

pChase is a flexible memory performance benchmark. For a given system, it measures the effective memory latency and bandwidth for different access patterns (random or strided), number of threads, number of memory references per thread, cache size, page size etc. pChase gets its name from "pointer chasing" because it walks through a chain of memory references that is initialized such that the pointer to the next memory location is stored in the current memory location. This ensures that the next fetch cannot be issued until the result of the previous one is available.

When the benchmark runs, each memory reference chain is traversed in a loop from the root of the chain to the end. If the chain is initialized in a random way, the reference chain traversal generates random memory fetches that should defeat hardware prefetching strategies.

An experiment run specifies the number of concurrent miss chains generated by a thread. The chains within a thread are independent, so their references can be resolved concurrently.

An experiment run also specifies the number of threads that access memory concurrently. Threads that run on different cores will use different paths to memory and may use different memory components. While concurrent references within a thread all use the same paths, references from threads on distinct cores and sockets use distinct on-core components, but will interleave and contend with one another in the downstream paths and components of the system.

For this work, we parametrize each experiment run of pChase by (a) the memory requirement for each reference chain, (b) the number of concurrent miss chains per thread and (c) the number of threads. We have kept the other parameters (page size, cache line size, number of iterations, access pattern etc.) fixed. We extended pChase to pin threads to cores and to perform memory initialization after the pinning.

3 Execution Environment

We performed our pChase experiments on eight system configurations. Six were various AMD workstations and blades from Dell that support either two or four AMD quad-core Opteron (Barcelona) processors. All six have DDR2/667 memory, though the number and type of memory parts vary. The remaining two are custom built systems with other memory configurations.. One is an AMD Nehalem with DDR2/1066 memory, and the last is an Intel Core i7 system using DDR3/1333 memory.

All Linux kernels were recent stock kernels downloaded from kernels.org with only the Perfmon2[6] package added to supply an interface to the hardware counters for the OS and application. Four different systems were used and two of those systems were retested with a portion of their memory removed to judge the effects of reduced DDR2 buffering and uneven memory bank sizes. By examining several systems with faster DDR2 and DDR3 memory, the effects of base memory speed and configuration are examined.

3.1 Systems

WS1 is a Dell PowerEdge T605 system running Ubuntu with a 2.6.25.9 kernel. It has 2 AMD 2.1GHz processors and 2GB of main memory for the 8 cores. The 2GB memory consists of 4 sticks of single-rank 512MB DDR2 memory.

WS2 is the same box as WS1 with twice the installed memory. The WS2 configuration has 8 sticks (fully loaded) of single-rank 512MB DDR2 memory for a total of 4GB.

WS3 is a second Dell PowerEdge T605. It has 2 AMD 2.2GHz processors with 8 cores and runs the Ubuntu with a 2.6.25.9 kernel. Besides having slightly faster processors, WS3 is fully loaded with 8 2GB dual-rank DDR2 memory sticks for a total of 16GB.

WS4 is a locally assembled 'gamer' AMD Phenom system based on an ASUS M3N-HT Delux motherboard. It has single 2.5GHz processor with 4 2GB dual-rank DD2/1066 memory sticks for a total of 8GB. This system has faster memory than any other of the tested AMD systems.

IN is a locally assembled Intel Core i7 system. It has a 2.93GHz clock and 3 single memory channels each with a single dual-rank 2G DDR2/1333 memory stick for a total of 6G.

LP is a Dell PowerEdge M600 dual-socket blade with 2 AMD 1.9GHz Low-power Opterons and 8 cores. It has CentOS Linux installed using the 2.6.26.2 kernel. For main memory, it has 8 dual-rank 2G DDR2 memory sticks for a total of 16GB.

QB1 is a Dell PowerEdge M905 quad-socket blade with 4 AMD 2.3GHz processors installed for a total of 16 cores. It runs CentOS Linux with a 2.6.26.2 kernel. QB1 has 24 dual-rank 2GB DDR2 memory sticks (fully loaded) for a total of 48GB.

QB2 is the same blade as QB1 configured with 16 dual-rank 2GB DDR memory sticks for a total of 32 GB. Each socket has 8G memory to balance the available memory.

3.2 Performance Variability

During initial testing, performance values varied tremendously (up to 60% on QB1/2). This was determined to be due to unfortunate interactions among the Linux scheduler, the Linux memory allocation/placement policy and the threading library. Threads were initially assigned randomly to idle cores. Since the threads were launched in quick succession, the more than one thread was placed on some (still idle) cores. The program then performed data initialization. The Linux memory allocator assigned the pages to memory near that core. The scheduler would then migrate threads to the threads to idle cores. By then then, however, data was already allocated in memory close to the other core. On a quad-core blade, the benchmark often used fewer than all four memory controllers, thus introducing bottlenecks that limit performance.

To reduce this effect, code was added to pChase to pin each thread to a unique available core based on the pthread thread number before data initialization. To limit execution to cores on specified sockets, a version of `taskset` that uses `sched_setaffinity` written at NC State was used, `cpuaff`. `cpuaff` allows the program use `sched_getaffinity` to determine the available set of cores. The need for and effects of pinning is further explored in [21].

4 pChase Memory Performance

For each system, we varied the number of threads from 1 to the number of cores available on the system. We varied the number of reference chains per thread from 1 to 15. For some systems, we also varied the memory requirements for each reference chain between 128MB and 16MB. We repeated each experiment run 10 times and reported the average memory bandwidth values in the results.

The bandwidth reported in our mode of operation by pChase for any number of concurrent references from a single single thread is lower than an optimized Stream benchmark. Hence, it is not indicative of the performance one can achieve for long strings of references amenable to hardware prefetching. On the other hand, we believe it is a fair indication of what a well optimized application that does random, but concurrent, references can expect.

4.1 Single Socket Measurements

The first series of experiments examined the performance when threads run on only one socket[4] on each of the 6 configurations. All of the graphs have the same general shape, but changes in the underlying systems

[4]Linux logical core numbering can change during the boot process. Care was taken to make sure all threads were assigned to correct sockets throughout.

result in some measurable differences.

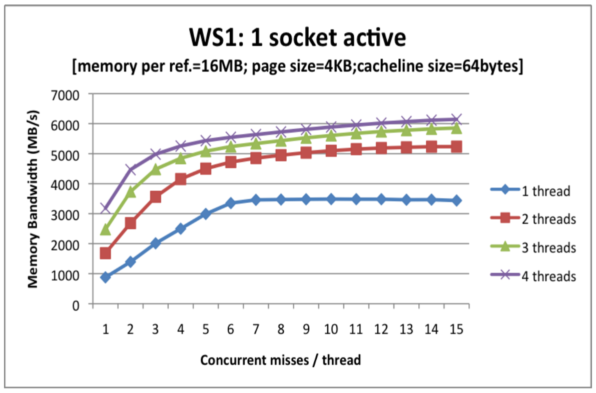


Figure 1: Memory Bandwidth vs. Number of references (WS1)

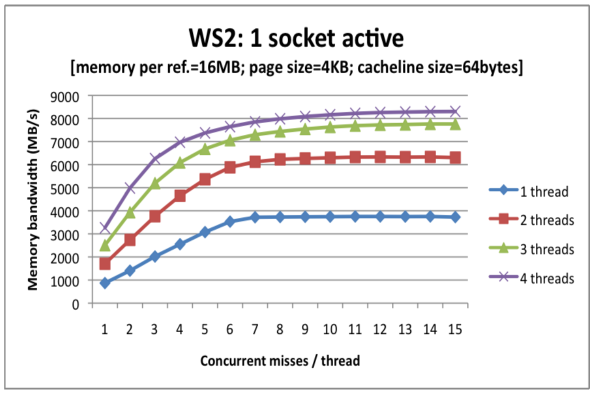


Figure 2: Memory Bandwidth vs. Number of references (WS2)

Examining Figure 1, it is clear that a single thread's performance does not improve after the 7th concurrent memory reference. A single core has a limited number of outstanding memory references, which the pChase graph easily identifies; the curve is in two nearly linear segments joint at a knee. Below 7 concurrent operations latency per operation is roughly constant (85-90 nanosec) and throughput increases linearly. Above 7, throughput is constant and residence time (latency) increases linearly as additional operations are queued.

As the number of threads increases, the curves exhibit knees further to the left but at higher bandwidths. A second bottleneck occurs when the memory concurrency exceeds 25-30 concurrent memory references. At the knee, the threads on one processor chip get about 95% of the memory bandwidth achievable with any number of outstanding references. Collectively, threads on the cores of a single chip can generate more concurrent memory references than this memory configuration can effectively support.

The two configurations that generated Figures 1 and 2 only differ in the number of DDR2 memory sticks installed (WS1-4x512MB, WS2-8x512MB). DDR memory has a small number of memory page buffers that support high speed data transfer. If the data is not present in a buffer, the page must be loaded from the lower speed dynamic memory cells into a buffer and then transferred. Doubling the number of memory sticks doubles the number of buffers available and doubles the number of buffers and memory cells that can potentially operate concurrently. Performance for WS1 matches (±2%) WS2, when there are 3 or fewer concurrent references in the system. For high levels of concurrent access, if only one thread is active WS1 performance falls to 92% of WS2. When 3 and 4 threads are active on WS1, total performance falls to about 75% of WS2 with about 15 total concurrent refences. The lack of memory chips (buffers and DRAM banks) greatly limits the number of concurrent accesses the hardware can handle.

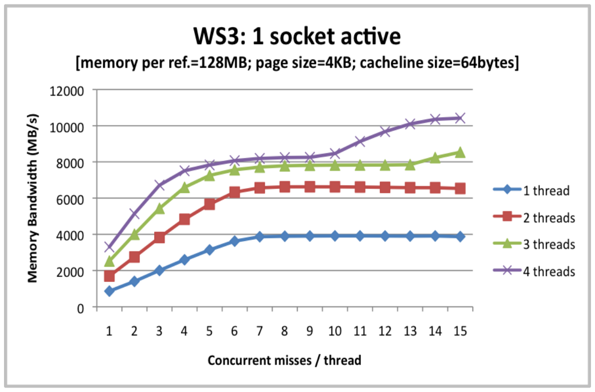


Figure 3: Memory Bandwidth vs. Number of references (WS3)

Compared to WS2, WS3 (Figure 3) has a sightly faster CPU (2.2 vs 2.1 GHz) and DDR memory sticks 4 times the size(8GB vs 512MB) and twice the number of ranks per stick. Surprisingly, when there are fewer than 7 or 8 concurrent references active, the performance of WS2 is higher than WS3(up to 8% with only 1 reference active). As the number of references increases WS3 performance passes WS2 and at the peak is 2-4% higher(less than the $\sim$ 5% clock advantage). WS3 outperforms WS2 by 4-8% when 3 or 4 cores are active and there are less than 30-35 concurrent memory references.

The most noticeable feature in Figure 3 is the increase in performance of 4 threads after 9 concurrent references per thread. Using the Perfmon2, we tracked this back to Linux memory allocation/placement decisions. When running on a single socket the memory is allocated on that socket until it exceeds 4GB, then it uses a second socket. For the reference chain size of 128MB, this happens when about 40 independent memory references chains are created. The increase in performance for 4 threads and 10+ memory references per thread and for 3 threads with 14+ memory references per thread, is a consequence of getting both memories and controllers active. There is a limit to how much memory traffic can be offloaded to the second controller, but the bandwidth benefits of some offloading can be substantial. If a small number of threads require a lot of bandwidth on such systems, distribution of data across multiple memory controllers may be an attractive strategy.

In Figure 4, we examine a blade (LP) with the the slowest clock processor tested. The graph resembles the WS3 graph (Figure 3), including the rise after 4GB data is allocated. When there are fewer than 10 concurrent references, memory bandwidth on LP is less than WS3 (up to 5-8%). As the number of reference increase LP performance is within ±1-2% (mostly higher). The slower processor clock rate on the low-power blade does not seem to impact memory bandwidth, making low-power system very attractive for memory

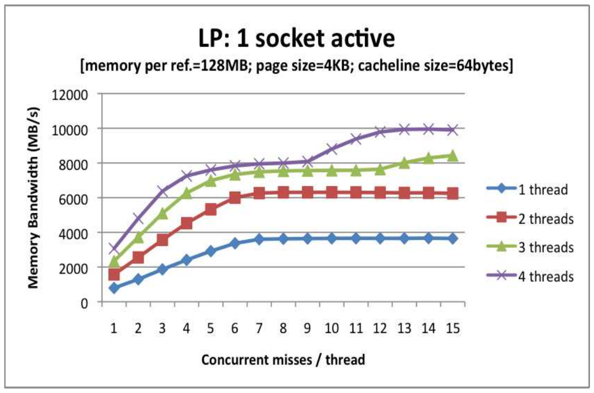


Figure 4: Memory Bandwidth vs. Number of references (LP)

bound applications.

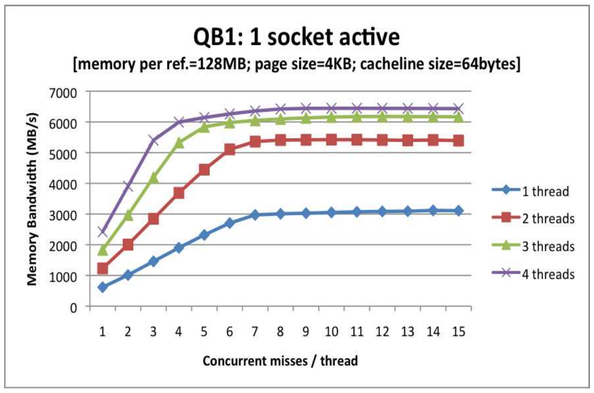


Figure 5: Memory Bandwidth vs. Number of references (QB1)

QB1 is a quad-socket system and the impact of supporting additional sockets is very evident in figure 5. When the load is all generated from one socket, the bandwidth is much lower than WS3 for all numbers of active references, from 71% (1 thread 1 reference) to 87% (3 threads 5 references).

QB1 has four controllers, 2 support 4 2G memory sticks and 2 have 8 2G memory sticks. QB2 (Figure 6) balances the memory bank size by reducing all of the banks to 4 2G memory sticks. Performance for low numbers of memory references is unchanged, but as the number of concurrent reference grows beyond 10, QB2's memory performance matches WS3 and exceeds it by 6% in three cases (4 threads 5-7 references). QB2 also exhibits the bandwidth increase as second socket's memory controller handles more load. As the size of the problem increases, the performance tails off (see section 4.3). Balancing the memory banks greatly improves peak memory performance.

For all of the graphs present so far, all the load is generated on one processor chip/socket. Bandwidth does not scale linearly, indicating the presence of bottlenecks. As the thread count goes from 1-4, achieved

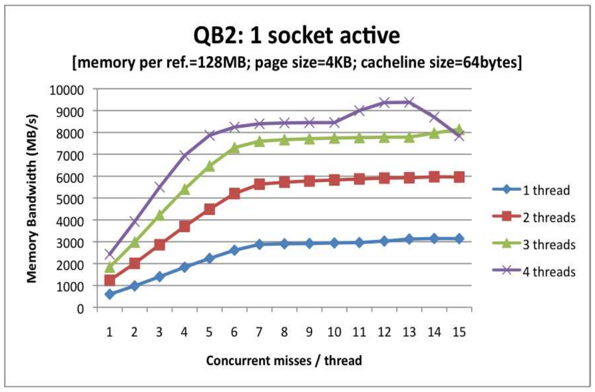


Figure 6: Memory Bandwidth vs. Number of references (QB2)

bandwidth increases by a little less than 2x in most cases and never more than 2.1x. Adding a second thread on a second core improves performance over the single thread case by about 50%, but a third only adds another 15% and the fourth core increase performance by only about 5%. As the number of active threads increases, the effective memory bandwidth for each thread drops noticeably. Memory-bound applications will have significant scaling problems on current multi-core chips.

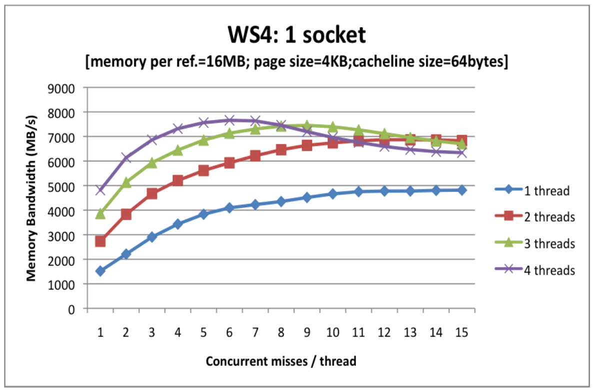


Figure 7: Memory Bandwidth vs. Number of references (WS4)

WS4 memory has memory that is about 33% faster than the other AMD systems tested. It also uses a different processor (Phenom) from the same family, and it has a 10 to 20% faster clock rate than the Opterons previously examined. These changes have some significant impacts. The peak bandwidth for one core is almost 5 GB/sec compared to 4 GB/sec for WS3. See Figure 7. Performance is also much better for two active threads. 3 and 4 active threads have a drop-off in performance when the entire chip has about 30 active memory references active. WS4 only reaches a peak of 7.8 GB/sec while WS3 achieved 8.1GB/sec (before the hump). The memory performance drop of about 20% as the number of references increases is particularly worrisome. This Phenom system requires a larger number of outstanding memory references from a single core to reach the peak. Furthermore, there's a performance dropoff with more than a total of about 30 outstanding references in the system. Since performance actually drops, this more than just a simple bottleneck, rather an indication of a problem in the handling of high loads in the memory controller.

Code tuning efforts, either manual or in a compiler, need to recognize the multi-core limitations and to strive to have only about 6 or 7 memory references outstanding per core in parallel sections of the code.

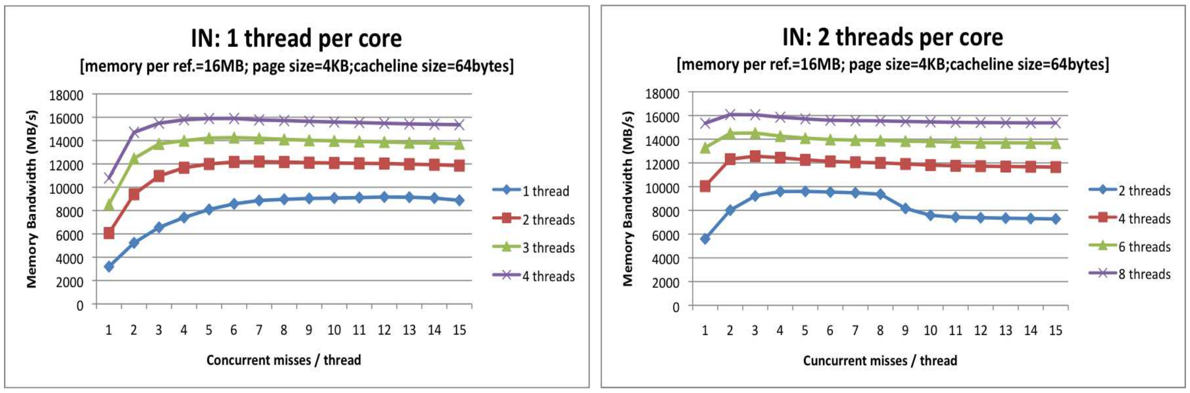


Figure 8: Memory Bandwidth vs. Number of references (IN)

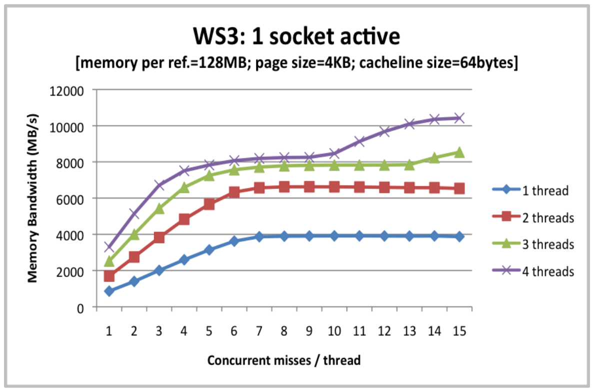
IN is an Intel core i7 system assembled from parts we ordered the first week the chip was available. The processor has a faster processor faster clock (2.9 GHz - compared to 1.9 - 2.5 for the various AMD systems) and a faster, more parallel DDR3 memory subsystems. Each socket of the AMD systems has a single dual-channel, dual-rank memory subsystem. The Intel Core i7 system has three independent single-channel, dual-rank DDR3/1333 memories. The processor supports two threads per core using Intel's Hyperthreading. With more memory channels and faster memory, then memory performance in Figure 8 is much higher than for the AMD systems.

A single thread can achieve over 9 GB/sec, better than double the highest observed AMD result. With one thread on each of four cores, the peak reaches 16 GB/sec. Using hyper-threading does not improve the peak bandwidth and it incurs 1-3% penalty in some cases. Using hyperthreading, fewer outstanding references are required from each thread in order to get the same level of memory concurrency. All of the performance curves quickly reach their peak and trail off slightly under higher load. This drop is more evident in the 2 threads per core graph. The Core i7 has a significant performance issue when two threads on one core present a load of more than 16 outstanding references. In this case, performance rapidly falls about by 20% as load increases.. This reduced performance is only slightly better than what can be achieved with two threads on an AMD system. We do not have an explanation for this falloff and the problem deserves further investigation.

With the same total number of outstanding references, the Intel Core i7 system gets about twice the memory bandwidth of a DDR2 Opteron system. Compilers and applications tuned for one should do well on the other with respect to this issue. As the number cores expands of outstanding memory references per core needed to saturate the system is falling. Restated, the number of concurrent references per core that the system can effectively serve is decreasing. Such systems will require less aqgressive compiler optimization in this regard.

4.2 Multi-socket Performance Experiments

For a multi-socket, multi-core system the real key to performance will be how system performance scales up as parallelism across processor chips is added. Adding cores and sockets must provide applications with higher performance.



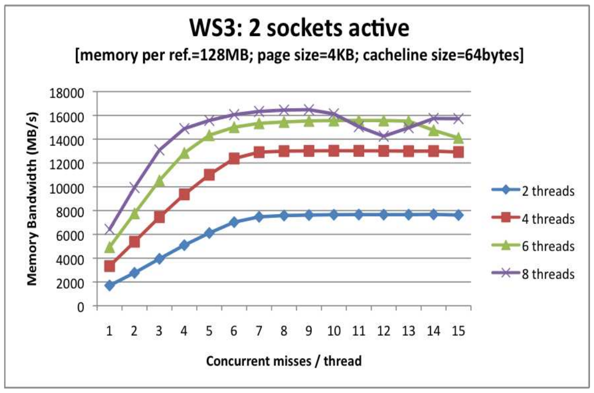


Figure 9: Memory Bandwidth vs. Number of references (WS3)

We've seen that for memory intensive benchmarks that scaling by adding threads and cores to one chip has problems. We will see that , scaling to a second socket on the same node is excellent. In Figure 9, which extends Figure 3 with the second socket results, performance nearly doubles comparing 1 thread - 1 socket to 2 thread - 2 sockets, 2 threads - 1 socket to 4 threads - 2 sockets etc. Scaling is good up to the point where the single socket experiment memory allocation spans onto a second memory. The dip in WS3 performance for 2 sockets and 6 and 8 threads is a product of the memory placement policy (See section 4.3).

One socket of WS3 reaches its memory bandwidth limit of about 8GB/sec with between 25 and 30 concurrent memory references. Adding a second socket increases the peak to 16GB/sec with about 45 memory references active. In both cases, reaching the peak requires that the concurrent memory references per thread be near the per core bottleneck limit of 7.

That performance on QB1 scales up to 4 sockets (Figures 10 and 11) is particularly interesting since this is approximately the base node configuration for both the biggest Cray XT-4 and Sun Sunfire systems (although they use different motherboards than we are testing). From 1 to 2 sockets maximum achieved bandwidth increases from 6+GB/sec to 12+GB/sec, again showing very good scaling. Adding a third socket increases the total bandwidth to 18GB/sec, but activating threads on the fourth socket adds very little at the maximum. There is a bottleneck at $\sim$ 50 concurrent memory references. A plausible explanation is the coherence protocol. Each DRAM buffer access corresponds to a cache miss that requires coherence communication across the HyperTransport links. For a quad-socket system the message overhead seems to be 8 or 9 messages for one coherence Probe (Figure 12). Hence, each memory controller seems to be limited to limited serving to DRAM operation every 9 cycles.

Adding a fourth socket (Figures 10 and 11) does not change the 18GB/sec limit. There are more threads supplying references, the limit is easier to reach. 8 threads (2 per socket) each with the 7 allowed per socket just reaches the peak. 12 and 16 threads reach the peak with substantially fewer than 7 concurrent memory

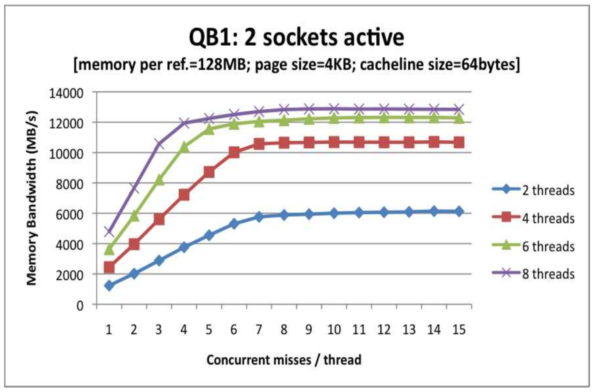
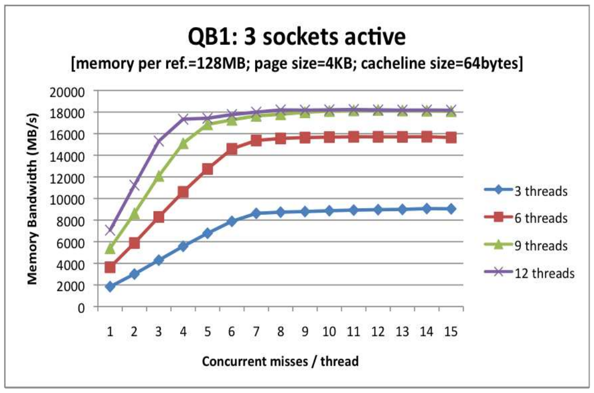


Figure 10: Memory Bandwidth vs. Number of references (QB1)



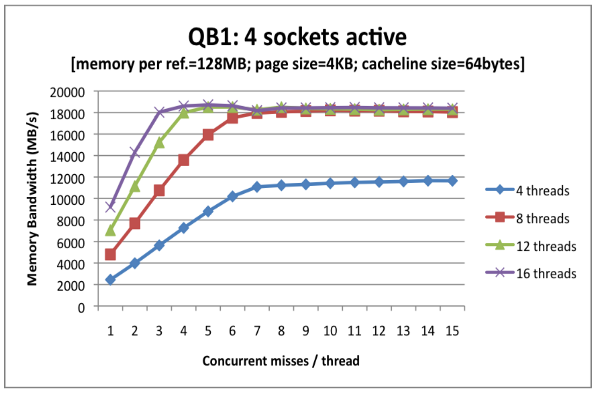


Figure 11: Memory Bandwidth vs. Number of references (QB1)

references. Since sixteen threads are active, the 50 concurrent reference limit is reached at slightly over three references per thread. As the core counts rise, unless memory systems change drastically, out-of-order execution and compiler optimizations to allow concurrent memory access from within a thread are going to become less important.

One interesting experiment to run will be on multi-socket Intel Core i7 to see if the coherence limitations is architecture specific or general across the x86_64 family. AMD has made public statements[8] about the inclusion of hardware in future generations to reduce the cohrence traffic (in the normal/average case), thereby increase the multi-socket boards total bandwidth.

Balancing the memory has little effect(Figures 13 and 14) on the maximum throughput performance or on the graphs that reach that maximum peak. The maximum does not seem to be caused by the size or configuration of banks.

```
12 References per thread (17610.172 MB/sec)
CPU0    35464602629 CPU_CLK_UNHALTED
CPU0     3969394299 PROBE:ALL
CPU0     3946977244 PROBE:MISS
```

Figure 12: Partial Hardware Counters 12 references per thread (16 threads)

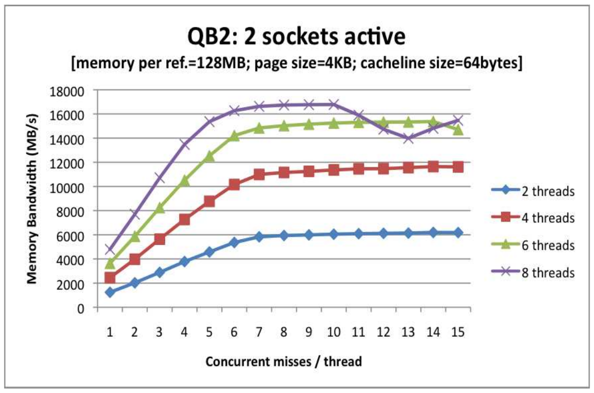


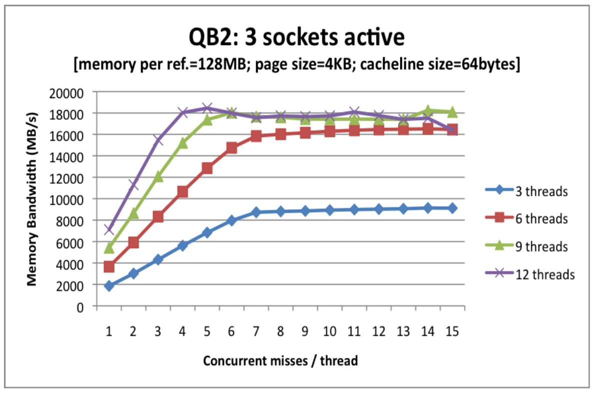
Figure 13: Memory Bandwidth vs. Number of references (QB2)

4.3 Multi-socket Performance Anomalies

Interesting anomalies occur in the portions of the curves with greater than 10 memory references per thread active. For one socket with 40 memory references active, a second knee occurs. Performance rises by 20-30% for 60 total active memory references. When 2 sockets are active, a dip in performance occurs between 80 and 120 total outstanding memory references.

We used Perfmon2 to collect node-wide hardware counter values, both of these anomalies were tracked back to Linux memory placement decisions. When running on a single socket the memory is allocated on that socket until it exceeds 4GB. For the reference chain size of 128MB, this happens when about 40 independent memory references are created. The increase in performance for 4 threads and 10+ memory references per thread and for 3 threads with 14+ memory references per thread, results from having both memories and controllers active. There is a limit to how much memory traffic can be offloaded to the second controller, but the bandwidth benefits of some offloading can be substantial. The performance dip in the 2 socket 8 thread case, is also explained by Linux memory placement policy. The memory is evenly split between the sockets until 8GB has been allocated, then all of the memory between 8 and 12G is allocated on the same core and all of the memory between 12 and 16G is allocated on the other. The effect is that as up to 2/3 of the memory references try to go to one memory controller, performance degrades towards the 1 socket peak. As the second memory eventually see closer to half the references the performance increases. pChase did not show us what was wrong with memory performance, but it does identify regions that require closer inspection.

The last anomaly is less noticeable, 4 sockets and 4 threads per socket of QB1 and QB2 have the highest



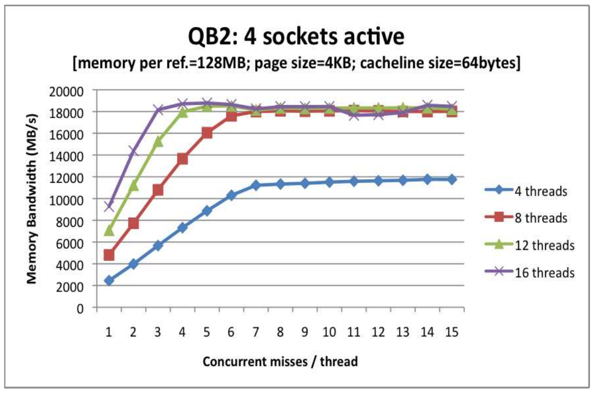


Figure 14: Memory Bandwidth vs. Number of references (QB2)

```
4 References per thread (18742.689 MB/Sec)
CPU0    12313670564 CPU_CLK_UNHALTED
CPU0       57704330 DRAM_ACCESSES_PAGE:HIT
CPU0      120925118 DRAM_ACCESSES_PAGE:MISS
CPU0       10233499 DRAM_ACCESSES_PAGE:CONFLICT

12 References per thread (17647.186 MB/sec)
CPU0    35263866876 CPU_CLK_UNHALTED
CPU0       13996124 DRAM_ACCESSES_PAGE:HIT
CPU0      483192511 DRAM_ACCESSES_PAGE:MISS
CPU0       19351122 DRAM_ACCESSES_PAGE:CONFLICT
```

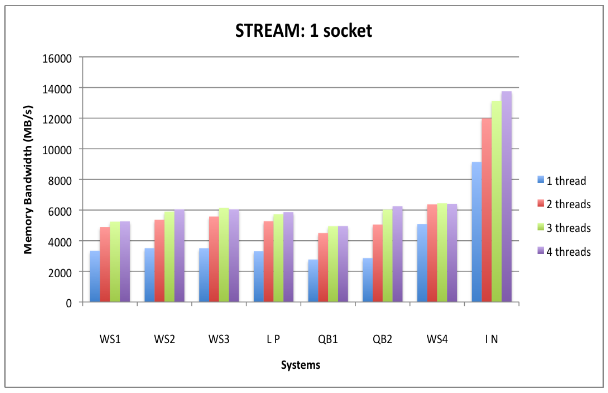
Figure 15: Partial Hardware Counters for 4 and 12 references per thread (16 threads)

bandwidth with 4-6 references per thread and then fall slightly to a lower steady state. The performance was examined using the hardware counters for multiple runs using Perfmon2 (Figure 15). As expected, time (as measured by CPU_CLK_UNHALTED) for the 12 reference case is roughly 3 times that of 4 reference case. Despite the randomization of the memory chains, we get a significant number of buffer hits when 4 memory chains are active. The number of DRAM_ACCESSES_PAGE:HITs is lower with 12 memory chains despite the increase in work. The four reference case has better locality in the DDR memory pages. With 12 references, the number of MISSes goes up by 4x and number of CONFLICTs doubles.

In spite of randomization, each thread's memory stream still has a significant degree of locality that manifests itself as an increase in the number of buffer hits. As the number of independent reference streams goes up, the locality in each thread's memory stream is masked by interleaving and the number of buffer page hits is reduced. The number of random buffer hits falls.

The "accidental" memory buffer hits, is an example of a performance opportunity for runtime schedulers. If threads that use data that is co-resident within a cache line or memory page can be co-scheduled on the system, then both will see improved performance from having the data preloaded by one thread for the other in shared caches and unified DDR data buffers. "False sharing" historically has been bad because cache lines or pages are moved needlessly; now it can be a benefit.

5 Related Work



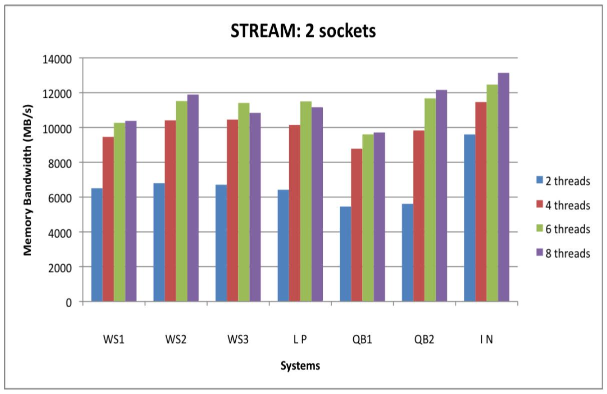


Figure 16: Stream Performance (2 socket IN uses Hyperthreading)

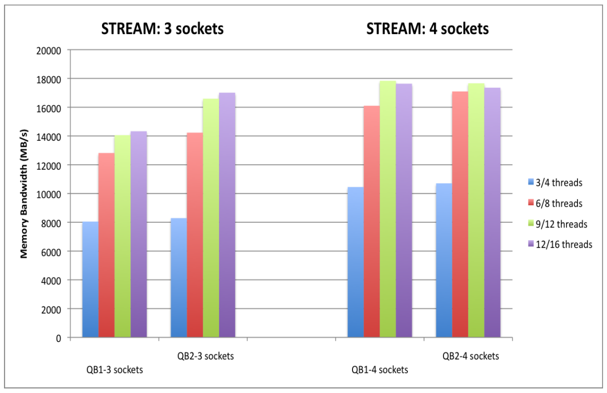


Figure 17: Stream Performance

Pase and Eckl [17, 19, 18] developed and use pChase to examine the memory latency and bandwidth of several IBM systems. Their studies have focused on single local/remote latency numbers, not on getting the whole memory picture.

Using the Stream benchmark by itself to quantify memory performance is ubiquitous in system benchmarking. For instance, the HPCC[11] benchmarking suite used for HPC Challenge awards and in the NSF Track2D call. Stream has been used in a wide variety of memory performance studies including [20, 15, 4, 7].

Stream is a set of four simple vector unit-stride access kernels. It provides lots of opportunity to optimize individual threads using instruction scheduling and prefetching. Multi-threaded systems, however, have less ability to take advantage of the simple nature of the loops. Although useful for determining what the absolute peak memory performance of a system including its compiler, it provides substantially less information for the system designer, system purchaser or the application tuner about how applications will run on the system than pChase does.

If we look at the stream numbers for the eight configurations tested, Figures 16 and 17[5] [6], they do not provide the user with any information about the number of references needed to reach the peaks, or about the origin of the the limitations for a thread, a processor, and the node. For systems where the performance tails off as the number of outstanding references grows, STREAM may not even be measuring the performance at the peak. Each of the limitations is different and is clearly identifiable from the pChase data.

LMBench[14] has also been used to quantify memory latency. It provides information about the size of the various levels of the memory hierarchy in a system. Providing information about latency implications of changing levels within the hierarchy. pChase can also be used in that domain and gives the user more information about the memory hardware support, allowing better optimization.

A popular memory model for application performance modeling in high performance computing [22, 3] uses linear models of memory performance. pChase gives us more realistic memory performance information about multi-socket multi-core bottlenecks that should be incorporated into future the application models. As nodes increasingly become multi-socket, multi-core, application performance will be determined by memory bottlenecks driven by interactions among the threads. Accurate application modeling will require data that complements the detailed system characterizations we present here.

6 Conclusions

These experiments have focused on memory access in a NUMA system. Data and thread placement are critical to achieve a high level of locality Pinning threads for local allocation is crucial on multi-socket systems. We have seen that "first touch" placement [2] is inadequate. It may be appropriate to again consider page migration [5] in NUMA systems, not just to improve base operation latency, but to balance memory load and reduce communication contention.

The complexity of memory systems needs to be considered in the modeling and evaluation of multi-socket, multi-core computation nodes. Using a conventional approach does not accurately model the multiple bandwidth limitations now present. It is important to identify the limit achievable by a single thread, the limit of a multi-core processor chip, and the limit of the multi-socket node. Each level scales until some underlying constraint turns into a bottleneck. Understanding the origins of these constraints will allow better decisions to be made during system acquisition, machine configuration, and during application development and tuning cycles.

We have shown experimental evidence of different constraints on memory concurrency levels at each level. A single thread can take advantage of only 7 concurrent memory reference. Three different X86_64 implementations all showed this same limitation to varying degrees. An AMD controller for a dual-channel DDR2/667 memory bank supports 25-30 references before it saturates. Intel Core i7, with 3 single-channel DDR3/1333 memory banks supports a much higher bandwidth, but, like the AMD systems, the memory system can be saturated with fewer than the full complement of available cores. A 4-socket, quad-core system AMD system

[5]For performance consistency, we modified Stream, so each OpenMP thread was pinned to the next available core before data initialization. The peak results remained the same and become predictable, except for QB1 and QB2 where the observed peak for pinned Stream was higher than the original version.

[6]The values for IN on two sockets are from running two threads on each core via Hyper-threadsing not a second socket.

supports approximately 50 outstanding memory references before something caps bandwidth. We believe that the bottleneck is serialization induced by the coherence protocol and we expect the Intel Core i7 to exhibit a similar limit. Since there are sixteen cores in the four-socket node, this limit corresponds to is about 3 outstanding references per core. Historically, out-of-order processor designs have used a lot of gates and energy attempting to allow as many concurrent memory references as possible. One of the most important compiler goals is to schedule memory references as independently as possible to facilitate concurrent access in the hardware. If the node architecture limits the outstanding concurrent access per thread to only 3 or 4, aggressive core designs and memory operation scheduling algorithms will yield little benefit. Hardware mechanisms to support lighter-weight threads will enable compilers to locate more parallelism and saturate the memory system though parallelism rather than concurrent accesses from within a thread.

With current memory system design, the rapid increase in number of threads that can run concurrently on a node reduces the need for the hardware, compiler and application to identify memory references that can be executed in parallel. This will allow compilers to more easily optimize applications to reach peak performance on simpler hardware designs. For small core counts, instruction level parallelism (ILP) and memory parallelism is necessary and useful. For high core counts, the memory-related payoff for thread ILP and thread memory concurrency is smaller. Unless a breakthrough in memory design occurs, optimizing an application could become entirely a parallelization problem. Individual thread optimization may become irrelevant because of the memory wall beyond each chip boundary.

The memory performance of any given system is highly dependent on the configuration and number of memory parts. Fully populating available memory slots increases capacity, but it can also increase the available memory concurrency by increasing the number of fast buffers and of banks of DRAM cells. Memory allocation and data placement strategies need to spread data across the parts of each local memory to exploit this concurrency.

As the performance anomalies detected indicate, memory and thread placement will have tremendous effects on effective memory bandwidth a memory system can supply an application. Tools and techniques to identify and correct any OS, compiler, runtime library data placement issues will be vital to tuning memory-intensive applications. The limitations in processor-wide and node-wide bandwidth will result in applications that are currently well balanced becoming memory-intensive, and some compute-intensive applications becoming balanced.

Operating system memory allocation and thread scheduling can have effects at least as large as any hardware parameters. The NUMA aspect of a multi-socket system needs to be recognized and respected by the operating system and any runtime library. Although the hardware seems to handle individual remote references efficiently, when one memory handles more than its share of memory references that, memory/memory controller rapidly becomes a performance bottleneck. Accesses to memory need to be evenly spread across memory units. The easiest method is, probably, to make threads reside on the same socket as their memory and the OS needs to evenly allocate memory across the sockets.

In the future, we hope to extend this work by examining multi-socket Intel Core i7 (Nehalem) nodes and IBM Power 6/7 nodes. Each of those systems have different memory coherence protocols and different cpu clock rate to memory clock rate ratios. Comparing different architectures should allow observations about what bottlenecks are fundamental and which are implementation dependent.

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS 1967 Spring Joint Computer Conf.*, pages 483–485, Apr. 1967.

[2] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. Numa policies and their relation to memory architecture. In *ASPLOS*, pages 212–221, 1991.

[3] Laura Carrington, Allan Snavely, and Nicole Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22:3:336–346, February 2006.

[4] Jonathan Carter, Yun (Helen) He, John Shalf, Hongzhang Shan, and Harvey Wasserman. The performance effect of multi-core on scientific applications. In *Cray User Group 2007 conference (CUG 2007)*, May 2007.

[5] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, Litchfield Park, AZ, December 1989.

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

[7] 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.

[8] Hoel Hruska. Amd talks shangai performance, features,roadmap to 2010. http://arstechnica.com/news.ars/post/20080507-aray-of-sunshine-amd-talks-shanghai-performance-roadmap.html.

[9] 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.

[10] J.D.C. Little. A proof of the queuing formula $lw\lambda w$. *Operations Research*, 9, 1961.

[11] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi. The HPC challenge (HPCC) benchmark suite. In *SC06 Conference Tutorial*, 2006.

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

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

[14] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–, 1996.

[15] 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.

[16] Douglas Pase. The pChase benchmark page. http://pchase.org/.

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

[18] Douglas M. Pase and Matthew A. Eckl. Performance of the AMD Opteron LS21 for IBM Bladecenter. Technical report, IBM, August 2006.

[19] Douglas M. Pase and Matthew A. Eckl. Performance of the IBM System x3755. Technical report, IBM, August 2006.

[20] Lu Peng, Jih-Kwon Peir, Tribuvan K. Prakash, Carl Staelin, Yen-Kuang Chen, and David Koppelman. Memory hierarchy performance measurement of commercial dual-core desktop processors. 54:8:816–828, August 2008.

[21] Allan Porterfield, Rob Fowler, Anirban Mandel, and Min Yeol Lim. Performance consistency on multi-socket AMD Opteron systems. Technical Report RENCI Technical Report TR-08-07, Renaissance Computing Institute, 2008.

[22] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, 2002.

[23] 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.

[24] 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.