LBNL-62500

Understanding and Mitigating Multicore Performance Issues on the AMD Opteron™ Architecture

John Levesque, Jeff Larkin, Martyn Foster, Joe Glenski, Garry Geissler, Stephen Whalen
Cray Inc.

Brian Waldecker
AMD Inc.

Jonathan Carter, David Skinner, Helen He, Harvey Wasserman, John Shalf
NERSC Division
Lawrence Berkeley National Laboratory

Hongzhang Shan, Erich Strohmaier
Computational Research Division
Lawrence Berkeley National Laboratory

March 7, 2007

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Understanding and Mitigating Multicore Performance Issues on the AMD Opteron™ Architecture

Abstract

Over the past 15 years, microprocessor performance has doubled approximately every 18 months through increased clock rates and processing efficiency. In the past few years, clock frequency growth has stalled, and microprocessor manufacturers such as AMD have moved towards doubling the number of cores every 18 months in order to maintain historical growth rates in chip performance. This document investigates the ramifications of multicore processor technology on the new Cray XT4™ systems based on AMD processor technology. We begin by walking through the AMD single-core and dual-core and upcoming quad-core processor architectures. This is followed by a discussion of methods for collecting performance counter data to understand code performance on the Cray XT3™ and XT4™ systems. We then use the performance counter data to analyze the impact of multicore processors on the performance of microbenchmarks such as STREAM, application kernels such as the NAS Parallel Benchmarks, and full application codes that comprise the NERSC-5 SSP benchmark suite. We explore compiler options and software optimization techniques that can mitigate the memory bandwidth contention that can reduce computing efficiency on multicore processors. The last section provides a case study of applying the dual-core optimizations to the NAS Parallel Benchmarks to dramatically improve their performance.[1]

Introduction

Recent trends in microprocessors have shown a decreased rate of growth in clock speed as a result of having approached the power and thermal limits of current chip technology. As a result, additional computing power is being added to CPUs in the form of multiple compute cores operating at roughly constant clock rates. While the impact of multi-core parallelism at the commodity/desktop level is modest, the HPC sector, which needs regular significant increases in available computing power, is confronted with concurrencies that race upward in lieu of faster compute elements. Today's supercomputers harness tens of thousands of cores, and those on the drawing board are built from millions of cores. These architectures are implicitly betting that additional cores can be efficiently used. It is therefore important to understand at the level of HPC applications how these many thousands of tasks can make good use of each core.

Multicore computing is a paradigm shift at least as dramatic as the transition from vector platforms to MPPs. Whereas the transition to MPPs required careful examination of how best to use local versus remote (off node) resources, the transition to multicores requires examination of how best to share resources within and connected to the CPU. Multiple compute cores on a single CPU also introduce an additional layer in the hierarchy of parallelism used in scientific

[1] AMD, Opteron, ACML, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Cray, XT3, XT4, CrayPAT, and LibSci are trademarks of Cray Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

codes. While this presents some new opportunities, it also presents new challenges in the form of inter-core resource conflict and contention.

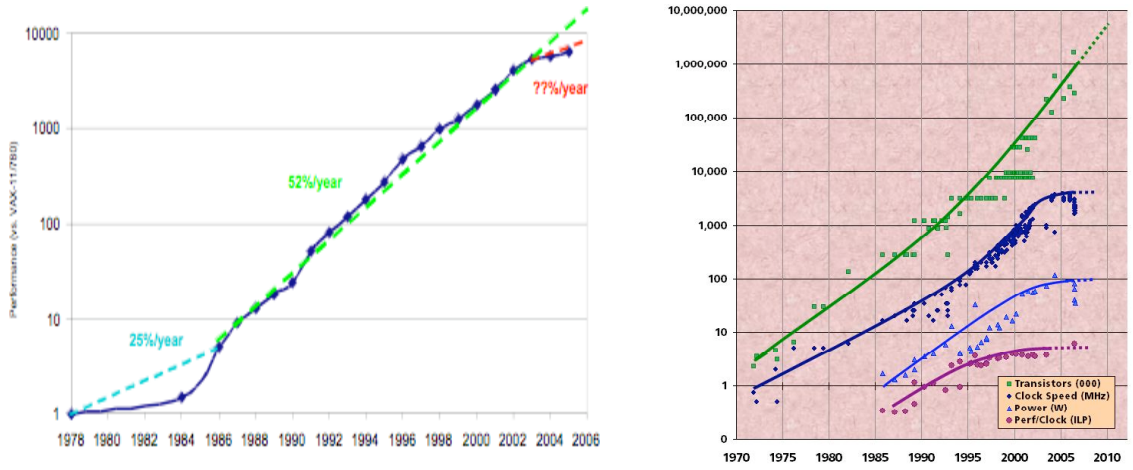


Figure 1: Recent trends in microprocessor performance and architecture. The figure on the left shows the average SPEC-Int performance over the past three decades (courtesy of David Patterson, from Patterson & Hennessy Vol. 4). The figure on the right shows that while the number of transistors that can be packed onto a chip continues to improve, all other traditional performance metrics (clock rate, ILP and power density) are flat-lining (courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith). The industry response is to move to multicore chips.

The microprocessor industry's move to multicore is well under way, and for many reasons it is inevitable. Figure 1 shows recent trends in microprocessor performance. The diagram on the left shows the past three decades of exponential performance improvements for individual microprocessor cores as measured by SPEC-Int. However, starting in 2003, that growth has tailed off dramatically. The chart on the right shows that Moore's Law, which says silicon lithography improvements will enable 2x more transistors to be fit onto a chip approximately every 18 months, is still alive and well. However, all of the traditional sources of processor performance improvement (instruction-level parallelism [ILP], clock speed, and increased power density that results from pushing up the clock speed) are all flat-lining around the same time that the graph on the left shows departure from the 52%/year SPEC benchmark scaling. This indicates that using more transistors per chip to boost performance of individual CPUs, either by clock frequency scaling or increased ILP, is no longer effective. Consequently, the industry response is to move to multiple cores as their primary strategy for maintaining the per-chip performance improvements that have, until recently, matched the rate of Moore's law lithography improvements.

Since the move to multicore processors is inevitable, it is important to examine the consequences of this architectural change for application performance and to understand how to mitigate the performance bottlenecks that arise from these architectural choices. This document focuses on the performance bottlenecks that may arise when using multi-core AMD Opteron™ sockets. While most of the data herein are have been gathered on dual core systems, we expect quad and higher core systems may further constrain available memory bandwidth per core. Given the current and likely continued trends in super-scalar systems, it is important that the application programmer understand the details of the imbalance of the system and techniques that can be used to restructure applications to more effectively utilize the caches and translation lookaside

buffers (TLB), and to take advantage of streaming SIMD extension (SSE) instructions to achieve the highest possible performance.

The first sections of the report document the architecture of the Opteron™ socket, concentrating on those features of the architecture that must be understood to avoid memory contention. We then describe the performance tools available to obtain hardware counter data and how that can be used to better understand how the application is using the cores. Next, we use performance counter data to examine the effects of multicore AMD Opteron™ processors on the efficiency of the NERSC-5 procurement benchmarks, the NAS kernels run in serial mode, and finally some microbenchmarks (STREAM and Apex-MAP) that isolate the source of performance bottlenecks. The last section describes how to optimize codes to improve multicore performance, including compiler options and techniques for restructuring an application to be more cache-friendly. In that section we will also discuss the use of SSE instructions, which while not particular to dual-core issues, are an important way to achieve higher performance on the socket.

AMD Opteron™ 1000, 2000, and 8000 Series Multicore Processor Architecture (also known as AMD Rev F or AMD NPT Family 0Fh)

Cache Organization

The Opteron™ processors have level 1 instruction (L1I) and data (L1D) caches. There is a level 2 cache (L2) that can hold either instructions or data. The L1 caches and L2 cache are exclusive of each other. This means, in the case of data for example, an item of cached data will not be in both the L1D and the L2 caches at the same time. This allows the combined L1 and L2 caches to hold more data but may require more write-backs of data from L1 into L2 than in a design where the L1 contents are a subset of the L2 contents.

The L1 data cache is 64 KB in size and organized with 64 byte lines and is two-way set associative. Virtual address bits 14:6 determine which set is selected, and each of the two ways (i.e., lines) of this set is checked for a hit.

The L2 cache is 1 MB in size, 16-way set associative, and functions as a *victim cache.* This means it is filled with data that was previously in the L1 cache but was evicted as a result of the L1 cache replacement policy. An exception to this is the hardware prefetcher that speculatively prefetches data into the L2 based on detection of access patterns. The most common scenario, and that most amenable to support from hardware and software assistance, is one where the program is sequentially striding through the cache lines of the array. In contrast to the hardware prefetch mechanism, using software prefetch instructions—`prefetchnta` and `prefetcht0/1/2` —will result in data being prefetched into the L1 cache. The rationale of these instructions is that explicitly issued software prefetches should be treated as less speculative and more likely to be used, therefore fetching into the L1 cache will be more optimal. The `prefetchnta` instruction biases the prefetched data to get evicted before other data as it means the data is non-temporal in nature.

For many applications the hardware prefetch mechanism should suffice, but for others that may access multiple streams of data, the software prefetch instructions allow explicit prefetching of

some or all streams into L1 cache. Many compilers provide tuning flags that give the compiler strong hints for utilizing prefetch instructions during code generation.

The L1 data cache is indexed using virtual address bits, but the tags are comprised of physical address bits. This allows indexing to be done before translation has completed. Due to the size of the 64 KB L1 cache, there are not enough untranslated bits (bits that fall through directly from virtual to physical address) available to index the cache using physical address bits only. The practical implication of this is that virtual to physical mappings can affect how data is placed in the cache and therefore affect performance.

Under Linux (or any OS that lazily allocates pages only when touched), there is a possibility that using translated address bits to index a cache can result in lost performance due to subtle interactions between the OS behavior and the cache design. An important rule of thumb for programmers is to initialize all data before reading it. This may sound obvious and is good programming practice, but there have been cases where data was assumed to be initialized to zero and read without initializing it to zero. However, under Linux, uninitialized data usually results in multiple virtual address pages being mapped to the same special physical page (called the *zero page*). Upon the first write to such a virtual page, Linux's *copy-on-write* mechanism causes a new physical page to be allocated and zeroed. Without such copy-on-write allocation occurring, accesses to different virtual pages that map to the same physical page can result in cache thrashing and poor performance. Since the L2 serves as a victim cache, the way in which a cache line would move from one index set to another set in the L1 (one form of thrashing) would be to get evicted to the L2 and then brought back into the L1.

A more subtle caveat has to do with each L1 cache line (64 bytes) being implemented in an eight-bank interleaved fashion. This means there are eight *banks* in the L1, and bits 5:3 of the virtual address select the bank. So a 64 byte cache line is really laid out evenly across the eight banks, with 8 bytes in each bank. A *bank conflict* can occur if two virtual addresses have different indices (bits 14:6) but the same bits for their bank (bits 5:3). In this case, only one load per cycle from L1 will be allowed rather than two per cycle if going to different banks. Practically speaking, if a loop appears to be having fewer loads per cycle than expected (possibly using the hardware performance counters), examination of the virtual addresses loaded in the loop may reveal such a conflict.

Besides the possibility of a cache bank conflict reducing performance as described above, the offsets between array addresses can, in rare instances, result in pathologically bad scenarios where only a portion of the cache gets effectively used. This can usually be solved by adding some padding to the suspect arrays, either by explicitly sizing them larger or using compiler flags to direct additional padding to be generated.

Translation Lookaside Buffers

The translation lookaside buffers (TLBs) provide a faster method of translating virtual addresses to physical addresses without requiring accessing the multiple levels of page tables. There are two levels of TLBs. The level 1 TLB (L1 TLB) can hold 40 address translations. Of these 40, 8 are for 2 MB *large* pages and 32 are for 4 KB *regular* pages. The level 2 TLB (L2 TLB) can hold 512 translations for 4 KB pages only. Thus the TLB reach is 8 * 2 MB = 16 MB for large

pages and 512 * 4 KB = 2 MB for regular pages. In large-page mode, two of the TLB pages are pinned by the OS, so in practice, only six of the pages are available to applications (12 MB of coverage).

The L1 TLB is fully associative, and the L2 TLB is four-way set associative. If a program's data will fit within the 16 MB reach of the L1 TLB large pages, using large pages may be of benefit. However if the data exceeds that size, using small pages may actually perform better due to the L2 TLB and actual access patterns. Even codes that have relatively low page miss rates in small page mode can suffer when using large pages because so few pages are available. For instance, a code that walks linearly through seven or more distinct arrays will also thrash a six-entry TLB. Codes that rely on indirect access, such as row-compressed sparse matrix solvers or codes that use explicit representations of unstructured meshes, will consume one page per level of indirection. In those cases, the smaller page sizes may offer a lower miss rate despite smaller coverage of the memory, due to the larger numbers of pages available to the application. The 1 GB page sizes available in the quad-core system may alleviate this problem, but the 1 GB pages may prove impractical if an entire page (1 GB of the node memory) must be owned by the OS image (that would make 1 GB of the node memory inaccessible to applications).

Data Prefetch

Both hardware and software based prefetch mechanisms are supported in the Opteron™ processors.

The hardware prefetch mechanism detects and prefetches cachelines from main memory into the L2 cache. Detection of a stream for prefetching is based on detecting a pattern of increasing or decreasing successive cachelines. It takes two contiguous cacheline fetches, l and l+1, that miss in the L2 to trigger the prefetch mechanism. Upon detection of this pattern, the prefetch hardware will initiate a fetch of the cache line l+3. Accesses in strides larger than a single cacheline will not trigger the hardware prefetcher. Any sequence of L1 misses to successive cachelines can trigger the hardware prefetcher if tracking resources are available, including software prefetches that miss in the L1. The `prefetchnta` is an exception to this; since its intent is to avoid cache pollution, `prefetchnta` will not trigger the hardware prefetcher.

The facilities for software prefetching consist of

- prefetching via loads (standard `mov` instruction)
- `prefetch, prefetcht0, prefetcht1,` and `prefetcht2` instructions
- `prefetchw` instruction
- `prefetchnta` instruction

The `prefetch` and `prefetcht0/t1/t2` instructions all do the same thing on current generations of processors. These software prefetch instructions will bring data into the L1 data cache rather than the L2 as the hardware prefetch mechanism does. The `t0/t1/t2` versions of the `prefetch` instructions are implementation dependent and may in future designs indicate what level of the cache hierarchy should serve as the destination for prefetched data. Software prefetch is always honored unless a load, store, or hardware prefetch to the same cacheline overrides it. There can be up to eight outstanding L2 misses at a time per core, regardless of whether they are due to load, store, or software prefetch instructions.

The `prefetchw` instruction provides a hint to the processor that the prefetched data is likely to be modified later and it may be better to bring it into the cache in exclusive mode rather than shared so that an upgrade to modified state can be done more efficiently. Currently this instruction is equivalent to a regular `prefetch` instruction, however.

The `prefetchnta` indicates the data being prefetched is non-temporal in nature and unlikely to be reused prior to being evicted from the cache. In this case, the data is still brought in to the L1 cache but is marked in a way to indicate it is not recently used and thus will be favored for replacement before other cachelines in its congruence class. This is useful for data that is not being written only. For data that is only being written, such as large array initializations or the destination of a memory block copy, it is recommended that the processor's write combining store mechanism be used.

Some general rules of thumb and remarks related to software prefetching are:

- Software prefetching six to eight cachelines ahead is typically good, as the latency of a prefetch instruction that misses in the L2 is on the order of 100 cycles. The latency for a software prefetch that hits in the L2 is on the order of 13 cycles.
- Try to have at least 100 cycles worth of computation in loop iterations between software prefetch instructions.
- Try to unroll loops so that each iteration works on an entire cacheline at a minimum. Multiple cachelines per iteration should be OK as well.
- Avoid multiple software prefetches to the same cacheline.
- Neither hardware nor software prefetches will be allowed to generate page fault exceptions. However, a prefetch that misses in the TLB will cause a TLB fill to be initiated.

For data that is being written and consists of one or more entire cachelines that are not likely to be used again soon, it is recommended that write combining store instructions be utilized. These are the `movnt` versions of the `mov` instructions. They will utilize special write combining store buffers in the processor and are flushed to main memory when an entire cacheline-sized buffer fills up. This reduces cache pollution for data unlikely to be needed again soon and enables more efficient write transactions into main memory. This will be faster than a `prefetchw` instructions since the cacheline is not first read from memory. Write combining instructions are not recommended, however, if the memory region being written to has been marked a "write-combining" in the processor's memory type range registers (MTRRs) or page attribute tables (PAT). One other caveat is that use of write combining stores may require a fence instruction be placed after the last such store in some cases for program correctness. Appendix B of the document "Software Optimization Guide for AMD64 Processors" describes more details about how the write combining store facility is implemented.

Table 1 shows what is generally the best prefetch strategy based on the dataset size and type of accesses. In this table, `movnt` refers to any of the various flavors of non-temporal `mov` instructions (integer, float, packed, etc.). The term `prefetch` refers to any of the software `prefetch/t0/t1/t2` instructions.

In the cases of sequential read-only and sequential read-write, it may be beneficial to issue two prefetch or two `prefetchw` instructions before entering a loop to jump start the hardware prefetcher. This may also cause subsequent software prefetches inside the loop to hit in the L2. If so, a software "prefetch-ahead" distance of less than the six to eight cachelines mentioned earlier may be preferable.

Table 1. Summary of prefetch strategies.

Data	Less than ½ L1 size	Less than ½ L2 size or of unknown size		Greater than ½ L2 size
		Reused	Not Reused	
Read only	`prefetch or prefetchnta`	`prefetch`	`prefetchnta`	`prefetchnta`
Sequential read only	`hwprefetcher + prefetch`	`hwprefetcher + prefetch`	`prefetchnta`	`prefetchnta`
Read-write	`prefetchw`	`prefetchw`	`prefetchnta`	`prefetchnta`
Sequential read-write	`prefetchw`	`prefetchw`	`prefetchnta`	`prefetchnta`
Write only	`prefetchw`	`prefetchw`	`movnt`	`movnt`
Sequential write only	`hwprefetcher + prefetchw`	`hwprefetcher + prefetchw`	`movnt`	`movnt`

Upcoming Quad-Core Changes

The upcoming quad-core processors provide a number of enhancements to improve performance. Highlights recently disclosed include:

- Support for an additional large page size of 1 GB. (4 KB and 2 MB are still supported.)
- A fully associative 48-entry L1 TLB where all 48 entries may be used for any of the three pages sizes (4 KB, 2 MB, 1 GB).
- A L2 TLB that can hold 512 4-KB page translations or 128 2-MB page translations.
- A 2 MB L3 cache is shared by the four cores and functions as a victim cache for each core's private L2 cache. The L2 caches are 512 KB per core. L1 caches remain at 64 KB and private to each core. The L3 is engineered for future expansion.
- Dual channel memory is now *unganged* or two independent channels. This should improve concurrency in memory accesses and improve the chances of hitting in an already open DRAM bank.
- Full 128-bit-wide floating point unit (4 flops/cycle).
    - This makes it even more important to use full-width SSE2 instructions as opposed to the "upper/lower" half varieties if data alignment is known. In other words, instructions such as `mulpd, addpd, movapd,` etc. are preferable to ones such as `mulsd, addsd, movhpd, movlpd,` etc. That is, aligned data and vectorizable code become even more beneficial.
    - Similarly, if the upper portion of a 128-bit XMM register does not matter, it is better to use `movsd` and `movss` to load scalar values into the lower portions of the XMM register instead of `movlpd` or `movlps`.
    - For unaligned data, it may be better to use `movupd` or `movdqu` for loads and `movhpd` or `movlpd` for stores.
- Two 128 bit loads from L1D per cycles (versus current 2 × 64 bit loads per cycle).
- Support for 48 bits of physical address (256 TB). Current Opteron™ processors support 40 bits of physical address.

- A new hardware prefetch buffer in the memory controller that can track positive, negative, and non-unit strides. It will not speculatively fill the L3, L2, or L1 caches, but instead prefetches into dedicated prefetch buffers in the memory controller.
- A new data cache prefetcher that is adaptive and detects positive or negative non-unit stride patterns (e.g., non-adjacent cachelines) and adapts the prefetch distance based on prefetching success. Further, the prefetched data now gets brought into the L1 rather than the L2.
- `Prefetchw` software prefetch instruction is now implemented separate from `prefetch` (`prefetchw` denotes intent to modify and provides a hint to set the cacheline state according to the MOESI protocol appropriately).

General Source Code Level Optimizations

While many optimizing compilers will detect and perform the following optimizations on their own, programming style can aid the compiler in detecting a potential optimization or knowing it is safe. A few ideas along these lines are:

- Declare functions that are not used outside the file in which they are declared as "static" functions. This will force internal linkage and may aid in function inlining versus the default external linkage.
- If it is not permissible to allow expression reordering (particularly floating point) by the compiler throughout the entire scope of a compilation unit, it may be useful to explicitly express parallelism in sections of code where the programmer knows reordering (e.g., reassociation) to be safe. An example of this would be a loop that computes a number of partial sums from different segments of an array rather than a single summation that is calculated by summing the array from start to end in order.
- Explicit calculation of common subexpressions and storing the value in a local variable may be useful in code where the compiler fails to detect the opportunity to compute a partial result and reuse it across loop iterations. A good example of this is a stencil operation in a 3D grid data structure where the nine elements at locations [i+1, j+/-1, k+/-1] become the nine elements at locations [i, j+/-1, k+/-1] in the next loop iteration (when i = i+1). In such cases, explicit calculation and copying of a partial result between temporary variables can help the compiler generate better code.
- Many compilers will pad arrays and structures to improve the alignment of their elements, but if the compiler is unable or fails to do so, the programmer might try doing so explicitly in the declaration of those data structures. In general, declaring elements of C `structs` in order from largest to smallest is best.
- The same rule of thumb applies to local variable declarations in functions. Declaration in order from largest to smallest usually improves the chances of favorable alignment and layout in memory.
- For parallel applications, avoid having separate data items fall into the same 64 byte cacheline since this can create a false sharing condition that causes the cacheline to thrash between the unshared caches of each core. Explicit padding by the programmer or flags to provide hints to the compiler are possible ways to alleviate this.
- Avoid repeatedly dereferencing pointers in a function. Particularly for C code where the compiler may not know at compile time whether different pointers will alias (point to) the same address at runtime, this can result in many more loads and stores than desired. Frequently compilers will provide flags to tell the compiler it is safe to assume no such

aliasing exists. Barring that, or just to make it more explicit, a programmer might consider dereferencing the pointer once at the start of the function, placing a working copy into a local variable, and copying it back at the end of the function.

Methodology for Gathering Performance Data

Except where otherwise noted, we analyzed benchmark performance on a dual-core Cray XT3™ system (Jaguar at ORNL) and a prototype Cray XT4™ system using data collected from the AMD Opteron™ hardware performance counters. CrayPAT™ was used to instrument the benchmark codes to gather runtime performance statistics for our evaluation. The following section describes how to use CrayPAT™ and the basic methodology used to gather these statistics.

Using CrayPAT™ to Gather Runtime Statistics

The Cray Performance Analysis Tools (CrayPAT™) were used to record the hardware efficiency of the application benchmarks. These tools are provided on all Cray XT3s™ and provide a simple interface to hardware counter and other instrumentation data. Users are able to load the CrayPAT™ environment model, rebuild their code, instrument the executable with the `pat_build` command, and receive reports from their application runs. CrayPAT™ captures performance data without the need for source code modifications. Event tracing was used to gather data during these experiments. Tracing records events at function entry and exit points, rather than interrupting execution periodically to capture events. Event tracing is added to an executable by the `pat_build` command, provided that the `craypat` module was loaded when the source was compiled. Loading the `craypat` module for compilation is necessary to ensure that the needed debugging headers remain in the object files. Because CrayPAT™ is able to leverage hardware-level performance counters, by means of the Performance API (PAPI) library, only minimal overhead is added to the user's application.

Because application tracing can produce enormous amounts of data, CrayPAT™ uses runtime summarization by default. Rather than storing the data for each function on each process, CrayPAT™ instead stores the data in aggregate. Summarization also ignores function parameter and return values, as well as stack information. While all of this information is valuable, especially when debugging program performance, it is unnecessary for the purposes of this report. The aggregate hardware counter data provides a representative look at memory and cache performance across the entire application execution.

The Opteron™ processor architecture has four 48-bit hardware event counters, each of which can be used to monitor a variety of different events. For this reason, CrayPAT™ defines nine sets of commonly used hardware counters, which are grouped by function. Two groups of hardware counters (groups 1 and 2) were used in these experiments to understand the cache and memory performance. The first group records total floating point operations, L1 data cache accesses, L1 data cache misses, and TLB misses. The second group records L1 and L2 data cache accesses, cache refills from L2 cache, and cache refills from system memory. Because MPI use can negatively influence each of the recorded metrics, the executables were instrumented in such a way that MPI functions were removed from the metrics. The results cited in Table 2, therefore, only include user code.

Table 2. Explanation of hardware counter events.

PAT_HWPC_EVENT_SET 1 : (FP, LS, L1 misses, & TLB misses)	
PAPI_FP_OPS	Floating point operations
PAPI_L1_DCA	Accesses to level 1 data cache
DC_MISS	Total level 1 data cache misses
PAPI_TLB_DM	Translation lookaside buffer misses
PAT_HWPC_EVENT_SET 2 : (L1 & L2 data accesses & misses)	
PAPI_L1_DCA	Accesses to level 1 data cache
DC_L2_REFILL_MOESI	Total data cache refills from level 2 cache to level 1 cache.
DC_SYS_REFILL_MOESI	Total data cache refills from system memory/level 2 data cache misses
BU_L2_REQ_DC	Accesses to level 2 data cache

Example Application Report

As an example of how each program was analyzed, this section will show and interpret reports for a single application. As was already explained, each application was profiled so that user and MPI functions are considered separately. In the report, this creates three major data sessions: Totals for program, User, and MPI. As the names imply, the Totals section is the complete data from the program execution, User includes only user functions, and MPI includes only MPI functions. Within the User and MPI sections, there is a breakdown of the most pertinent functions from each group. Below is an example of a program totals section for a report using hardware counters group 1.

```
Totals for program
-----------------------------------------------------------------------
 Time%                        100.0%
 Cum.Time%                      100.0%
 Time                       6732.607664
 Calls                      162397731109
 PAPI_TLB_DM       659.911M/sec  4441480049284 misses
 PAPI_L1_DCA      86841.249M/sec 584478099205200 ops
 PAPI_FP_OPS      14501.875M/sec  97603713650150 ops
 DC_MISS         1193.517M/sec  8032871839125 ops
 User time        6730.420 secs  17499092506162 cycles
 Utilization rate                100.0%
 HW FP Ops / Cycles                5.58 ops/cycle
 HW FP Ops / User time 14501.875M/sec  97603713650150 ops     4.4%peak
 HW FP Ops / WCT    14497.163M/sec
 Computation intensity               0.17 ops/ref
 LD & ST per TLB miss              131.60 ops/miss
 LD & ST per D1 miss               72.76 ops/miss
 D1 cache hit ratio                98.6%
 % TLB misses / cycle               0.4%
```

Each report section shows how much time was spent in that section and the percentage of total execution time. For the Totals section, this is obviously 100% of the execution time. Below these metrics is the total number of function calls. The next four metrics are the hardware counters included in counter group 1, `PAPI_TLB_DM` (translation look-aside buffer misses), `PAPI_L1_DCA` (L1 cache accesses), `PAPI_FP_OPS` (floating point operations), and `DC_MISS` (data cache misses), and their counter values. It is important to remember that these values are the aggregate from every MPI task and not from a single CPU. The next item is another measure of execution time,

this one derived from the hardware performance counters. Below that are several useful metrics derived from the counter data. For example, `HW FP Ops / User time` is the common megaflop count for the program, in this case 14.5 GF or about 4.4% of the peak performance. The remaining derived metrics will be explained shortly. Below is a USER section from a report for hardware counter group 2.

```
USER
------------------------------------------------------------------
 Time%                        100.0%
 Cum.Time%                    100.0%
 Time                      6724.668036
 Calls                     162396953744
 PAPI_L1_DCA       86890.056M/sec  584116763286676 ops
 DC_L2_REFILL_MOESI    1191.189M/sec   8007744664830 ops
 DC_SYS_REFILL_MOESI    155.048M/sec   1042309740361 ops
 BU_L2_REQ_DC       1256.872M/sec   8449296500998 req
 User time        6722.481 secs  17478450934102 cycles
 Utilization rate                  100.0%
 L1 Data cache misses   1346.237M/sec   9050054405191 misses
 LD & ST per D1 miss                64.54 ops/miss
 D1 cache hit ratio                 98.5%
 LD & ST per D2 miss                560.41 ops/miss
 D2 cache hit ratio                 87.7%
 L2 cache hit ratio                 88.5%
 Memory to D1 refill    155.048M/sec   1042309740361 lines
 Memory to D1 bandwidth  9463.401MB/sec  66707823383104 bytes
 L2 to Dcache bandwidth 72704.398MB/sec 512495658549120 bytes
```

The example above is similar to the previous one, except it only includes data from user functions. This report is especially useful when MPI functions are considered separately, as they were in this example. Notice that for this report the specific hardware counters have changed. They are now `PAPI_L1_DCA` (L1 data cache accesses), `DC_L2_REFILL_MOESI` (L1 refills from L2 cache), `DC_SYS_REFILL_MOESI` (L2 refills from system memory), and `BU_L2_REQ_DC` (L2 data cache requests). This will, of course, change the derived metrics.

These two different reports include several important metrics derived from the raw hardware counter data. The *computational intensity* metric shows how many operations are executed for every memory reference. If this number is large, then a lot of computation is done for every trip to memory, but if it is small, then the program is not reusing the items it fetches from memory. In the Totals report above, the computation intensity is 0.17, which means that for all the time spent fetching from memory, only one-fifth of that time is being used for calculations. It is desirable to have a computational intensity above 1, and the larger, the better. The `LD & ST per TLB miss` and `LD & ST per D1 miss` are similar metrics, showing the number of load and store operations per TLB and L1 data cache misses. As with the computation intensity metric, larger numbers are better.

The next two metrics to focus on are the `D1 cache hit ratio` and `% TLB miss/cycle`. The `D1 cache hit ratio` shows how many memory references were found in cache. The hit ratio is affected by cache-line reuse and prefetching, so it can and should be a high percentage. It is impossible to have 100% of all data references hit cache, but a well-written code should be able to achieve a cache hit ratio very near 100%. This metric can usually be improved over the most naïve implementation by using techniques described below. The opposite is true for the

percentage of TLB misses per cycle. Because TLB misses are an extremely expensive operation, it is important that the program have as few TLB misses as possible. This metric essentially shows the chance that any given operation results in an expensive TLB miss. It is critical to program performance to keep this metric below 1%. The Users example contains several more load/store operations per miss and miss ratio metrics, which should be interpreted as already explained, and also cache and memory bandwidth measures.

Application Performance on AMD Multicore Processors

The first step in understanding and mitigating multicore bottlenecks is determining whether we actually have a problem to start with! In this section, we examine the impact of memory bandwidth contention on the performance of application kernels (represented by the NAS Parallel Benchmarks run in serial mode) and full applications. Not all applications suffer equally from the move to dual core. Next, we use STREAM and APEX-MAP microbenchmarks to show that memory bandwidth contention is the primary source of performance loss when moving from single- to dual-core systems. The conclusion is that many applications are not limited by memory bandwidth and that remedies for memory bandwidth contention will be beneficial only for those codes that are provably dependent on memory bandwidth.

NERSC-5 SSP Applications

NERSC uses the SSP (Sustained System Performance) performance metric to assess the performance of HPC systems. The SSP uses a set of applications derived from the NERSC workload to predict the effective *sustained* performance delivered to real scientific workloads rather than focusing on the *peak* performance that is reflected in the LINPACK benchmark. In this section, we look at the effect of dual-core processors on the processing efficiency of the NERSC SSP applications.

The medium NERSC-5 application benchmarks were profiled on dual-core XT3™ and XT4™ hardware using CrayPAT™, as described in the previous sections. Since we are only interested in single-node performance for this report, the results exclude the performance characteristics of the MPI sections. All applications except CAM were run on 64 processors, with CAM running on 54. The applications were run in both serial and virtual node mode (single and dual core) and also using small and large pages.

Before discussing the applications in detail, we note that the hardware performance counter data collected does not indicate a clear source of the dual core penalty in regards to the cache and TLB subsystems. That is, the operations per TLB or cache miss are virtually unchanged on going to dual core in every case. The counter data does have a predictive value, in that applications with good ratios show little performance decrease on going to dual core. This is only to be expected given the discussion in previous sections, and our conclusion is that if each core is using less of a shared resource such as memory bandwidth, less contention can occur.

Regarding the merits of using small or large pages, however, the hardware performance counter data is conclusive. In general, runs utilizing large pages show an increased number of TLB misses and decreased performance on both single and dual core. Again, this is clearly explained from our understanding of the hardware.

MILC

The benchmark code MILC represents part of a set of codes written by the MIMD Lattice Computation (MILC) collaboration to study quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics. Strong interactions are responsible for binding quarks into protons and neutrons and holding them all together in the atomic nucleus. MILC performs simulations of four-dimensional *SU*(3) lattice gauge theory on MIMD parallel machines. The test case for the data shown here is for a lattice size of 32^4 with two trajectories of five steps each. MILC uses inlined SSE assembly code for some routines to aggressively perform software prefetching.

From the data in Table 3, we can see that small pages outperform large pages by a slight margin on both the XT3™ and XT4™, due to a modest reduction in TLB misses. The dual core penalty is significant on both the XT3™ and XT4™; MILC has the largest penalty of all the applications studied. Using small pages, MILC runs 44% and 43% slower on the dual-core XT3™ and XT4™ systems, respectively. From the TLB and cache miss ratios, we see very little spatial or temporal locality in the memory references generated by the MILC application. The ratios are uniformly low for both small and large pages. This is to be expected due to the type of scattered memory addressing that is performed in this application. Even so, the ratios seem at odds with the high computational intensity. Significant work has been done to include software prefetch instructions and SSE instructions in many of the kernels, and we conjecture that the effect of these optimizations may be skewing the composite hardware counter results.

Table 3. MILC performance on single-core and dual-core AMD Opteron™ processors using small and large pages.

	Small Pages		Large Pages	
XT3™	**Single**	**Dual**	**Single**	**Dual**
Wall Clock Time	160	230	166	232
Sustained MFLOPS	69370	48402	67138	47976
Percent of Peak	21%	15%	20%	14%
Computational Intensity	2.1	2.1	2.1	2.1
References/TLB Miss	308	309	68	68
References/D1 Cache Miss	16	16	16	16
References/D2 Cache Miss	32	32	31	31
XT4™				
Wall Clock Time	127	181	130	184
Sustained MFLOPS	87840	61482	85447	60538
Percent of Peak	26%	18%	26%	18%
Computational Intensity	2.1	2.1	2.1	2.1
References/TLB Miss	307	308	106	106
References/D1 Cache Miss	16	16	16	16
References/D2 Cache Miss	33	33	33	33

GTC

GTC is a three-dimensional code used to study microturbulence in magnetically confined toroidal fusion plasmas via the particle-in-cell (PIC) method. It solves the gyro-averaged Vlasov equation in real space using global gyrokinetic techniques and an electrostatic approximation. The Vlasov equation describes the evolution of a system of particles under the effects of self-consistent electromagnetic fields. The test case studied here is 10 particles per cell and 2000 time steps.

In Table 4, we can see that small pages outperform large pages by almost 40%. This is the largest effect observed over all the applications studied here. The change in TLB miss ratio on going to large pages explains the very large performance hit; the number of TLB misses increases by a factor of almost 300. From closer inspection of the GTC trace data, the three routines most significant in terms of time consumed all contain loop constructs that address more than eight arrays, causing thrashing of the TLB. The dual core penalty is small, amongst the lowest of all the applications, at 4% slower when run with small pages. From the cache miss ratio, we see a reasonable amount of spatial or temporal locality in the memory references generated by the GTC application.

Table 4. GTC performance on single-core and dual-core AMD Opteron™ processors using small and large pages.

	Small Pages		Large Pages	
XT3 ™	**Single**	**Dual**	**Single**	**Dual**
Wall Clock Time	614	639	851	879
Sustained MFLOPS	71219	68557	51584	49920
Percent of Peak	21%	21%	16%	15%
Computational Intensity	1.17	1.17	1.17	1.16
References/TLB Miss	4858	4853	21	21
References/D1 Cache Miss	44	43	44	44
References/D2 Cache Miss	355	355	206	206

PARATEC

The benchmark code PARATEC (PARAllel Total Energy Code) performs *ab initio* quantum-mechanical total energy calculations using pseudopotentials and a plane wave basis set. Total energy minimization of electrons is done with a self-consistent field (SCF) method. Force calculations are also done to relax the atoms into equilibrium positions. PARATEC uses an all-band (unconstrained) conjugate gradient (CG) approach to solve the Kohn-Sham equations of density functional theory (DFT) and obtain the ground-state electron wavefunctions. In solving the Kohn-Sham equations using a plane wave basis, part of the calculation is carried out in Fourier space, where the wavefunction for each electron is represented by a sphere of points, and the remainder is in real space. Specialized parallel three-dimensional FFTs are used to transform the wavefunctions between real and Fourier space, and a key optimization in PARATEC is to transform only the non-zero grid elements. The test case used as input to collect data is bulk silicon with a unit cell containing 125 atoms, running a single SCF calculation.

Data in Table 5 shows the performance of PARATEC on the XT3™ and XT4™. Interestingly, on the XT3™, small pages lead to marginally better performance, but the situation is reversed on the XT4™. This is because the ratio of TLB misses running with small and large pages changes quite significantly (more than a factor of 4) according to the architecture. The major source of this change is improved large page TLB performance on the XT4™. Given that the processor is identical on both these platforms, we assume that the performance difference is due to a newly installed version of the compiler and ACML library. The dual core penalty is small for both architectures, at 5% slower with small pages. The TLB and L2 cache miss ratios are good for both large and small pages, but the L1 cache miss ratio falls to around the average across all the applications.

Table 5. PARATEC performance on single core and dual core AMD Opteron™ processors using small and large pages.

	Small Pages		Large Pages	
XT3™	**Single**	**Dual**	**Single**	**Dual**
Wall Clock Time	593	622	598	630
Sustained MFLOPS	221864	211696	220223	208938
Percent of Peak	66.8	63.6	66.2	62.8
Computational Intensity	1.51	1.51	1.51	1.51
References/TLB Miss	6659	6670	1325	1309
References/D1 Cache Miss	61	61	61	61
References/D2 Cache Miss	1133	1129	1226	1222
XT4™				
Wall Clock Time	549	572	548	570
Sustained MFLOPS	239915	230774	240621	231337
Percent of Peak	72%	69%	72%	70%
Computational Intensity	1.48	1.48	1.48	1.48
References/TLB Miss	6749	6736	5643	5473
References/D1 Cache Miss	105	105	105	105
References/D2 Cache Miss	1139	1151	1241	1247

CAM

The Community Atmosphere Model (CAM) is the atmospheric component of the Community Climate System Model (CCSM) developed at NCAR and elsewhere for the weather and climate research communities. Although generally used in production as part of a coupled system in CCSM, CAM can be run as a standalone (uncoupled) model as it is here. The NERSC benchmark runs CAM version 3.1 at D resolution (about 0.5 degree) using a finite volume dynamical core.

From the data shown in Table 6, we can see that small pages outperform large pages by a slight margin on the XT3™, although on the XT4™ the difference in performance is almost nonexistent. As with PARATEC, the ratio of TLB misses running with small and large pages changes significantly (more than a factor of 2) according to the architecture. In this case, however, the change is due to both a performance improvement of large page TLB miss rate and

the decrease in small page TLB miss rate on the XT4™. Again, we believe the origin of this change is the upgraded compiler. The dual core penalty is significant, the second largest of all the applications, at 12% slower with small pages on the XT3™. However, this penalty drops to 10% on the XT4™. The TLB and cache miss ratios fall into the middle of the range when comparing all the applications.

Table 6. CAM performance on single core and dual core AMD Opteron™ processors using small and large pages.

	Small Pages		Large Pages	
XT3™	**Single**	**Dual**	**Single**	**Dual**
Wall Clock Time	1733	1937	1806	2002
Sustained MFLOPS	30574	27357	29334	26464
Percent of Peak	10.5	9.4	10.1	9.1
Computational Intensity	0.56	0.56	0.55	0.55
References/TLB Miss	2847	2851	172	183
References/D1 Cache Miss	33	33	33	33
References/D2 Cache Miss	518	513	545	536
XT4™				
Wall Clock Time	1216	1335	1215	1339
Sustained MFLOPS	43584	39691	43599	39573
Percent of Peak	13%	12%	13%	12%
Computational Intensity	0.89	0.89	0.89	0.89
References/TLB Miss	1913	1910	314	317
References/D1 Cache Miss	22	22	22	22
References/D2 Cache Miss	335	330	354	349

MADbench

The benchmark code MADbench is a stripped-down version of MADCAP (Microwave Anisotropy Dataset Computational Analysis Package). This package contains several computational tools developed for analysis of data from cosmic microwave background (CMB) experiments. The goal of these experiments is to extract the wealth of cosmology information embedded in the CMB that is related to the universe at an age of about 400,000 years after the Big Bang. Such experiments typically involve scanning a significant amount of the sky for long periods at very high resolution. The reduction of the resulting datasets, first to a pixelized sky map and then to an angular power spectrum, is extremely computationally intensive. The primary computational challenge is an out-of-core solution to a dense linear algebra problem on distributed matrices. The test case used here is 18,000 pixels with 24 bins operating in single gang mode.

The performance of MADBench is summarized in Table 7. For this application only, on the XT3™, running with large pages slightly outperforms running with small pages. Virtually identical performance between large and small page runs is observed on the XT4™. The dual core penalty is one of the smallest, at 3% slower with large pages on the XT3™ and 2% slower on the XT4™. TLB and cache miss ratios are the best of all the applications on both XT3™ and

XT4™, probably due to the main computational work being matrix-matrix multiplication, which is carried out via an optimized ACML™ routine. Interestingly, TLB misses are vastly improved for large pages on the XT4™. This data point is the only occasion where the ratio of operations per TLB miss is better running with large pages than with small pages.

Table 7. MADBench performance on single core and dual core AMD Opteron™ processors using small and large pages.

	Small Pages		Large Pages	
XT3™	**Single**	**Dual**	**Single**	**Dual**
Wall Clock Time	1318	1336	1248	1291
Sustained MFLOPS	219981	216986	232294	224640
Percent of Peak	66%	65%	70%	68%
Computational Intensity	1.73	1.7	1.72	1.72
References/TLB Miss	7807	7880	3281	3265
References/D1 Cache Miss	122	124	121	121
References/D2 Cache Miss	2481	2348	2922	2880
XT4™				
Wall Clock Time	1248	1272	1236	1263
Sustained MFLOPS	230868	226415	233129	228185
Percent of Peak	69%	68%	70%	69%
Computational Intensity	1.67	1.67	1.67	1.67
References/TLB Miss	7911	7918	12521	12721
References/D1 Cache Miss	122	122	121	122
References/D2 Cache Miss	2407	2399	2989	2968

GAMESS

The benchmark code GAMESS (General Atomic and Molecular Electronic Structure System) performs various *ab initio* quantum chemistry simulations. Several kinds of SCF wavefunctions can be computed with correlation. Correlation corrections to these SCF wavefunctions include configuration interaction, second order perturbation theory, and coupled-cluster approaches, as well as the density functional theory approximation. A variety of molecular properties, ranging from simple dipole moments to frequency-dependent hyperpolarizabilities may be computed. Many basis sets are stored internally, together with effective core potentials, so all elements up to radon may be included in molecules. The two benchmarks used here test DFT energy and gradient, RHF energy, MP2 energy, and MP2 gradient.

From the data in Table 8, we can see that small pages outperform large pages by about 10%. The dual core penalty is minimal, one of the smallest of all the applications at 2% slower with small pages on the XT3™. The TLB and cache miss ratios are about average when compared across all the applications studied, except the TLB miss ratio for the small pages runs which is relatively small.

Table 8. GAMESS performance on single core and dual core AMD Opteron™ processors using small and large pages.

	Small Pages		Large Pages	
XT3 ™	Single	Dual	Single	Dual
Wall Clock Time	6573	6732	7087	7334
Sustained MFLOPS	14976	14643	14310	13645
Percent of Peak	5%	4%	4%	4%
Computational Intensity	0.17	0.17	0.17	0.17
References/TLB Miss	131	131	109	109
References/D1 Cache Miss	72	73	73	73
References/D2 Cache Miss	561	561	566	566

Analysis of NERSC-5 SSP

Figure 2 shows the impact of moving from single-core to dual-core processors on application performance. Overall, the majority of N5 SSP applications suffered very little from the move to dual-core processors. On average, the performance penalty of moving from single- to dual-core processors was on the order of 10%. This indicates that memory bandwidth is unlikely to be the bottleneck for application performance. MILC suffered the worst effects from the move to dual-core processors. In that case, the dominant portion of the MILC kernel is hand-coded in assembly, so it makes optimal use of memory bandwidth. It is interesting to point out that, except for MILC, the difference in performance between using large pages vs. small pages was actually greater than the performance impact of moving to dual cores. The importance of TLB page size was examined in detail in the discussion of the Opteron™ architecture above.

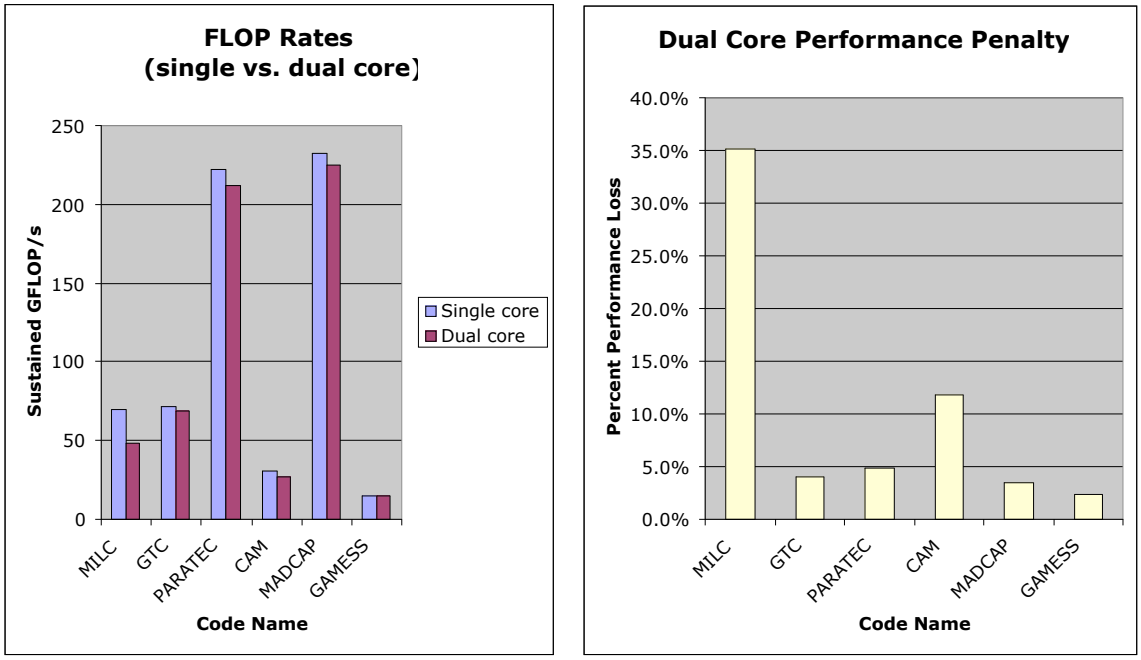


Figure 2. NERSC SSP Performance on single-core vs. dual-core AMD Opteron™ processor. Except for MADCAP, small-page performance results were used. The average performance penalty for moving from single to dual core was 10.3%.

NAS Parallel Benchmarks

Several of the NAS Parallel Benchmarks—CG, MG, SP, LU, and BT—were run in serial mode on the dual core for examining a worst case scenario comparison of packed and unpacked execution. The addressing characteristics of the kernels are as follows:

- BT: 5 x 5, stride
- SP: 5 x 5, stride
- LU: 5 x 5, stride
- MG: Contiguous
- CG: Indirect

Methodology

The latest 3.x.x version of the serial versions of the NAS benchmarks were built for the Class B problem size using `-O3` and `-fastsse` options using the Portland Group (PGI) compilers. Each applet/kernel is executed in a serial fashion, from a top level MPI wrapper with two MPI ranks. In order to run in this fashion, the code had to be slightly modified to facilitate our study. The modification is summarily:

```
<main program declarations>
mask=0
read(10,*)cpuflag
if (cpuflag.eq.1)mask(0)=1
if (cpuflag.eq.2)mask(1)=1
if (cpuflag.eq.3)mask(0)=1
if (cpuflag.eq.3)mask(1)=1

if (mask(my_mpi_rank).eq.1)then
   <original serial code>
endif
call MPI_Barrier(MPI_COMM_WORLD, ierr)
call MPI_Finalize(ierr)
<original program end>
```

Each of the applications was executed on one dual core socket with two MPI ranks for values of `cpuflag` of 1, 2, and 3, yielding performance data for execution on core 0 and core 1 separately (single core runs) and core 0 and 1 simultaneously (dual core runs). Runs were also performed replacing the final barrier operation with an `mpi_abort` while parking the non-executing core in a tight loop. No timing discrepancy was found between these two methods, indicating negligible effect from idle tasks parked in the barrier operation.

The data presented was generated on a Cray internal machine, "Seal," running Catamount v1.5.x using both 4 KB page size (small pages) and 2 MB pages (large pages) selected via the `yod` command. Seal is a Cray XT4™ platform with the following node attributes:

- **Sockets:** 1
- **Processor:** AMD Opteron™ 1218 (Rev F)
- **Frequency:** 2.6 GHz, peak floating point performance of 5.2 Gflops
- **Memory:** 667MHz DDR2, peak bandwidth of 10 GB/s (8 GB/s STREAM)

We note in contrast to the NERSC-5 SSP application tests, these NAS benchmarks are configured so that the tasks do not communicate with one another. This enables us to isolate our performance study to scalar performance without consideration of message-passing performance.

Execution times for the applications obtained are given in Table 9. In the table, single core runs are provided for each of the two cores independently, while dual core run timings are shown for each core from the same run.

Table 9. Execution times (seconds) for the NAS benchmarks.

	Single Core Runs				Dual Core Runs				Averages			
	Large Pages		Small Pages		Large Pages		Small Pages		Large Pages		Small Pages	
	core 1	core 2	core 1	core 2	core 1	core 2	core 1	core 2	S/core	D/core	S/core	D/core
BT	431.5	431.5	406.0	405.9	472.2	472.2	454.9	454.9	431.5	472.2	405.9	454.9
SP	359.6	359.5	327.8	327.9	468.3	468.3	456.9	456.9	359.5	468.3	327.9	456.9
LU	536.1	536.0	539.9	539.9	738.1	738.1	743.8	743.8	536.1	738.1	539.9	743.8
CG	189.3	189.2	188.2	188.1	208.7	208.7	206.8	206.8	189.3	208.7	188.2	206.8
MG	16.0	16.0	16.2	16.2	23.5	23.5	23.5	23.5	16.0	23.5	16.2	23.5

We compare the execution performance of the NAS benchmarks on single and dual-core processors. The effect of enabling small pages on both dual and single core execution of the kernels is shown in Table 10.

Table 10. Performance penalties for using dual core for both small and large pages.

	Decrease in Time from Enabling Small Pages		Time Increase for Dual Core	
	Single Core	Dual Core	Large Pages	Small Pages
BT	5.92%	3.66%	109.42%	112.05%
SP	8.82%	2.44%	130.24%	139.35%
LU	-0.71%	-0.77%	137.69%	137.77%
CG	0.59%	0.91%	110.29%	109.93%
MG	-0.84%	0.04%	146.35%	145.07%

Analysis of NAS Benchmarks

We see a range of effects from dual core execution ranging from 10% to 45%, with MG exhibiting the greatest effect and CG least (Figure 3). Of interest is that the enablement of small page sizes in execution has an apparently different effect on overall execution time of a given application depending upon whether it was executed in single or dual core mode. Applets exhibiting most sensitivity remain so in this regard. The trend appears to suggest that the effect of small pages is less pronounced when dual core mode is employed. The average degradation in execution time is approximately 27%, which is more significant than the typical performance drop on the NERSC-5 SSP applications. Notably, all NAS kernels exhibit greater overall flop rate in dual core mode.

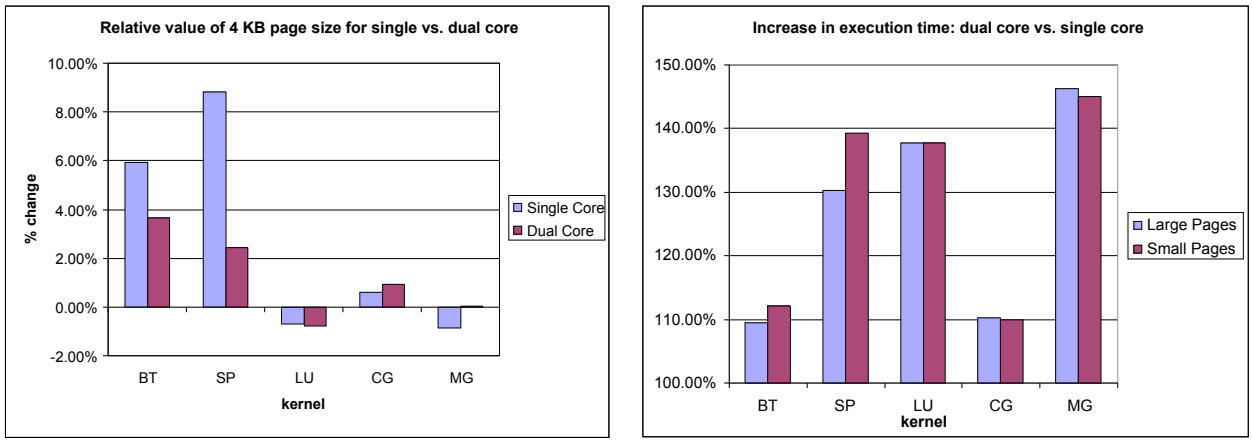


Figure 3. NAS benchmark performance. The figures show the change in NAS kernel performance as a result of the move from single to dual core for both small pages and large pages.

Microbenchmarks and Probes

While application benchmarks provide the most direct evidence of the effective performance that a given system architecture will deliver for a code, they also tend to be too complicated to infer the source of bottlenecks that result in performance differentiation. Microbenchmarks and probes are much simpler proxies that enable a more detailed study of specific architectural bottlenecks and behaviors that would otherwise be hopelessly intertwined when observed in a full-fledged application code. We will focus primarily on a handful of microbenchmarks that analyze the performance of the memory hierarchy with varying degrees of complexity and scope—STREAM, Membench, and Apex-MAP.

STREAM

The STREAM benchmark is used to assess the effective bandwidth delivered by the memory subsystem. Table 11 and Figure 4 show STREAM performance on the Cray XT3™ and Cray XT4™ systems. The stream results used an array size of 53,687,091 elements, offset of 0 and a total memory requirement of 1228 MB (well outside of the L2 cache). The primary source of contention is the memory interface, which is shared between the two cores on chip.

Table 11. Observed memory bandwidth (Mb per second per core) for Cray XT3™ and XT4™ from STREAM benchmark.

	One Core XT3™	One Core XT4™	Two Core XT3™	Two Core XT4™
Copy	5137	8196	2345	4074
Scale	5067	7257	2348	4012
Add	4734	7482	2309	3469
Triad	4135	7464	2310	3626

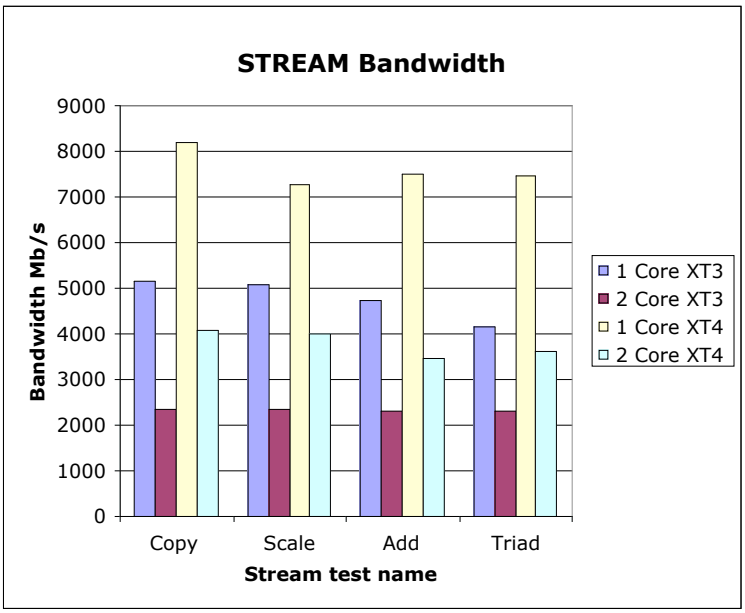


Figure 4. Graphical depiction of STREAM performance.

Membench

The Membench test performs a STREAM-like copy operation to examine memory bandwidth limited performance, but sweeps over a range of buffer sizes. The benchmark was run on both the XT3™ and XT4™ systems. The XT3™ uses the same performance processor cores as the XT4™, but contains slower DDR1 memory, whereas the XT4™ uses DDR2 memory interfaces, which offer a much higher bandwidth.

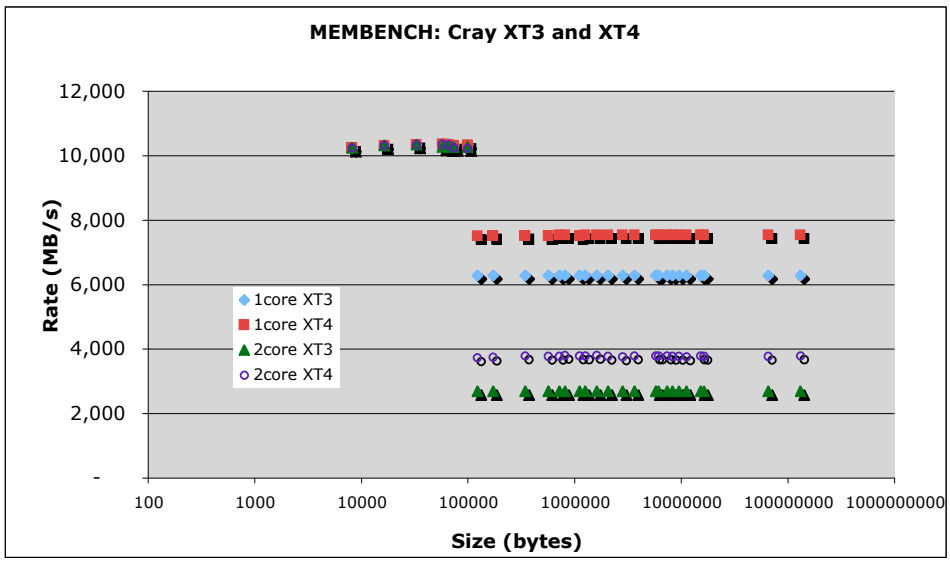


Figure 5. Membench results for XT3™ and XT4™.

We can see clearly from Figure 5 that when the benchmark fits within the L2 cache, the Opteron™ processors (both Rev-E on XT3™ and Rev-F on XT4™) maintain constant

performance at 10+ GB/s due to the independent L2 caches. However, once the processors must go to main memory, the effective memory bandwidth per core is cut in half. For the XT3™, Membench goes from 6 GB/s sustained per core to 2.8 GB/s sustained per core. For the XT4™, the performance drops from 7.5 GB/s for single core down to 3.9 GB/s per core when running on both cores.

This makes it clear that the primary source of bandwidth contention for the AMD processors is bandwidth on the memory interface. Otherwise, the independent L2 caches ensure that both cores can operate without interfering with one another for independent cache-resident data.

Apex-MAP

Apex-MAP (Memory Access Probe) is a synthetic benchmark that was designed to trace out a richer variety of application memory access patterns than benchmarks such as STREAM, Membench, or Cachebench are capable of producing. Whereas STREAM and Membench only deal with unit-stride memory access patterns, Apex-MAP (as shown in Figure 6) provides access patterns that vary in both temporal locality (the α parameter) and spatial locality (the L parameter). The α parameter is expressed as an exponent to a power distribution for the memory addresses that varies from 0 to 1, where 0 represents a high degree of spatial locality in the memory accesses and 1 represents a uniform random access that spans memory space. The L parameter simply refers to the length of the contiguous data access, which varies from 1 to 64 K words of memory. These two parameters can be varied continuously to trace out a 2D map of memory subsystem performance for a broad range of memory access patterns. More details can be found in the SC2005 paper describing Apex-MAP's capabilities (http://crd.lbl.gov/DOEresources/SC05/ApexMap_EStrohmaier.pdf).

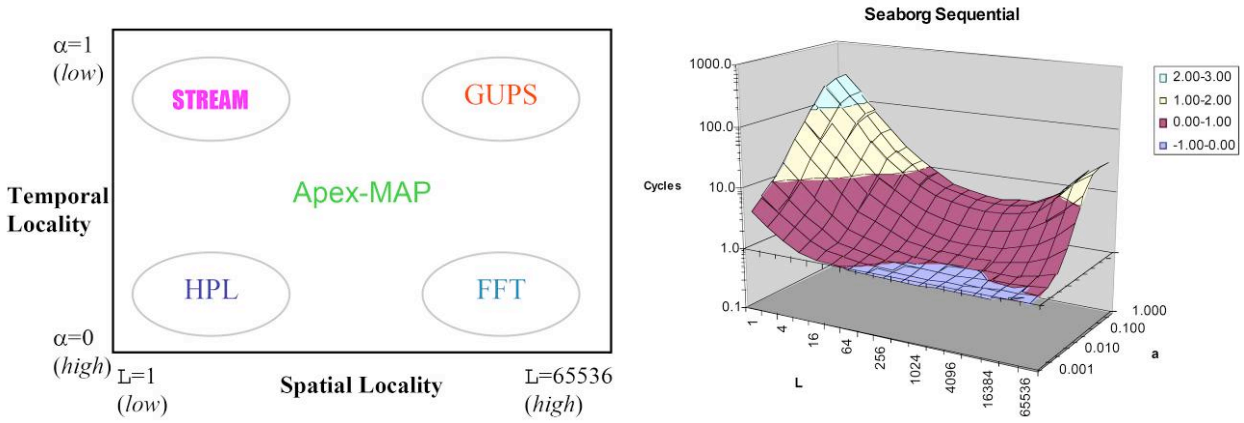


Figure 6. Apex-MAP traces out a 2D space of memory access patterns characterized by their spatial and temporal locality. The spatial locality, L, describes the size of contiguous accesses to a given memory location. The temporal locality, α, is an exponent for a power law distribution of memory accesses. The diagram on the left shows how the memory access patterns of various benchmarks correspond to the continuous 2D space traced out by Apex-MAP. The diagram on the right shows a typical Apex-MAP plot of the performance of a Power3 microprocessor on the NERSC-3/Seaborg system. The performance (vertical direction) is given in terms of cycles per memory access.

The serial version of Apex-MAP was run on the ORNL Jaguar system, which has dual-core 2.6 GHz Rev-E Opteron™ processors. A single copy of Apex-MAP was run on each node for the

unpacked (single-core) case, and two copies of serial Apex-MAP were run for the packed (dual-core) case.

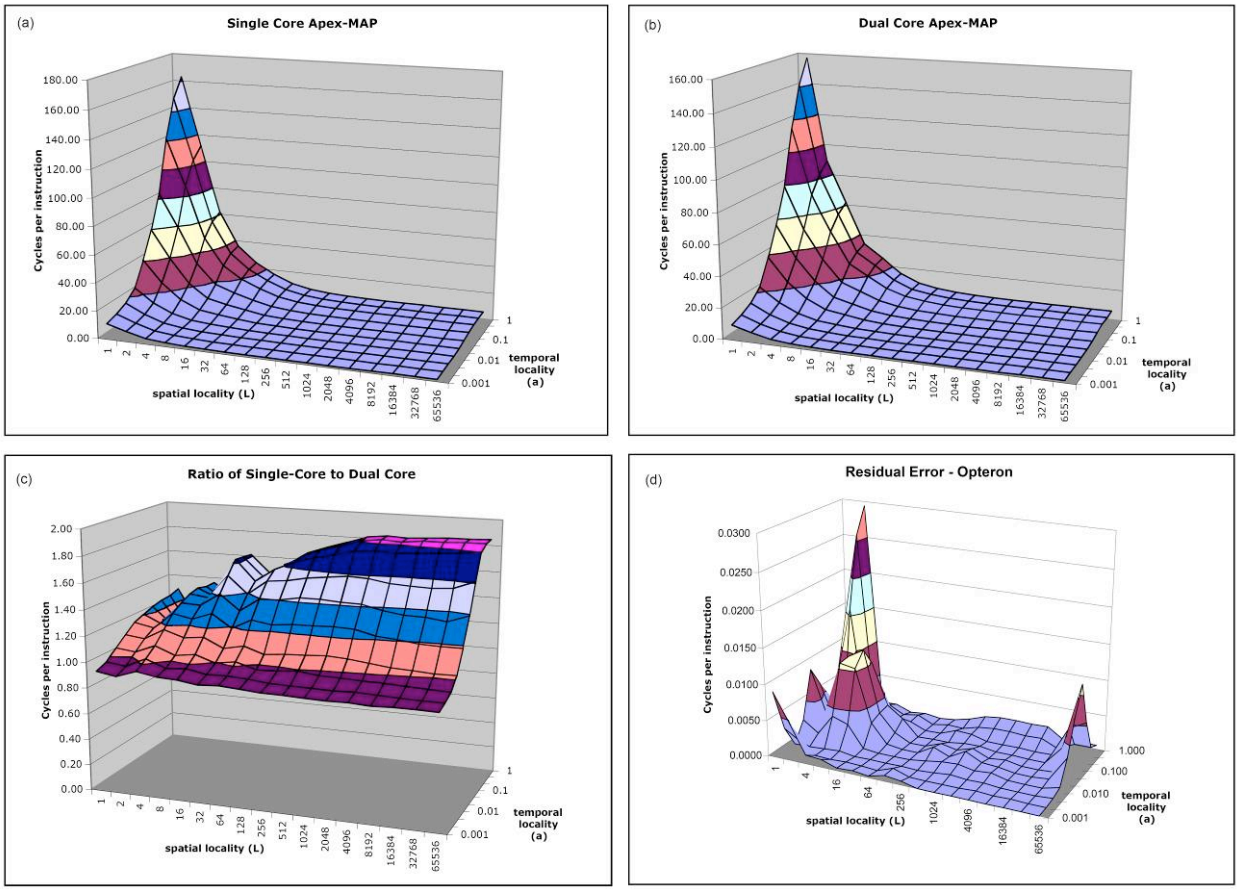


Figure 7. Apex-MAP results for AMD Opteron™ processors on XT3™.

Figures 7a and 7b show the Opteron™ processors' response to the different memory access patterns presented by Apex-MAP. Figure 7c shows the difference between the single-core and dual-core performance. It points out quite clearly that the dual-core processor performance is cut nearly exactly in half when the two cores have an access pattern that is primarily bandwidth bound. However, when the memory access pattern causes performance to be bounded by latency, there is very little performance penalty for running with both cores. So the degree to which code performance suffers from moving from single core to multi-core is dependent on how much of the code is in fact bandwidth bound. As we will see from the application results, it is not clear that many codes are indeed in the bandwidth bound regime.

Erich Strohmaier created a simple linear model for the response of the memory subsystem based only on the bandwidth and latencies to memory and to the L2 cache. In Figure 7d, we see the residual error between the performance predicted by this simple model and the actual Apex-MAP performance. We can see across the entire parameter space, the very simple model correctly predicts the memory response except in the case with the lowest spatial and temporal locality. The error is likely due to the fact that the model does not account for branch mispredicts, TLB misses, and much lower memory efficiency for the random accesses across the

entire memory space. What this means is that the predicted dual-core performance across a wide range of memory access patterns is correctly accounted for using the simplest possible model for memory performance—namely, that the effective latency of the memory subsystem and bandwidth + latency for the L2 cache remains the same regardless of how many cores are used, but the main memory bandwidth is cut in half. This result assures us that there are unlikely to be additional sources of contention in the processor design aside from the memory bandwidth contention.

Extrapolating to Quad-Core Performance

We observe from the AMD architectural discussion that when excluding messaging performance, the primary source of contention when moving from single core to dual core is memory bandwidth. The testing with STREAM and Membench microbenchmarks confirms this assumption, as the performance of the 2.6 GHz AMD cores only becomes differentiated when the data sizes become larger than the L2 cache and must go to main memory. Finally, the analysis with Apex-MAP shows that a simple performance model that presumes only that memory bandwidth is cut in half when processors access main memory is highly accurate in predicting dual-core performance for a wide variety of memory access patterns. Therefore, investigation of more complex models for dual-core performance is unlikely to yield higher-fidelity results.

With these conclusions in mind, we go back to the task of extrapolating the performance of the NERSC benchmarks on quad-core systems. We begin by enumerating the assumptions of our model:

1. The only source of performance difference between single- and dual-core runs is memory bandwidth contention.
2. The 2.6 GHz RevE and RevF AMD cores execute code at roughly the same performance in the absence of memory bandwidth contention.
3. We can therefore break execution time into the portion that is stalled on shared resources (memory bandwidth) and the portion that is stalled on non-shared resources (everything else).
4. Under this circumstance, we can use the timing difference from single- to dual-core runs to compute the fraction of execution time spent in memory bandwidth contention.
5. We can then extrapolate the quad-core performance by assuming the time spent in the execution component remains the same, but the time spent in memory bandwidth contention will increase proportional to the reduction in effective memory bandwidth per core.

We began by testing the above assumption using the XT3™ performance data to predict the effective performance on the XT4™. The XT4™ in this test operates at the same clock frequency as the XT3™, but the DDR2-667 memory subsystem is 30% faster than the DDR1-400 MHz memory of the XT4™. Starting with the MILC data:

- The execution time for single-core runs on the XT3™ is 160 seconds, and the time spent in dual-core is 230 seconds.
- The STREAM benchmarks indicate that the memory bandwidth for dual core is approximately half that of the single core, so if the five assumptions above hold true, we should expect execution time to obey the relationship:

- single core: core_exec_time + bandwidth_contention_time = 160 s
- dual core: core_exec_time + 2*bandwidth_contention_time = 230 s

- Solving the above system of equations provides us with an estimate of 90 seconds spent executing in the core (for both single and dual core) and 70 seconds spent in memory bandwidth contention for single core and 140 seconds (2x longer) spent in memory bandwidth contention for dual core.
- We use the STREAM bandwidth numbers to project the time spent in memory bandwidth contention for the XT4's™ faster memory subsystem and compare to the actual time spent in Figure 8 below.

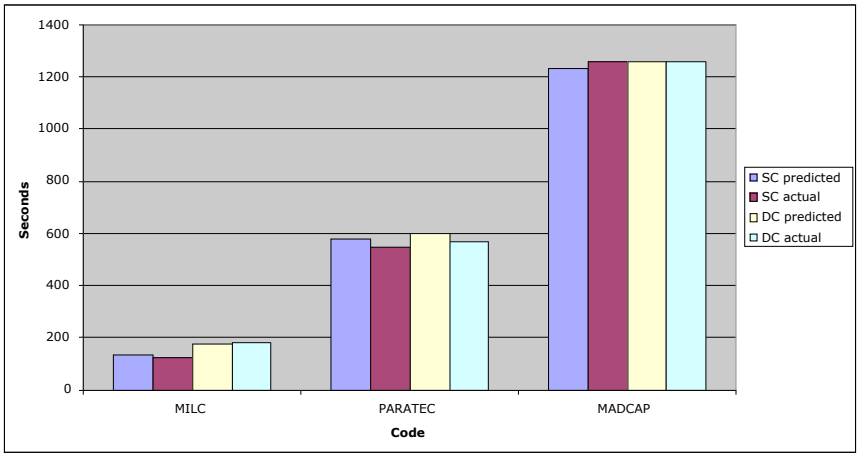


Figure 8. Predicted and actual execution times on XT4™ using our performance estimation methodology.

Figure 9 shows that the prediction error for MILC, PARATEC, and MADBench is very low (<5%). The prediction for CAM is far off, but we also observe that the compiler version changed between the XT3™ and XT4™ CAM runs, so it probably should not be used in the prediction.

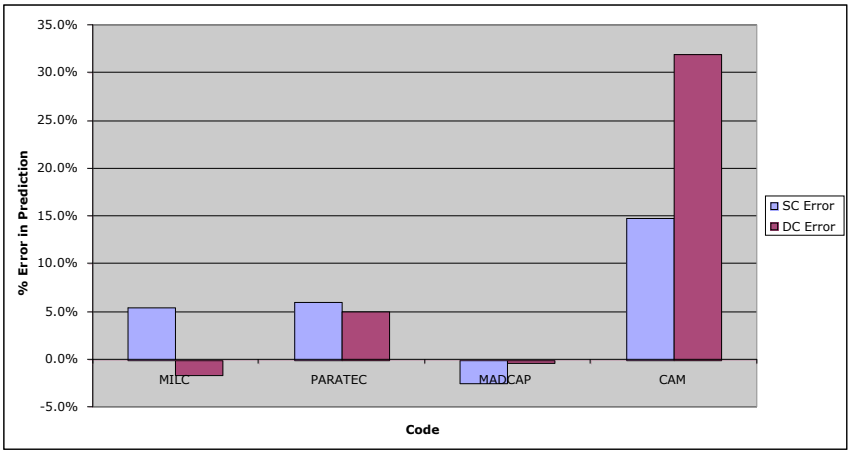


Figure 9. XT4™ Prediction error for our methodology. Most of the predictions are good within 6% of the actual runtimes.

Next, we use this prediction methodology to estimate the impact of moving to quad core without any improvements in memory bandwidth. We assume for the moment that the clock frequency

for the quad-core processors remains the same at 2.6 GHz, ignore changes to the core microarchitecture (particularly the new SIMD units), and assume that the L3 cache can be used effectively to mitigate the smaller L2 cache.

Figure 10 shows the net effect on the execution time for the SSP benchmarks using this simple method for prediction. The average performance impact of moving from single core to dual core without improving memory bandwidth is 7.4%, but the impact of moving from single core to quad core without some improvement in memory bandwidth increases to 21.4% on average. This model, of course, ignores other microarchitectural improvements to the AMD Opteron™ architecture and ignores interconnect bandwidth contention, but does indicate clearly that some improvement in memory bandwidth will be required to maintain computational efficiency for the NERSC-5 SSP.

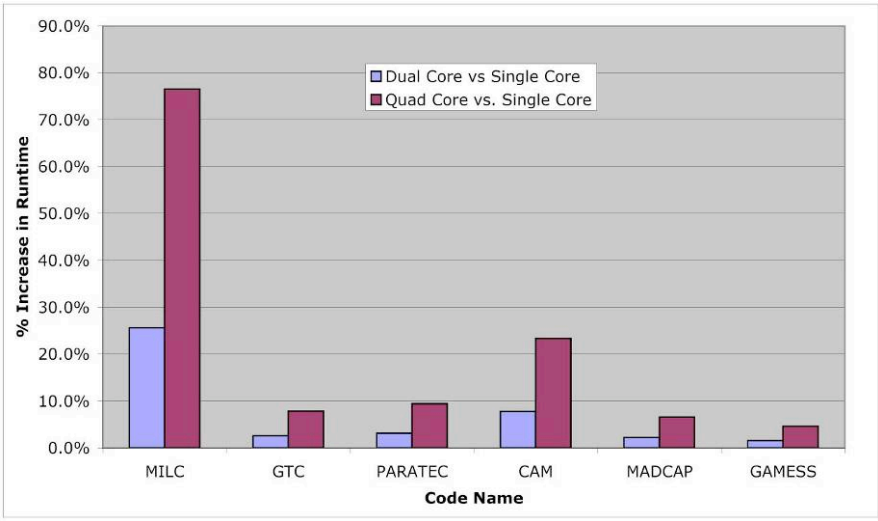


Figure 10. This graph shows the net increase in runtime for the NERSC-5 SSP benchmarks under the assumption that memory bandwidth contention can be treated exclusively from the other components of the runtime.

Optimizing Code for the AMD Multicore Processors

This section describes compiler options and code restructuring techniques that can be used to improve code performance on multicore processors. Some of these techniques simply represent "due diligence" for optimizing codes for the AMD Opteron™ processor architecture, regardless of the number of cores involved.

Compilers, Libraries, and Profiling

The Portland Group (PGI) compiler suite, version 6.1, was used for the Cray XT3™ results cited in this document. For all benchmarks except CAM, the `-fastsse` compiler optimization option was selected. Due to issues with result accuracy, `-fast` was instead used when compiling CAM. Where necessary, version 3.0 of the AMD Core Math Library (ACML)™ and Cray LibSci™ provided basic mathematical kernels.

The `-fast` compiler option is designed as a safe but good optimization level. This option begins at `-O2` and adds `-Munroll=c:1, -Mnoframe,` and `-Mlre.` The `-Munroll=c:1` option tells the

compiler to completely unroll all loops with a loop count of 1 or less, effectively eliminating loop overhead when a loop is unnecessary. Even if a loop does not meet the criteria to be completely unrolled, the compiler will consider partially unrolling the loop to reduce loop overhead. The `-Mnoframe` option prevents the generation of code to set up the stack frame, which is generally only needed for debugging purposes. Lastly the `-Mlre` option eliminates loop-carried redundancies, which can reduce memory references and arithmetic operations.

The `-fastsse` option starts with all of the same optimization options as `-fast`, but further extends the optimizations to include vectorization through the use of SSE. The option expands to `-fast -Mvect=sse -Mscalarsse -Mcache_align -Mflushz`. The `-Mflushz` option simply tells the processor to flush SSE registers to zero. Since this is a processor-level option, it is a no-cost optimization. `-Mvect=sse` instructs the compiler to do two things. First, it enables automatic vector pipelining so that the compiler will attempt to vectorize loops. Doing so increases the number of operations performed per memory operation, in turn increasing throughput. Second, the option instructs the compiler to generate SSE instructions to handle the vectorized loops. SSE (streaming SIMD extensions) instructions support fetching a single instruction to execute on multiple data items, which improves processor throughput by reducing instruction fetches and by hiding memory latency. The `-Mscalarsse` option instructs the compiler to use SSE instructions for scalar math operations, which often results in faster execution. Lastly the `-Mcache_align` option forces unconstrained data objects of at least 16 bytes in size to be aligned on cache line boundaries. A data object is considered unconstrained if it is not in a FORTRAN common block or a member of an aggregate data structure. In order to assure that stack-based local variables are cache aligned, this option must be used.

For one application, Apex-Map, an attempt was made to improve performance by manually unrolling the main program loop. Instead we found that this manual unroll actually caused slightly worse performance. We were not able to conclusively determine why this occurred. It is likely that the manual unrolling affected the compiler's ability to optimize the loop, causing poorer performance.

Restructuring to Achieve Better Cache Performance

Utilizing cache effectively is quite simple in concept; however, the implementation can be difficult. The basic idea is to structure your program to work on a set of data that can reside in cache while numerous updates are performed on that data. In most codes this can be accomplished by blocking the logic of the program to work on subsets of the data.

On massively parallel processors, we frequently see a greater than expected increase in performance when the number of processors increases. When the problem size is fixed, a larger increase in performance can occur when the dataset on each processor is small enough to fit in cache. This phenomenon is called super-linear speedup and is attributed to increased cache reuse.

If this is the case, why can't one simply block the applications using an inner/outer loop structure whereby each invocation of the outer loop accesses a dataset that is small enough to fit in cache? Conceptually this is an excellent approach and will be covered later in this section.
A complication to this approach is that caches have associativity that can hinder attempts to completely use the cache. The application programmer can avoid this complication by paying

strict attention to the alignment of arrays in memory. This issue will also be addressed later in this section. Another issue related to alignment of arrays has to do with the utilization of SSE2 and SSE3 instructions. To minimize the overhead to utilize these vector instructions, arrays should be aligned on 128-bit boundaries.

Lastly an application programmer can increase the computational intensity of a DO loop to perform more floating-point operations for each fetch/store. This technique needs to be coordinated with the compiler, since it tries to do the same thing when higher levels of optimization are employed.

Memory Layout and Alignment

The user has significant control over the allocation of arrays in memory, but most users do not understand how important that allocation is. On the Opteron™ architecture, level 1 cache is the most important to use effectively, and it is the most difficult cache to use. First, it is only 64 KB, and second, it is two-way associative. A cache line on this system is 64 bytes, and any single cache line can only go into two slots in level 1 cache. This is the problem.

If the user views memory arranged in 512 columns, each the width of a cache line, as in Table 12, we can see that the cache lines in the first column of memory can only go into C11 or C21 (Cache Associativity Class 1 – 1 and Cache Associativity Class 2 – 1). While there are a good number of cache lines in the entire cache, if we require three lines from the first column of memory, then there will be thrashing of the level 1 cache. Effectively, data will be flushed from level 1 cache to level 2 cache, and when we need it again, there will be a delay until it is retrieved from level 2 cache.

Table 12. Memory Layout and Alignment

Level 1 Cache

C11	C12	C13	C14	…	…	…	C1510	C1511	C1512
C21	C22	C22	C24	…	…	…	C2510	C2511	C2512

Memory

1	2	3	4	…	…	…	510	511	512
513	514	515	516	…	…	…	1022	1023	1024
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
64257	64258	64259		…	…	…	64766	64767	64768

Now let us investigate a poor allocation of data that will significantly impact performance on this cache architecture. The following Fortran code should be illustrate the point:

```
REAL * 8 A(64,64),B(64,64),C(64,64)
DO I = 1,64
  C(I,1) = A(I,1) + B(I,1)
ENDDO
```

Regardless of which column of memory A(1,1) is being allocated, since the size of the A array is exactly the size of one associativity class, the first element of B will reside in the same column of

memory, and consequently the first element of C will reside in the same column of memory. When the DO loop executes, A(1,1) through A(8,1) will be fetched into C11; next B(1,1) through B(8,1) will be fetched into C21. Once A(1,1) is added to B(1,1), the application must fetch up C(1,1) through C(8,1) to replace C(1,1). This fetch of C(1,1) must go into C11 or C21. It will result in one of these cache lines being evicted to level 2 cache. The next pass through the DO loop will require A(2,1) and B(2,1), fetching previously evicted cache lines from level 2 cache and resulting in C(1,1) through C(8,1) being evicted to level 2 cache. This DO loop will result in significant thrashing of cache, since only two of the three cache lines can reside in level 1 cache at the same time.

In this situation, nothing would prohibit the compiler from simply padding between the arrays to avoid this situation. Adding a temporary array the size of one cache line between the arrays would turn this horrible cache-thrashing example into a reasonably cache-friendly example.

While the compiler can do this padding in the above example, if the arrays were in a COMMON block, or if the arrays were passed into a subroutine as arguments, the compiler could not do the required padding. A compiler has significant restrictions imposed to assure storage and sequence association of the Fortran program.

Now this is a simple example. How about a large application? This example is easily fixed by changing the arrays slightly. What if there are too many arrays to examine all of their starting memory locations and determine if you are indeed being victimized by cache associativity?

Looking closer, we see that level 1 cache will hold 1024 cache lines of 8 –8 byte words, that is, 8192 words. If we allocate our arrays a power of 2 plus a cache line, we have a good chance that we will not have significant cache thrashing. In the upcoming section on cache blocking, we must understand that all data that is used—Loop indices, addresses, etc.—must be stored in the cache. When analyzing the DO loops data usage, it is usually best to leave some cache lines available for some of these scalar quantities.

The proposed alignment discussed here is also conducive to using SSE instructions. On today's hardware, SSE instructions can deliver twice the performance in 32-bit mode, and the data move instructions can improve 64-bit performance. These instructions can only be issued on data that is aligned on 128-bit boundaries. If all of our arrays are of length equal to a power of 2 plus the width of a cache line (for example, 1024 + 8 = 1032) then we have the required alignment. The compiler will try to do some shuffling to align data to use these instructions; however, if the user performs the alignment, fewer overheads will be incurred. Also the user must compile the application with `-fastsse` to use these instructions.

As was mentioned in the previous section, the quad-core systems will have the ability to perform four floating-point ops/clock. This means that the SSE instruction on 64-bit instructions will also result in double the performance of non-SSE work.

Unrolling to Increase Computational Intensity

Some cases of Fortran DO loops can benefit significantly by unrolling to increase computational intensity. *Computational intensity* is defined as the number of floating-point operations divided

by the number of array fetches/stores. Unrolling to increase computational intensity effectively reduces the number of fetches and stores, which is always a good thing to do. Consider the following DO loop:

```
   DO 46011 J = 1, 4
     DO 46010 I = 1, N
      C(J,I)=0.0
46010 CONTINUE

      DO 46011 K = 1,4
      DO 46011 I = 1,N
       C(J,I) = C(J,I) + A(J,K) * B(K,I)
46011 CONTINUE
```

The computational intensity of the innermost DO loop 46011 is 2/3—we do not count the fetch of A(J,K) since that is a scalar with respect to the DO loop 46011. How can we improve this piece of code? We can always unroll short DO loops inside the longer DO loops as follows:

```
!    THE RESTRUCTURED
   DO 46012 I = 1, N
    C(1,I) = A(1,1) * B(1,I) + A(1,2) * B(2,I) &
      + A(1,3) * B(3,I) + A(1,4) * B(4,I)
    C(2,I) = A(2,1) * B(1,I) + A(2,2) * B(2,I) &
       + A(2,3) * B(3,I) + A(2,4) * B(4,I)
    C(3,I) = A(3,1) * B(1,I) + A(3,2) * B(2,I) &
       + A(3,3) * B(3,I) + A(3,4) * B(4,I)
    C(4,I) = A(4,1) * B(1,I) + A(4,2) * B(2,I) &
       + A(4,3) * B(3,I) + A(4,4) * B(4,I)
46012 CONTINUE
```

Now the computational intensity increases from 2/3 to 28/12, and the performance for N = 250 increases from 500 Mflops to 2800 Mflops. If we use `-fastsse` on the original, some amount of unrolling is achieved, and without the restructure, the compiler gets 1800 Mflops. We count both a fetch and a store for C(1,I), C(2,I), C(3,I), and C(4,I) since this array will have data elements replaced into it.

This loop unrolling is employed extensively in the ACML™ libraries, and often the user should call a BLAS (using the highest level BLAS possible)or LaPACK routine. Whenever the code has matrices of rectangular shapes with a short dimension much shorter than the long dimension, the Fortran code, if correctly written, will usually beat the call to the library.

Blocking for Cache Reuse

Blocking for cache reuse is the most powerful restructuring technique for achieving the best performance on today's chip technologies. Blocking refers to taking a multi-nested DO loop and updating the multi-dimensional arrays with chunks that will fit within the cache. Blocking gets the best results when the data has already been allocated as discussed in the previous section. The best example of blocking is a simple matrix multiply kernel:

```
DO 46031  K = 1, N
    DO 46031 J = 1, N
```

```
    DO 46031 I = 1, N
     A(I,J) = A(I,J) + B(I,K) * C(K,J)
46031 CONTINUE
```

Quick analysis tells us that this matrix multiply will fit into level 1 cache as long as the arrays do not conflict in cache and N is less than 52 for 8 byte reals and 75 for 4 byte reals. Assuming that N is 1000, the innermost `DO` loop will not fit into cache, so we will introduce some outer `DO` loops to chunk the data so that it will fit. Assuming we are using `real*8`, we want the inner chunk to be a square chunk of 50 × 50. To achieve that we can:

```
DO JJ = 1,20
DO II = 1,20
DO 46031  K = 1, N
    DO 46031 J = 1+(JJ-1)*50, JJ*50
    DO 46031 I = 1+(II-1)*50, II*50
     A(I,J) = A(I,J) + B(I,K) * C(K,J)
46031 CONTINUE
ENDDO
ENDDO
```

Notice that the innermost `DO` loop only updates chunks of 50 × 50 at a time, and this should fit into cache as long as the arrays are aligned appropriately.

Not everyone has matrix multiply kernels; however, this approach can be used when any 2- or 3-D grid is used. If one considers the situation where the outermost loop of a 3-D grid is located at a high level and the computational planes are updated within the subroutines, most times the planes of arrays are larger than the cache can hold. Often times these planes can be broken into chunks of pencils (several one-dimensional chunks that fit in cache). It is important that all the computation be done on the pencil chunk—requiring the introduction of a `DO` loop within the outer grid loop that goes over the second dimension, as indicated above in the MATMUL example. This is not an easy restructuring task, and is best accomplished when originally writing the application.

As we see more and more cores and a higher mismatch between memory bandwidth and flop rates, these restructuring techniques will become even more important.

NAS Kernel Optimization

In this section, we demonstrate how to apply the above optimizations using the NAS Class-B MG kernel as an example. We point out that the MG kernel saw the largest performance drop when moving from single core to dual cores. Execution time increases by 1.4x, which corresponds to a 40% drop in efficiency when moving from single to dual cores. We show below how a simple cache-blocking optimization can yield substantial performance benefits that are actually amplified when moving from single to dual core.
Using the advice from the previous section, we applied a simple blocking of the loops in `psinv()` and `resid()`, thus

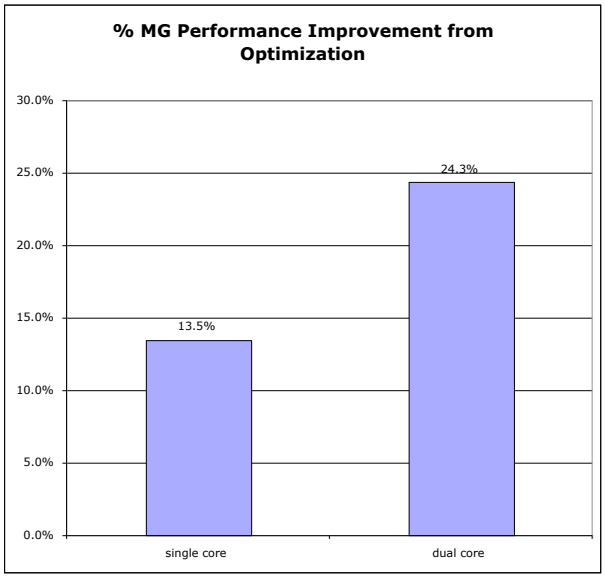
```
do i3=2,n3-1
 do i2=2,n2-1
```

```
      do i1=1,n1
```

is replaced by

```
do i3block=2,n3-1,BLOCK3
 do i2block=2,n2-1,BLOCK2
  do i3=i3block,min(n3-1,i3block+BLOCK3-1)
   do i2=i2block,min(n2-1,i2block+BLOCK2-1)
    do i1=1,n1
```

We did a search to find an appropriate parameterization of `BLOCK2` and `BLOCK3`. As shown in Figure 11, this optimization is effective on single-core execution, providing 14% performance improvement; however, on dual-core runs, the effect is more pronounced, giving 25% improvement in the packed run with the optimal page size.



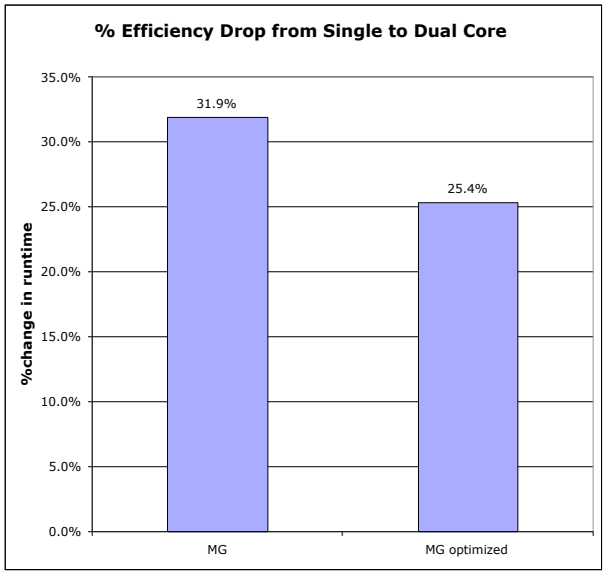


Figure 11. The graph on the left shows the percent improvement on NAS MG performance derived from the cache-blocking optimizations. The graph on the right shows the optimization also reduces the impact of moving from single core to dual core.

Advice to Code Developers

The following suggestions should only be applied to those routines that use most of the CPU time in your program. You cannot always design a data layout that is good for all parts of the program, so concentrate on the major computational routines.

Data Layout

Going forward into an era of multi-core sockets, code developers must understand that the only way they will achieve a high percentage of peak performance is by organizing their data to be cache friendly. As we have seen in this report, memory bandwidth/core is decreasing and unfortunately in four years the dual-core Opteron™ architecture will look well balanced

compared to what is coming. On the plus side, we are getting more levels of cache and all in all more cache memory. Some general rules for laying out your data are:

1. Make sure your innermost loop is contiguous and vectorizable.
2. Make sure that the sizes of the dimensions of the arrays are not a large power of 2; it is best to have them a power of 2 plus a cache line. Make sure that the length of the array dimensions are a multiple of 128 bits.
3. If possible, variables that are used together should be grouped together in memory.

Shared Memory Parallelization

Start thinking about shared memory parallelization now; in the future, this programming paradigm on the socket will allow you to more effectively use bandwidth off the node. Additionally, if you consider using OpenMP and/or Pthreads on the socket, you should see better utilization of the shared level 3 caches. The potential for four MPI tasks on the quad core sharing usage of the level 3 cache is zero; however, if OpenMP and/or Pthreads are used across the quad cores, you may be able to share data through the level 3 cache. Once again, cores/socket will get larger, and a hybrid programming paradigm may perform well on these new sockets.

Vectorization

With the quad-core system there will be an even larger benefit from using the SSE instructions and it will get even more beneficial in the future. Vectorizing for SSE requires that the arrays are being accessed contiguously and the variables are on 128-bit boundaries. The loops do not have to be too long. Loops of length 8 are just fine.

Prefetching

Many loops can benefit from the use of prefetch directives. These are very compiler dependent. If you really understand how the cache works and the amount of data being stored in the cache, then intelligent use of prefetch directives can be beneficial.

Advice to System Designers

The Memory Wall

We observe that the primary source of contention on the AMD Opteron™ dual-core processor is memory bandwidth, due to the shared memory interface. The relative simplicity of this bottleneck results in a correspondingly simple remedy. AMD would do well to maintain this level of simplicity for future generation processor designs by limiting the degree to which on-chip resources are shared or ensuring that such resources are shared in a fashion that provides predictable, easy to understand, and (in particular) *measurable* responses. It is essential that AMD provide performance counters that directly measure contention on *any* contended resources! Currently, even for very simple sources of resource contention (e.g., memory bandwidth contention), AMD does not provide adequate performance counter resources to measure this directly. One must take multiple timings to acquire enough different counters to infer the quantity of memory bandwidth contention. AMD must address this in future generation processors as the *additional* sources of contention arise.

There has been considerable concern over the growing imbalance between memory system performance and processor performance (the "memory wall"). Many fear that multicore has further exacerbated this imbalance, placing us on the wrong side of the memory wall. In fact, multicore does not represent a dramatic shift from the historical trajectory regarding memory wall. Had clock rates continued to increase at historical rates (doubling clock frequency every 18 months rather than number of cores), the same issues regarding bandwidth starvation would exist. The move to multicore continues to exacerbate memory bandwidth problems, but the stall in clock frequency has diminished the corresponding worsening of memory latency in terms of processor-clocks.

Evidence collected in this report shows that many codes are clearly still bounded by other performance bottlenecks. We find that the majority of codes in our study were clearly not limited by memory bandwidth contention—those with low computational intensity are likely suffering more from memory latencies. While AMD's on-chip memory controllers have improved efficiency for such latency-bound codes, additional attention to advanced (possibly user or compiler directed) prefetch capabilities could go a long way towards reducing latency-bound performance bottlenecks.

Opteron™ Processor TLB

The TLB on the Opteron™ processor turns out to be a more serious cause of performance differentiation than the move from single to dual cores. With 512 TLB entries, the 4 KB small pages do not provide adequate coverage of physical memory for demanding HPC codes. However, the TLB can only hold 16 entries for the large 2 MB pages! So although the TLB coverage is increased from 2 MB to 16 MB of physical memory, the small number of entries increases the amount of TLB thrashing for codes that must access multiple arrays that are laid out in distinct parts of memory —creating a catch-22 for most application codes.

The move to 128 2-MB page translations will enable coverage of 256 MB of memory. The availability of 1 GB pages is not going to be helpful in this regard because the OS must pin at least one physical page for the kernel memory—making an entire gigabyte of physical memory on the node unavailable to applications. Therefore, the 1 GB page size does not offer a credible solution unless it can be intermixed with smaller page sizes. AMD should, therefore, target larger coverage for the 2 MB page size in future generation processors by offering more TLB entries (for example, offering 512 entries for both 4 KB and 2 MB pages) for this intermediate page size.

Quad Core Cache Changes

In order to squeeze more cores onto a single die, the Opteron™ quad-core processor will offer much smaller L2 caches but will add an L3 cache. This leads to two concerns.

The most immediate impact of the change in L2 cache size will be on PDE solvers using block-structured grids (stencil codes). The MSP2005 paper by Kamil et al.[2] provides a performance model that indicates that the on-chip cache must be large enough to store at least three planes of

2 S. Kamil, L. Oliker, J. Shalf, "Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations," Proceedings of the ACM Memory Systems Performance 2005 Conference (MSP05), Chicago IL, 2005. (http://crd.lbl.gov/~oliker/papers/msp_2005.pdf)

the block-structured grids required by the calculation to achieve peak performance. The smaller caches will require a correspondingly smaller working set size per core. Overall, the grids supported efficiently by the quad-core processor will be roughly the same size as those supported by the dual-core processors it replaces.

The shared L3 cache will likely provide some benefits for shared-memory codes where cache-lines would otherwise thrash between the L1/L2 caches of different cores that touch the same line. However, codes that are exclusively MPI-based (without shared cache lines) may see higher conflict misses in L3 for poorly aligned memory references.

Dual Memory Controllers

As the number of cores on chip increases, the stream of memory addresses presented to the memory controllers is likely to be less regular. When running MPI codes on the current dual-channel memory subsystem, the address stream will reflect two different pages in memory, resulting in thrashing of row addresses presented to memory and reducing performance due to the penalty of having to constantly reopen memory banks. The quad-core processor will introduce independent memory channels that will improve the apparent coherence of the address stream. This will benefit codes that run in MPI mode (running in separate address spaces). But as the Cray XT™ nodes move to support hybrid and shared-memory programming models, it may be useful to offer the option to use the memory channels in either fashion depending on the application. It would be useful if the controller would support both modes of organizing the memory channels (independent and interleaved), depending on which kind of code will be run on the nodes.

Cache Coherency Protocol: *Necessary but Not Sufficient*

While concerns revolving around memory bandwidth for multicore are overblown, concerns regarding the ability to write code that exposes enough parallelism to take advantage of the multicore chip are essential. While this has resulted in new activity in the HPC languages community, the hardware vendors can play an essential role in enabling simpler access to on-chip concurrency.

Initially, applications are likely to treat multicore and manycore (8+ cores) chips simply as conventional symmetric multiprocessors (SMPs). However, looking forward, multicore processors offer unique capabilities that are fundamentally different from SMPs, and which present significant new opportunities:

- The intercore bandwidth on a multicore chip can be many times greater than is typical for an SMP, to the point where it should cease to be a performance bottleneck.
- Intercore latencies are far less than are typical for an SMP system (by at least an order of magnitude).
- Multicore chips could offer new lightweight coherency and synchronization primitives that only operate between cores on the same chip. The semantics of these fences are very different from what we are used to on SMPs, and will operate with much lower latency.

If we simply treat multicore chips as traditional SMPs—or worse yet, by porting MPI applications—then we may miss very interesting opportunities for new architectures and algorithm designs that can exploit these new features. Therefore, AMD and Cray should be

deeply involved in collaborating with the programming model development community to find ways to exploit these unique capabilities.

The computer architecture and languages community has made a number of recent advances that take better advantage of the available on-chip bandwidth and that look well beyond the traditional SMP model for multithreading.

For example, developers currently depend on cache-coherency protocol to implement mutual exclusion locks that are typically used for intercore coordination. However, using MOESI or MESI protocol for such locks creates huge amounts of redundant or unnecessary traffic across the HT fabric that could be eliminated if the scope of the transactions were limited to on-chip or if additional features were added to the cache-coherency protocol to directly support such features. Support for producer-consumer data transfer models between cores would enable more bandwidth-conservative stream programming models. More flexible data coherency schemes can better leverage the on-chip bandwidth and reduced latency without being having efficiency compromised by the need to enforce strict global ordering of memory transactions. In addition, real-time embedded systems such as the STI Cell processor and the NVidia 128-core processor offer more direct control over the memory hierarchy, and so could benefit from on-chip storage configured as software-managed scratchpad memory.

Transactional memory can make multithreading and auto-parallelization much simpler and more robust. Normally the compiler or a programmer (using OpenMP) must make assumptions about memory correctness and the scope of data arrays when translating serial code into loop-parallel code. The programmer must allocate and use explicit lock variables to ensure mutual exclusion where memory hazards exist. If any assumptions are incorrect, the code will fail in inconsistent or unexpected ways. Transactional memory simplifies mutual exclusion because programmers do not need to allocate and use explicit lock variables or worry about deadlock. Each loop iteration in a parallel loop can be couched as a transaction. If any of the parallel loop iterations have a data dependence, then the transactions will automatically detect and resolve the conflict by rolling back and re-trying the conflicting transaction. The advantage over locks is that if programmer (or compiler) assumptions about the safety of parallelizing a loop prove to be incorrect, the program will still produce the correct answer—it will just do so without a parallel speedup. Such mechanisms will not necessarily improve multicore performance, but they can definitely make multithreading and hybrid programming more accessible, simpler to debug, and less prone to errors.

It is much easier to implement transactions efficiently using on-chip resources than it is to implement them between cores on an SMP. Hardware-supported transactional memory can coexist with cache logic using relatively minor changes.
The ultimate message is that cache-coherence protocol is a necessary minimum level of service for coordinating the actions of cores on a multicore processor, but it is not sufficient! Multicore chip vendors and system integrators such as Cray should engage the software development community in investigating these various alternatives for making parallel programming and multithreading more tractable for the software development community at large.

Conclusions

The applications tested generally made efficient use of a second core, with an average 10.3% slowdown compared with single-core execution, and with a factor of 2 decrease in real cost (power, cooling, platform support infrastructure, etc.).

In contrast, the NAS kernels examined provided a less favorable result, exhibiting 27% degradation on average. Though this is still a positive overall result, it shows that there are certain cases in which such a good result may not be exhibited. Notably, no application exhibited an aggregate performance negative result, that is showed greater than 50% reduction in performance for an at most 50% reduction in available resources. We note that the NAS kernels are not entirely independent observables of the dual core performance space, having certain commonalities in structure, size and systems (architecture) constraints at the time of authorship. For the NAS kernels operating in dual-core mode, we observe a shift in the effect of changing the page size employed on the Opteron™ processor.

Execution of the STREAMS suite indicates that even in highly memory-intensive kernels, there is a possibility for gaining greater utilization from the available fixed bandwidth resource.

As mentioned previously, the manner of execution of the NAS kernels (unlike the applications) may also affect overall performance, arising from different degrees of synchronicity between the cores in a socket. We would also draw attention to the fact that different degrees of optimization of a given application will lead to different effects of operation in dual-core mode—that is, a highly optimized code will make more efficient use of the memory channel in single-core mode (the reference point). This is somewhat analogous to the observation that a parallel application will scale more readily if compute sections are detuned, that is, the parallel component of execution time is artificially inflated.

Moving forward, we observe that the shift to multicore processors will not bring linear increases in performance with core count, but will bring very compelling benefits. The “memory wall” and associated mitigating programming techniques will be of increasing significance in that scenario. Offsetting the memory bandwidth disadvantages, we expect to see some positive benefits: extremely low latency and high bandwidth messaging between cores as multicore architectures adopt shared caches; the ability to rapidly re-load-balance algorithms by virtue of zero-cost relocation of data between processes sharing a socket in a parallel application; or highly efficient hybrid and global address space programming models exploiting flat access speeds to local memory and shared cache.

References

AMD Opteron™ Processor Technical Documentation
http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_9003,00.html

Software Optimization Guide for AMD64 Processors
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF

BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/32559.pdf

E. Strohmaier and H. Shan, "Apex-Map: A Global Data Access Benchmark to Analyze HPC Systems and Parallel Programming Paradigms," *Proceedings of SC|05*, Nov. 2005. LBNL-58409.

J. Weinberg, A. Snavely, M.O. McCracken, and E. Strohmaier, "Measurement of Spatial and Temporal Locality in Memory Access Patterns," *Proceedings of SC|05*, Nov. 2005. LBNL-58441.

E. Strohmaier and H. Shan, "Apex-Map: A Synthetic Scalable Benchmark Probe to Explore Data Access Performance on Highly Parallel Systems," *Proceedings of EuroPar2005*, Lisbon, Portugal, Aug. 2005. LBNL-58410.

S. Kamil, L. Oliker, and J. Shalf, "Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations," proceedings of the ACM Memory Systems Performance 2005 conference (MSP05), Chicago IL, 2005.
http://crd.lbl.gov/~oliker/papers/msp_2005.pdf