Appears in the 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)

Evaluation and Optimization of Multicore Performance Bottlenecks in Supercomputing Applications

Jeff Diamond[1], Martin Burtscher[2], John D. McCalpin[3], Byoung-Do Kim[3],
Stephen W. Keckler[1,4], James C. Browne[1]

[1]The University of Texas at Austin, [2]Texas State University, [3]Texas Advanced Computing Center, [4]NVIDIA

Abstract

The computation nodes of modern supercomputers commonly consist of multiple multicore processors. To maximize the performance of such systems requires measurement, analysis, and optimization techniques that specifically target multicore environments. This paper first examines traditional unicore metrics and demonstrates how they can be misleading in a multicore system. Second, it examines and characterizes performance bottlenecks specific to multicore-based systems. Third, it describes performance measurement challenges that arise in multicore systems and outlines methods for extracting sound measurements that lead to performance optimization opportunities. The measurement and analysis process is based on a case study of the HOMME atmospheric modeling benchmark code from NCAR running on supercomputers built upon AMD Barcelona and Intel Nehalem quad-core processors. Applying the multicore bottleneck analysis to HOMME led to multicore aware source-code optimizations that increased performance by up to 35%. While the case studies were carried out on multichip nodes of supercomputers using an HPC application as the target for optimization, the pitfalls identified and the insights obtained should apply to any system that is composed of multicore processors.

1 Introduction

The compute nodes of modern servers and supercomputers are typically constructed with multicore processors. However, many application codes have been migrated to these architectures with little or no optimization for the multicore environment. As a result, these programs can often only use a fraction of the available cores effectively and may not obtain maximum performance on the cores they do utilize. While performance optimization for multicore chips is receiving intense attention, a systematic methodology is still lacking. Detailed measurement and analysis studies are scarce, even though it is well established that the analyses and optimizations for performance bottlenecks on unicore chips are not sufficient for multicore processors [8]. Multichip nodes, where the chips are multicore processors, add even more complexity to performance analysis through NUMA and interference effects. As a result, diagnosis of performance bottlenecks on multicore-based systems is difficult and laborious, as are multicore targeted optimizations.

Early source-code optimizations tended to be CPU centric. Such optimizations focus on reducing computation, by employing strategies of lowering branch costs through unrolling or memoizing previously computed values. Over time, the increasing importance of the memory latency and bandwidth made data-access issues dominate. Typical optimizations for unicore chips are core-local in scope and target better use of caching through blocking arrays and padding data structures. Other optimization techniques involve reading and writing data sequentially or with constant strides to aid prefetchers. Comparatively little work has been done on non-local memory optimization techniques.

Due to shared resources in the memory hierarchy, multicore applications tend to be limited by off-chip bandwidth. At first glance, other optimization strategies would seem moot as all applications would simply run at the speed of memory. However, we found that when the memory system is saturated, the order and pattern of data accesses becomes a performance-determining factor. We call this state of operation the *multicore regime*. As discussed in Section 3, the same optimization can yield very different results in unicore and multicore regimes. For example, many optimizations that increase performance in the multicore regime can slow down uniprocessor code.

This paper makes the following contributions.

- It examines traditional unicore metrics such as cache miss rates and demonstrates how they can be misleading in a multicore system.
- It presents an in-depth study of performance bottlenecks originating in multicore-based systems. The study identifies and characterizes three important bottlenecks (shared L3 cache capacity, shared off-chip memory bandwidth, and DRAM page conflicts) that are exacerbated by multicore chip architectures.
- It describes performance measurement challenges that arise in multicore systems including unpredictable and unrepeatable memory behavior, execution skew across cores, and measurements that disturb program behavior. It suggests remedies for each of these challenges and incorporates the these remedies into a systematic process for multicore specific performance measurment.
- It introduces a source-code optimization called loop microfission that is designed specifically to alleviate

multicore-related performance bottlenecks. We observed a performance increase of up to 35% when applying microfission to a well-known supercomputing application.

We conducted this study using the HOMME atmospheric modeling benchmark code from NCAR, a complex application, running on supercomputers built upon AMD Barcelona and Intel Nehalem quad-core processors. While our studies were carried out on this single application, the process for measurement and optimization derived and the insights obtained apply broadly to systems comprising nodes of either single or multiple multicore chips that are used for executing computations that have predictable access patterns and good spatial locality.

The rest of the paper is organized as follows. Section 2 describes our methodology. Section 3 introduces performance issues that arise in measurement and optimization of multicore chips and multichip nodes. Section 4 presents the optimizations and results. Section 5 sketches related work. Section 6 provides conclusions and ideas for future work.

2 Methodology

The results reported in this paper are based on experimental measurements on two compute clusters: Ranger, a half petaflop AMD Barcelona-based supercomputer, and Longhorn, an Intel Nehalem-based supercomputer, both located at the Texas Advanced Computing Center. The HPC application we chose for our case studies is the High Order Method Modeling Environment (HOMME), developed by NCAR for their Climate Model 2 [10]. We used the performance tools gprof [9], mpiP [17], Pin [14], PAPI [21], perfctr [22], TAU [24], HPCToolkit [1] and PerfExpert [4] as well as the PGI and Intel compilers. This paper primarily presents our insights and results; additional data and details can be found in the accompanying technical report [7].

2.1 Systems

Ranger [23] consists of 3,936 16-way SMP compute nodes, each housing four 2.3 GHz AMD Barcelona-class Opteron quad-core processors for a total of 15,744 processors or 62,976 cores, and a theoretical peak performance of 579 teraflops. Each node has 32 GB of DRAM, of which each quad-core chip controls 8 GB. Nodes are connected by a 1 GByte/second InfiniBand network.

Longhorn [13] is a hybrid system with 256 nodes. Each node contains two quad-core Nehalem-EP processors operating at 2.5 GHz, between 48 GB and 144 GB of DRAM, and two NVIDIA Quadro FX 5800 GPUs. The nodes are connected via a QDR InfiniBand interconnect. For this study, only the 2,048 Nehalem cores were used since our focus is on performance of homogeneous multicore nodes.

Ranger's Barcelona chips support almost 500 hardware performance counter events. Moreover, Ranger has a wide spectrum of performance tools installed, which is why we chose it as our primary host for measurements. However, the Barcelona chips have known core scalability issues [15], whereas the Nehalem chips were designed to overcome these issues. Hence, we chose Longhorn as the secondary host to ensure that our methods, analyses, and insights extend to other processor architectures.

2.2 HOMME Benchmark

Many HPC applications can be categorized by three general computational patterns. *Regular parallel applications* have patterns of computations and memory references that tend to be predictable. Examples include finite difference PDE solvers, dense linear algebra solvers, and stencil codes. *Irregular parallel applications* have dynamically changing data structures, such as adaptive meshes, to handle levels of detail or time-varying geometry, but may still maintain reasonable locality through strategies such as irregular blocking. *Graph parallel applications* have their computation time dominated by graph traversal and no spatial locality is guaranteed. Examples arise in bioinformatics and the intelligence community. While load balancing and communication are classical performance bottlenecks for irregular and graph-based applications, as will be shown later, we are interested in key intranode scaling issues that are readily present in all three categories of parallel applications.

For this study, we chose to employ a single large-scale benchmark (HOMME), rather than study a suite of small-scale kernels. HOMME contains dozens of functions and a wide spectrum of loop structures implementing numerous algorithms. Therefore, we believe that the measurements and analyses executed on HOMME span a substantial fraction of the interesting space for regular HPC applications. HOMME is widely used by the supercomputing community and is one of the five HPC Challenge benchmarks [19] that are required for supercomputer acceptance testing.

HOMME (High Order Method Modeling Environment) [10] is an atmospheric general circulation model (AGCM) that provides 3D atmospheric simulation similar to the Community Atmospheric Model (CAM). The code consists of two components: a dynamic core with a hydrostatic equation solver and a physical process module coupled with sub-grid scale models. HOMME is based on 2D spectral elements in curvilinear coordinates on a cubed sphere combined with a second-order finite difference scheme for the vertical discretization and advection. It is written in Fortran 95 and is parallelized with both MPI and OpenMP. However, we are using the benchmark version of HOMME, which uses only MPI.

2.3 HOMME Scaling

HOMME is a very sophisticated and diverse example of a "regular" HPC application. It has been designed to scale well to tens of thousands of nodes and is highly optimized for computation and locality. In fact, it exhibits near perfect weak internode scaling (computation per core is constant) to tens of thousands of cores, and excellent strong scaling (total computation is constant), requiring a 900-fold increase in node count before the efficiency drops to half.[1] Despite all these advantages, HOMME's intrachip/intranode scalability, both weak and

[1] If p is the number of cores and n is the total size of the data, "strong scaling" means $n(p) = n$ and "weak scaling" means $n(p) = np$.

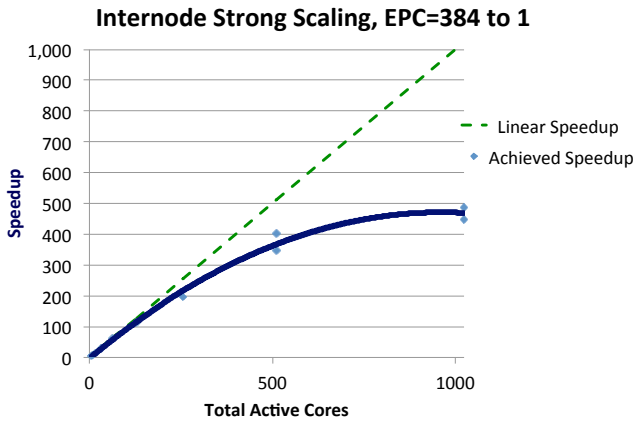

Figure 1: With fixed problem size, HOMME's node count can be increased 900 fold before the computation efficiency halves. The relative cost of communication increases linearly. Elements Per Core (EPC) describes the work done per core.

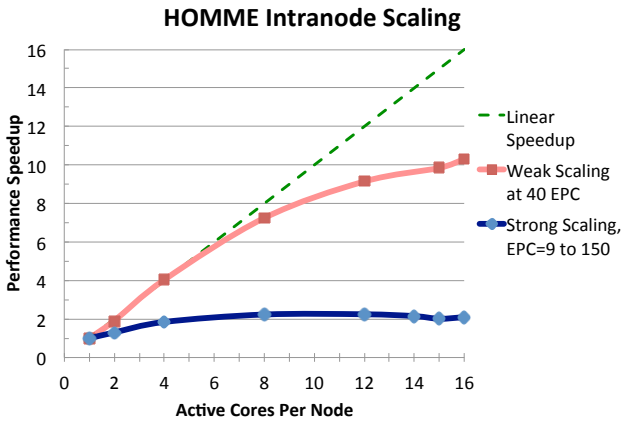

Figure 2: With intranode scaling, HOMME's efficiency falls below 15% when all 16 cores are used on a fixed problem size. Even weak scaling performance still drops by about 40%.

strong, can be three orders of magnitude lower than its internode scalability.

HOMME is only locally regular. The cube mapped sphere is specified at a certain resolution, and that grid is then broken into elements consisting of 8x8x96 regular meshes. Each element stores physical flow properties as a structure of arrays, requiring 48 KB per value that a given function accesses and up to 9 MB per element. A distributed, general graph connects each element with its four neighbors. When running, each active core is assigned a list of elements to process. Each core computes physical quantities for each element, then in a communication step exchanges boundary data with four neighboring elements. The available memory per compute node limits the possible amount of work per core from a single element up to 801 elements. We use the metric elements per core (EPC) to indicate the amount of work each core performs for a particular mapping of the application to the available cores.

To examine the strong internode scalability of HOMME, we keep the problem size fixed at the standardized "large" resolution of 1,536 elements, and then vary the number of active Ranger cores from one to 1024. We use 4 cores per node (1 core per quad-core chip) to isolate inter-node from intra-node scaling effects. The solid line in Figure 1 shows that, as the core count is increased and work per core drops, the efficiency (performance relative to perfect linear speedup) drops relatively slowly. The ratio of communication time to computation time increases linearly as work is spread across more cores. That the efficiency drops slowly as the communication overhead increases demonstrates the excellent strong scaling characteristics of HOMME across multiple nodes.

To examine weak scaling, we chose an input data size as close as possible to the standardized "standard" workload of 54 elements per core. Loads other than 54 elements per core were sometimes necessary to ensure that the total number of elements is divisible by the number of cores, removing issues of load balancing and interference. We found that HOMME's internode weak scaling is ideal, never falling below 94% of linear speedup. This scalability is a result of using only nearest neighbor communication.

For intranode strong scaling, we chose a fixed resolution that roughly centered on the standard work load of 54 elements/core. For weak scaling, we chose a constant 40 elements/core, the closest we could get to 54 elements/core without imbalancing the load. Sensitivity tests showed that intranode scaling properties were strongly independent on the load per core.

Figure 2 shows the strong and weak intra-node scaling profile from 1 to 16 cores in a four-chip node. For weak intranode scaling, the speedup at 4 cores per chip is 60% of linear and falls off rapidly as more cores are used. For strong intranode scaling, performance increases very slowly when increasing cores per node, and provides little performance benefit beyond 4 cores per node. Regardless of the scaling regime (weak versus strong), improving single-node multicore performance would provide a substantial benefit to the performance of the application, independent of the inter-node scaling characteristics.

Our primary motivation in this paper is to investigate causes that prevent HPC applications with excellent conventional scaling properties from being able to utilize entire multicore supercomputers. If HPC applications cannot make good use of quad core chips now, how will they make use of 8, 12, or even 16 cores per chip in the future?

3 Multicore Performance Analysis

This section explores the ways that conventional performance bottlenecks, metrics, and measurement error may be qualitatively and quantitatively different for multicore processors. Section 3.1 examines the basics of good multicore performance and why classical unicore performance metrics may be misleading. Section 3.2 describes three fundamental architectural bottlenecks unique to multicore chips that are the primary barriers to intranode scaling, new performance metrics needed to classify each bottleneck, and some common code styles that may exacerbate them. Section 3.3 describes how measurement artifacts

change with multicore chips, providing a detailed categorization of memory measurement issues, the impact of *core skew*, and the increased importance of lightweight measurements. The final section summarizes a systematic approach to multicore performance analysis.

3.1 Multicore Performance Metrics

This section introduces a fundamental approach to measuring multicore performance along with comments about general performance metrics. It then demonstrates why traditional unicore performance metrics, such as L1 and L2 cache miss rates, do not adequately capture multicore memory issues.

3.1.1 Isolating intranode scaling effects

To simplify our experimental search space, we first determined how small a subset of Ranger we could employ to study intranode scaling. We varied both the number of nodes and the number of cores per node, and found that the efficiency (performance as a function of total cores) was largely independent of the number of nodes and extremely dependent on cores per node. This feature is a testament to the internode scalability of HOMME and enables studying intranode scaling using as few as four 4-socket nodes. For the rest of the paper, we employ a total of 16 threads and spread them across 1 to 4 nodes, keeping the total work constant. We use a *density scaling* metric in which *core density* is defined as the number of cores used per active socket; core density varies from 1 to 4 in our experiments. We then used gprof to determine which of HOMME's procedures contribute most to the execution time and which of these functions suffer most from intrachip scaling issues.

3.1.2 Metrics for "good" performance

The performance of each function of an HPC application can be characterized by three rate metrics: Flops Per Cycle (FPC), the rate of algebraic computation; Instructions Per Cycle (IPC), how hard the CPU is working; and Loads Per Cycle (LPC), how hard the memory system is working. If any of these metrics approach the expected maximum for that architecture, then performance is good and further improvement must be accomplished through algorithmic optimizations. Advertised peak rates for performance metrics in most chips are well-known to be unrealistic. However, reasonably attainable peak rate for application programs can be derived from targeted micro-benchmarks written in high-level languages; these empirically measured rates should be used for assessing performance of application codes. Note that the ratio of these three metrics to each other is fixed by the dynamic instruction mix, so generally only one of them can reach the hardware's maximum. If all three metrics are sub par, then in most cases the performance bottleneck is in the memory system.

Figure 3 shows these three metrics on Ranger for the most important functions in HOMME. For most programs, a combination of a high IPC, a high FPC and a high LPC indicate good performance. An IPC value of 2 is very good for Barcelona chips. In the middle of the graph are three functions with dramatically higher values on all performance metrics that are CPU-bound. The highest performance levels reached by these three CPU-bound functions is roughly 1.8 IPC, 1.3 FPC, and 0.5 LPC. These functions also simultaneously achieve the highest memory performance (LPC) as well. The remaining eight functions are memory bound, with an FPC of 0.1 or less and similarly low IPC and LPC. The multicore optimizations described in Section 4, which are aimed at improving the LPC, increase the FPC value to 0.25 FPC, still far below the reasonably attainable peak rate but more than a factor of two over the FPC of the original code.

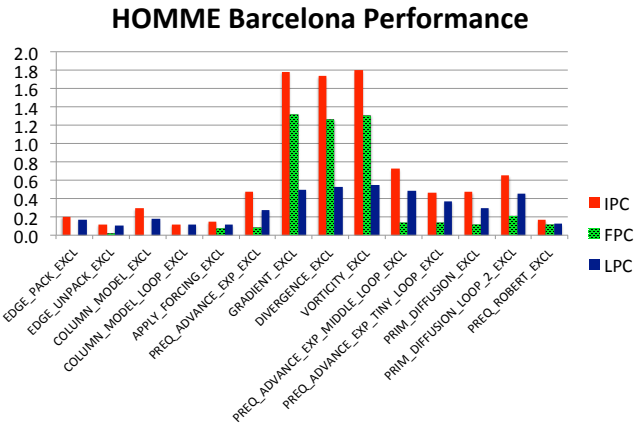

Figure 3: Performance metrics for HOMME's major functions.

3.1.3 Traditional unicore metrics are insufficient

In traditional performance analysis, L1 and L2 cache miss ratios typically provide insight into suspected memory bottlenecks. Furthermore, computation optimizations are often assumed to be irrelevant to performance, since the processor is waiting idly for data. However, we found that on modern multicore systems, these *core local* performance metrics were not good indicators of multicore performance issues for three reasons.

L1 miss ratios may be misleading: A low L1 cache miss ratio by itself is not indicative of good memory performance on many chips because of the way in which hits are counted. When a load accesses a missing cache line, it counts as a miss, but all subsequent loads from that same cache line (which typically holds eight double-precision floating-point values) count as a hit, even before the hardware has been able to bring the cache line into the cache. By itself, this way of counting lowers the effective miss ratio to 12.5%, assuming eight values per cache line, for a regular stride-one application with all misses. In the presence of hardware prefetches, all accesses to a cache line that is not yet in the cache may be counted as hits, resulting in no L1 misses being recorded at all. Finally, once the (relatively small) maximum number of outstanding loads supported by the CPU has been reached, all further loads are stalled. The subsequent delay can be at least as long as a cache miss without being counted as one. As a consequence, a modern application can have great L1 performance but severe bottlenecks lower in the memory hierarchy. In fact, prefetching makes it fairly common to see functions with nearly 100% L1 cache hits actually fetch-

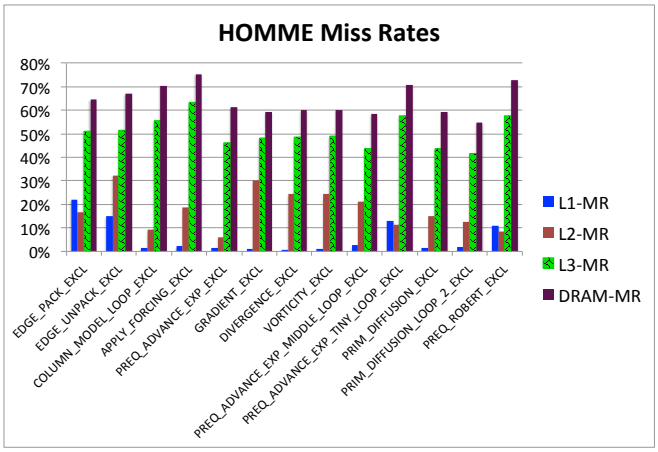

Figure 4: Miss ratio variability of major HOMME functions across the entire memory hierarchy.

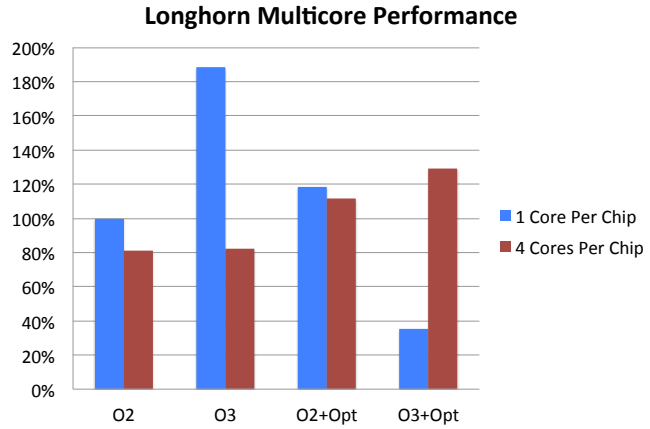

Figure 5: Effect of compiler optimization flags on Longhorn. "Opt" is a multicore optimization involving microfission and blocking arrays for the L1 cache.

ing every value from DRAM and running at DRAM throughput rates. Due to bandwidth limitations of current CPUs, such direct streaming algorithms often suffer a 10-fold slowdown relative to the expected performance.

L2 cache miss ratios may not be predictive: We initially tried to infer memory bottlenecks by looking at the way the core density affected the L2 miss ratios. We assumed that if the L2 miss ratio was low or did not vary much with density, the system did not likely suffer from significant L3 and DRAM scalability problems. This assumption turned out to be false for functions with poor intrachip scalability, but could be corrected by looking at a few additional metrics such as L3 hit and miss ratios and DRAM page hit ratios. We found that the actual L2 traffic can be significantly higher due to prefetching, load replays, and coherence snoops, which we observed to account for as much as 80% of the L2 bandwidth, even in pure MPI codes without any direct data sharing between processes. Figure 4 illustrates the miss rates at all levels of the memory hierarchy for the most important functions in HOMME. Correlating the heights of the bars reveals two important multicore effects: (1) L3 and DRAM miss rates are much higher than L1 and L2 miss rates and (2) the degree of L1 and L2 miss rates is not predictive of L3 or DRAM miss rates, and in fact is often uncorrelated with intrachip scalability. Miss rates increase further down the memory hierarchy, since locality is being skimmed by the higher caches.

Conventional CPU optimizations may be counterproductive: Applications are typically memory bound when multicore performance problems are present. However, CPU metrics and optimizations are still relevant because in the *multicore regime*, the exact order of load instructions has a first-class effect on performance. As such, many CPU unicore optimizations actually have the potential to hurt performance in the multicore regime. Figure 5 shows compiler flag effects on a baseline and aggressive multicore fission and blocking optimization on PreqRobert, as described in Section 4. Aggressive compiler optimizations such as "-O3" nearly doubled performance in the unicore regime of 1 core per chip, but the combination of optimizations drastically reduced performance in the multicore regime at 4 cores per chip by creating irregular access patterns. This example quantifies the degree to which CPU optimizations may behave differently depending on core density. The degraded performance stemming from the irregular access patterns generated by higher levels of unicore compiler optimization is why HPC users commonly use the -O2 instead of the -O3 optimization level.

In our experiments with the two compilers available on Ranger (PGI and Intel's icc 10.1), we also found that different compilers produce code with very different memory access patterns from the same source code. Specifically, we observed that the performance metrics for individual functions may vary by up to 40% between compilers, and metrics like cache and DRAM miss rates can vary by 3x.

3.2 Multicore Bottlenecks

This section describes three main architectural resources that we found to cause scalability issues in multicore chips: L3 cache capacity, off-chip bandwidth, and DRAM banks. This section also discusses how to measure and differentiate the effects of each of these bottlenecks, an important process as the optimizations for each bottleneck can differ.

Generally, issues in the memory system flow downwards: L3 capacity issues increase off-chip bandwidth demands, and off-chip bandwidth may exacerbate DRAM bank misses. However, it is still possible to experience any of these issues as the primary bottleneck.

3.2.1 L3 cache capacity

As seen in the previous section, the causes of poor intrachip scaling cannot be resolved from only L1 and L2 miss ratios. Further complicating measurements, the performance counters needed for L3 measurements and beyond are typically not supported directly in general purpose performance tools such as PAPI. We were able to use native hardware counters as defined in processor user guides [2, 11]. Fortunately, all of the tools we used for our case studies support passing native counter IDs to the hardware.

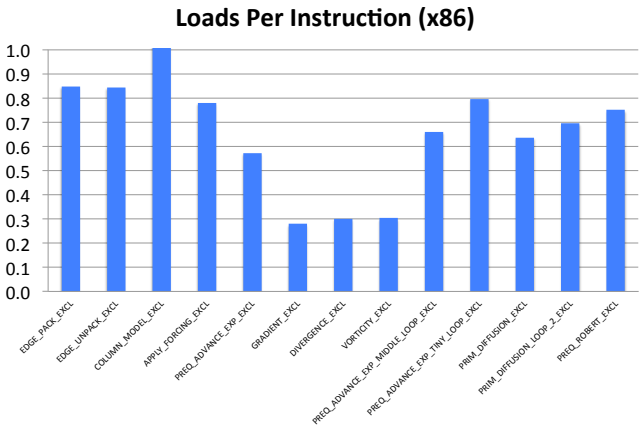

Figure 6: Loads Per Instruction: HOMME has an arithmetic intensity typical of HPC programs, with 60%-100% of the instructions accessing memory in 8 of the 11 most important functions.

Diagnosis: If the L3 miss ratio significantly increases when going from the minimum to the maximum core density, it is likely that at least part of the performance problem is the L3 capacity, since each core can now only use a fraction of the shared cache. Note that the effective cache capacity can further be reduced by associativity issues and false sharing between processes. Since L3 cache misses directly translate into off-chip (i.e., node-level) bandwidth, applications with L3 capacity issues also tend to have memory bandwidth issues. Common solutions involve reducing a function's cache footprint, either in absolute terms or ephemerally, by using data serially. Some common pitfalls are listed next.

Storing intermediate values to reduce computation. When cache capacity is an issue, a good optimization is to reduce the memory footprint of a function by reducing or reusing temporary variables and redundant arrays. DRAM streaming speeds are 10 to 20 times slower than normal instruction execution, leaving ample time to compute values redundantly instead of storing them. Interchanging or promoting loops to minimize passes over the data can boost arithmetic intensity. Although these types of optimizations are worthwhile, they may be infeasible for large programs or when a programmer has limited familiarity with the code.

Operating on many values at once, instead of one at a time. An even easier optimization which does not require knowledge of the code is described in Section 4, in which a technique we call loop *microfission* results in drastically reducing the short term memory footprint of an application without effecting the algorithm.

3.2.2 Off-chip bandwidth

Off-chip bandwidth is now recognized as a first order bottleneck with multicore chips. Figure 6 illustrates LPI, or Loads Per Instruction, for the key functions in HOMME. More than half of the key functions require a memory access with almost every instruction executed. Because the x86 instruction set allows most instructions to include a memory operand, the LPI metric can effectively double as compared to a load/store architecture. HOMME is typical of HPC applications, which commonly have little reuse of data values and therefore do not benefit much from a cache hierarchy. The off-chip memory accesses can be estimated by the L3 misses or DRAM data accesses. It can also be instructive to look at a function's "taper", defined as the number of bytes a given function needs to read per instruction at a given level of the memory hierarchy, typically off-chip DRAM. Most studies focus on bytes read from off-chip DRAM per instruction. Many functions in HOMME require one in three memory accesses to go off-chip, which can be used to determine the suitability of a given supercomputer to run a given HPC application.

Averages can be misleading. The performance weighted average for HOMME is just 0.8 bytes/flop off chip, but this average includes a significant number of functions that have no off-chip traffic. Peak values are critical to maintaining average performance, and we therefore recommend studying functions individually.

Determining a function's off-chip bandwidth needs requires minimum thread density. The bandwidth requirements of a function can only be judged when it is running unconstrained at one process per chip. If examined at full core density, it may paradoxically appear that the function's bandwidth usage has decreased, yet performance is worse.

Diagnosis: The magnitude of bandwidth bottlenecks, if any, can be gauged by the amount the function exceeds its share of off-chip bandwidth when running at 1 core per chip, which is the total off-chip bandwidth divided by the number of cores per chip.

3.2.3 DRAM pages

Perhaps the least studied multicore performance bottleneck involves contention over DRAM pages (not to be confused with OS pages.) A DRAM page represents a row of data that has been read from a DRAM bank and is cached within the DRAM for faster access. DRAM pages can be large, 32 KB in the case of Ranger, and there are typically two per DIMM. This means that a Ranger node shares 32 different 32 KB pages among 16 cores on four chips, yielding a megabyte of SRAM cache in the main memory.

Diagnosis: Most systems have straightforward counters to estimate the total number of DRAM accesses and DRAM page hits, which may be significantly greater than the number of L3 misses due to prefetched data. The performance effect of DRAM page misses has become a first order concern in the multicore regime due to the following reasons.

DRAM contention can significantly impact performance. When a DRAM request is outside an open page, the page must first be written back to DRAM (closed) and the next page read out (opened), which adds 30 ns (69 cycles at 2.3 GHz) to the access time and reduces DRAM performance. As an optimization, the DRAM controller will attempt to close a row (write back the results) after a certain amount of time so the next access only needs to open a row, saving 15 ns. While this represents only 20% of the access time on Barcelona chips, newer processors have been reducing absolute latency to DRAM, so the impor-

tance of page misses will grow with time. As the number of cores per node grows, contention will increase, and making effective use of DRAM pages will be more difficult. More significant than this individual increase in latency is the reduction in effective DRAM bandwidth when DRAM bank miss rates are high. It is not uncommon to have a more than 50% DRAM bank miss rate when multiple cores contend for banks. While DRAM controllers have buffers and schedulers to mitigate these issues, they tend to by myopic due to limited buffer space.

DRAM Optimization is not necessarily complex. Many papers in the literature have focused on optimizing memory controllers and thread schedulers to reduce DRAM page contention, but few have explored the potential of employing high-level code transformations to reduce DRAM conflicts. Managing DRAM locality at the program source level is difficult because we cannot easily control where memory is physically allocated, where those physical pages reside in the node, or when exactly the cores on a node access different pages. However, high-level code transformations can alter the locality in the access patterns to reduce the average number of conflicts and thereby increase performance. Section 4 illustrates the most important multicore optimization we found, using specially targeted loop fission to reduce DRAM page conflicts.

3.3 Multicore Measurement Issues

The previous subsections described *what* is important to measure. However, these new scaling and optimization behaviors resulting from multicore architectural bottlenecks also lead to performance *measurement* issues that are not merely different, but that can be extremely difficult to overcome. This section categorizes the ways in which multicore measurement issues fundamentally differ from traditional performance measurements, and follows with a description of the disturbance impact of *core skew* and techniques required to minimize measurement disturbances.

Classical optimizations focus on CPU performance and CPU monitoring. Their primary metric is speed, and aside from uncertainty regarding out-of-order execution, timing code could simply subtract out its own execution time and have minimal disturbance on running code. Although memory performance is critical for a wide span of applications ranging from HPC to databases that tend to have low arithmetic intensity (Figure 6), multicore scalability issues tend to arise from complex interactions of a hierarchical memory structure contested by many cores. This observation leads to the following fundamental difficulties when trying to trace a memory performance issue to its cause in software.

- Memory effects are highly context sensitive. They depend on the state of all the caches and DRAM banks in the system. The performance of a given function that always performs the same amount of computation will tend to have a much higher degree of variability, and depend much more on its calling context, since performance will depend on the data the calling functions have brought on chip and the state of the memory system upon entering the function.

- Memory bottlenecks tend to be bursty. This results in extreme local variations in performance that might be hidden in larger averages.

- Memory interactions are nondeterministic. Just as debugging parallel programs can be hard due to nondeterminism, memory bottlenecks caused by the interactions of threads on the memory system also tend to be nondeterministic.

- Memory bottlenecks can be highly non-local. Because the latency of the memory system can be hundreds to thousands of cycles, and because memory access speed is affected by the activities of previous memory operations, an apparent memory bottleneck often is caused by earlier functions. Additionally, the natural time skew between the activities of different cores and chips tend to spread the range of memory effects far beyond the time of complete cache turn over, the time it takes for all data in the on-chip memory hierarchy to be replaced with new data.

- Disturbing memory behavior in the attempt to measure it is much easier in the multicore regime, due to the difficulty in bracketing memory effects. The influence of a performance measurement may be felt much later, while the counter events recorded during the actual timing calls may have been caused by code in an earlier timing interval or may even have been influenced by a different core running different code. Any sophisticated timing library will need to make multiple round trips to main memory to update counter totals, resulting in time dilations of thousands to millions cycles, which along with such effects as prefetching and cache interference can create disturbances in the entire memory system that last for tens of billions of cycles. More details on this appear in the following subsection. For this reason, we have found that developing or utilizing extremely light weight timing libraries is required to see an optimization's effects on the running code and not simply the optimization's effects on the timing code.

As a result of these fundamental issues, the notion of a given function having "average performance characteristics" is less certain, and the performance effects of the measurements themselves are much more difficult to isolate and remove. The next subsection discusses these issues in more detail and suggests techniques to overcome them.

3.3.1 Effects of Core Skew

An important performance effect in the multicore regime is what we call "core skew" effects. Because each core might be in a slightly different phase of execution, different functions may be running on different cores at the same time. This effect can be due to natural drift between synchronizations, such as from NUMA effects or non-deterministic memory contention, or due to an intense memory event having a somewhat serializing effect, pushing the cores further out of phase. Finally, this effect can be caused by a single core performing unique duties, such

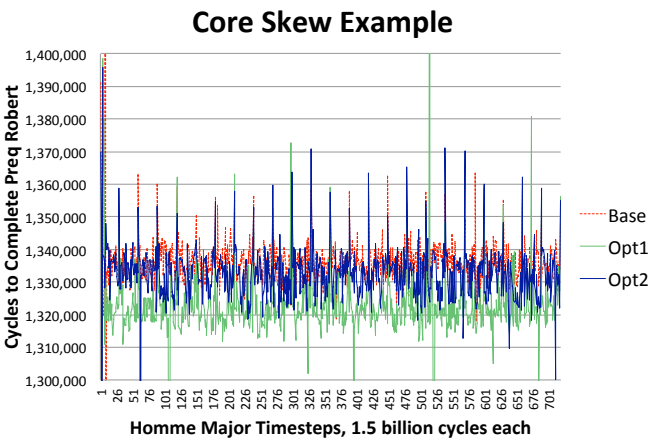

Figure 7: Core skew causes significant jitter and prolongs disturbances for tens of billions of cycles, making temporal context critical in measurement.

as printing results or doing performance monitoring, that kicks that core further out of phase relative to the others. The effect is that an intense event gets "smeared out" over time, because it remains in operation from the time the first core starts the section until the last core finishes it. At a phase change boundary, light computational functions may slightly overlap with memory intensive functions, slowing their performance.

Figure 7 shows all these effects in HOMME. The figure demonstrates the average execution time of a single function, PreqRobert, for each major computational timestep, across a trillion cycle window. These are not short term performance fluctuations, but averages of hundreds of executions over more than a billion cycles. In particular, the extreme oscillations caused by program initialization (not shown on graph) still cause performance oscillations of 35% (clipped in graph) tens of billions of cycles later. The latent perturbations are large enough to throw off total performance averages dramatically. Note that these perturbations last many times longer than the complete turnover of all data in the memory system, which occurs many times per major timestep, and even longer than a single major timestep, during which all cores resynchronize execution through a local barrier.

In addition, because the continuous performance oscillation (on the order of a few billion cycles, or 3 timesteps on Ranger and 8 timesteps on Longhorn) is larger than the difference in average performance, average performance values do not correctly convey which optimization is better. While this graph demonstrates background oscillations of just a few percent, we have observed periodic oscillations up to 20%.

Many people already know to fast forward over the initialization and cache warm-up time. But in the multicore regime, one must fast forward *tens of billions of cycles* into regular code execution for the skew disturbances to die down. It is also critical to minimize clock skew by having all cores do as similar operations as possible.

Performance oscillations and extreme perturbation delays

function size	% exec time
2,000 cycles or less	20%
2,000 to 10,000 cycles	10%
10K to 200K	15%
200K to 1 million cycles	15%
1 million to 10 million cycles	0%
10 million or more cycles	35%

Table 1: Duration of important HOMME functions

combine and make it critical for performance measurements to preserve temporal context. The variation spikes on the left side of Figure 7 are so high that the average performance of many key functions depends on exactly when the timing starts. This is why for HOMME, temporal context was much more important than calling context. Another classic example where temporal context is critical is when touching a page for the first time causes TLB misses with page initializations taking 25,000 cycles each. If the system in use does not support huge pages, initialization can consume a large fraction of execution time.

Because conventional instruction logging is completely impractical for functions that recur tens of millions of times a second or more, we developed a hierarchical scheme in which performance metrics were averaged over sub millisecond periods, and then these period averages were used to accumulate statistical information about variation with time.

3.3.2 Minimizing measurement disturbance

Many measurement tools are very heavyweight, but rely on the concept of sampling versus explicit measurements. The idea is to mitigate huge perturbations to the code by only doing the measurement once in a while. As a result, it is not uncommon for many tools to dilate execution time several, if not dozens of times over [1]. While this approach will always work if only a single measurement is taken during an entire run, it can lead to serious disturbance issues when averaging many samples over many functions by completely changing the behavior of the memory system. This paper has already shown that memory disturbances can last for billions of cycles. The result is that perceived bottlenecks and optimization solutions are actually optimizing the measurement code.

On modern CPUs, cycle latencies are much larger. A single assembly instruction to read one performance counter takes 9 cycles. Reading 4 counters at once takes 30 cycles and is done twice per interval. A user function call can take 40 cycles. A running total must be updated, and in typical HPC functions, will be flushed from the cache between calls. Even a single low level user call to PAPI_READ to get a performance counter value takes 400 cycles, and a system call can take 5,000 cycles. Yet, even these calls are trivial compared to more heavyweight timing code that may take tens of millions of cycles and completely disrupt the memory system.

In contrast to this trend, Table 1 shows the distribution of lifespans of the most important HOMME functions. Roughly half of HOMME's run time (45%) is spent in very small functions, with 20% of execution time spent in functions only 2,000

cycles long, less than a single microsecond. Note that only the largest category in the table includes functions that run for 1 millisecond or longer. Additionally, short lived functions are called with extremely high frequency, with some exceeding tens of millions of calls per second. And the effect of core skew means that these functions exhibit a large variation in performance characteristics. While we found conventional measurement techniques would work for relatively large functions taking several milliseconds or longer, about half the important HOMME functions require special measurement care to avoid arbitrarily wrong measurements. This is even more essential when evaluating potential code optimizations.

The important message is that correct assessment of bottlenecks requires utilizing multiple performance analysis tools to get qualitatively correct answers. Higher level measurement tools, which are simple to use but extremely heavy weight, are convenient for initial classification and for analysis of long running functions of tens of milliseconds or more. But once small, important functions are discovered, proper analysis requires a more delicate and targeted measurement approach such as custom PAPI or PerfCtr calls, or using lightweight or targeted tools like gprof and TAU. We provide a brief summary of our tool experiences below.

We found gprof [9] was the easiest tool to use and had the lowest overhead of any available tool aside from our custom PerfCtr code. gprof uses statistical sampling, but was accurate for all but the three smallest functions in HOMME. Tau had the lowest overhead of any explicit sampling tool, and could handle any kind of sampling context [24]. HPCToolkit had high overhead, but was the only tool that found code hot spots in arbitrary loop nests [1]. In fact HPCToolkit was responsible for isolating three of the top 11 most important code regions. PerfExpert [4, 6] is a new tool designed to be as easy to use as gprof, and it worked reasonably well on this task. However, for our most difficult measurement tasks, we resorted to code instrumentation to measure performance counters with low enough disturbance to see meaningful results for the actual code and optimization effects, instead of just measuring the performance measurement code. We found we sometimes needed this approach even on medium sized functions taking up to a million cycles.

3.4 Multicore Analysis Summary

Based on the general challenges highlighted in the previous sections, we have distilled out the following recipe that provides a systematic approach to optimizing performance on multicore systems.

- Begin the analysis in a traditional way, i.e., by determining a representative run configuration and using a tool like gprof to identify the most important functions by total execution time. Running this test at minimum and maximum core density will determine the functions with the poorest intracore scalability.

- Compare the IPC, FPC and LPC values for these functions at maximum core density to good values for the underlying system to determine the optimization headroom. For functions with poor performance, gather multicore performance counter information such as L3 and DRAM miss rates for minimum and maximum core densities.

- If the L3 miss rates increase with core density, focus on optimizations that minimize cache footprints or temporary variables, or serialize data accesses.

- If there are functions that, at minimum core density, greatly exceed their share of off-chip bandwidth, but capacity is not an issue, focus on loop interchanges that may allow greater data reuse, or replace stored data with redundant computation where possible.

- If there is an increase in DRAM bank miss rates with core density, focus on rearranging loops to access only a single array at a time.

- For medium and large functions that take multiple milliseconds or longer to execute per invocation, any convenient measurement tool, such as TAU, HPCToolkit or PAPI will suffice. Shorter functions require minimal overhead timing approaches, such as a selective TAU run or PerCtr code. Averages should be checked for variations at different points of the execution.

4 Multicore Optimizations

The goal of performance measurement and analysis is usually optimization. We consider two categories of optimizations: algorithmic optimization, in which the algorithms are changed, and source-code optimizations, which are typically local to a loop nest or function and leave the algorithm unchanged. Source-code optimizations often require little knowledge of the algorithm and are simpler to implement. Some source-code optimizations can be automatically applied by the compiler.

The previous section identifies and establishes procedures for measuring and analyzing three multicore-specific performance bottlenecks: L3 capacity, off-chip bandwidth, and DRAM page misses. Algorithmic optimizations for alleviating these bottlenecks include making an algorithm more computationally heavy or helping reduce off-chip bandwidth and bus contention. But algorithmic optimizations often require more detailed program comprehension, spread across large code sections, and may not be portable across applications. Therefore, we concentrate on source-code optimizations.

The most effective local multicore optimization we found is *microfission*, which is a specialization of two well-known compiler optimizations, loop fission and loop fusion [3]. This optimization first splits (fissions) complex loops that reference multiple arrays into simple loops such that no more than two independent arrays are accessed in each loop nest, and no more than one array is brought in from main memory per loop nest. (Each loop nest is of the form $C[i] = f(C[i], X[i])$, where $C[i]$ is cached and $X[i]$ is streamed in.) Then, wherever possible, the optimization combines (fuses) loop bodies that operate on the same two arrays. For example, Figure 8 shows a key loop in

```
do k=1,nlev
 do j=1,nv
  do i=1,nv
   T(i,j,k,n0) = T(i,j,k,n0) + smooth*(T(i,j,k,nm1) &
                 - 2.0D0*T(i,j,k,n0) + T(i,j,k,np1))
   v(i,j,1,k,n0) = v(i,j,1,k,n0) + smooth*(v(i,j,1,k,nm1) &
                 - 2.0D0*v(i,j,1,k,n0) + v(i,j,1,k,np1))
   v(i,j,2,k,n0) = v(i,j,2,k,n0) + smooth*(v(i,j,2,k,nm1) &
                 - 2.0D0*v(i,j,2,k,n0) + v(i,j,2,k,np1))
   div(i,j,k,n0)  = div(i,j,k,n0) + smooth*(div(i,j,k,nm1) &
                  - 2.0D0*div(i,j,k,n0) + div(i,j,k,np1))
  end do
 end do
end do
```

Figure 8: Loops in HOMME typically iterate over many different arrays at the same time (code shows loop from PreqRobert update).

```
do k=1,nlev
 do j=1,nv   ! Load T(i,j,k,n0) into cache
  do i=1,nv  ! May need to block across all loops in T
    T(i,j,k,n0) = (1.0 - 2.0*smooth) * T(i,j,k,n0)
  end do
 end do
end do

do k=1,nlev
 do j=1,nv
  do i=1,nv
   T(i,j,k,n0) = T(i,j,k,n0) + smooth * T(i,j,k,nm1)
  end do
 end do
end do

do k=1,nlev
 do j=1,nv
  do i=1,nv
   T(i,j,k,n0) = T(i,j,k,n0) + smooth * T(i,j,k,np1)
  end do
 end do
end do
```

Figure 9: Applying microfission to the first line of the loop body in Figure 8. At any one time, one array stays in the private cache while a second array is streamed in.

HOMME that accesses four different arrays and 24 different array sequences in a single loop. Figure 9 shows how the first line of the loop body is broken up so that each time only one array is brought into the private cache and at most one array is kept in the private cache.

This optimizations offers two critical benefits to code executing on a multicore chip or multichip node. First, it reduces the loop-level working set to two arrays. This may significantly reduce L3 cache misses. Second, it reduces the total number of independent locations being requested from the memory system across all the cores in a node. The result is a reduction in the number of DRAM page misses, while empowering the memory controllers to batch requests more intelligently. Finally, compilers are good at optimizing small loop bodies and thus end up producing better code once microfission has been applied.

A practical complication is that modern compilers are "smart" enough to fuse small loops and undo the optimization. To save the microfission optimization from such compiler transformations, we encapsulate each micro-loop in a separate function. While this process introduces a substantial CPU overhead (which could be avoided by better control of compiler optimization), it still improves performance. This result clearly suggests defining a compiler switch or pragma to disable loop fusion.

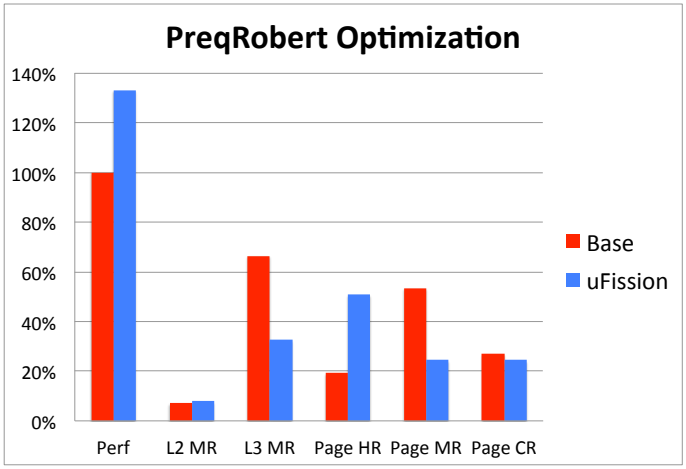

Figure 10: Effect of performing the microfission optimization: L3 miss rate and off-chip BW cut in half, DRAM page hits more than doubled, and performance increased by 35%

One important assumption of microfission is that two arrays fit completely in one core's share of the on-chip caches. For the HOMME data sets used, this is the case, as each array for a single blocked element takes up just 48KB. If the arrays were too large to fit in the caches, blocking would be necessary.

The performance effects of microfission are illustrated in Figure 10 as measured on Ranger. The second set of bars (L2 MR) show that the L2 miss rate is essentially unaffected, increasing slightly from 7.4% to 7.9%, despite the increased private cache bandwidth incurred by the optimization. This confirms that private cache bandwidth is ample to support the lowest arithmetic intensity HPC codes, and that microfission is a multicore optimization having minimal effect on individual cores. The next bars to the right (L3 MR) illustrate the reduction in contention for shared on-chip resources. L3 miss rates, typically above 50% for HPC applications, are cut in half from 66% to 33% due to the local drop in working set size. Off-chip bandwidth needs are reduced accordingly. The DRAM page hit rate (bars labeled Page HR), previously below 20%, is 2.5 times as high, breaking the 50% mark, while the page miss rate (Page MR) is down from 53% to 25%. DRAM conflict miss rates (Page CR), representing the worst type of DRAM contention, are down by a more modest 11% from an already low rate of 27%. As a result of reducing resource contention, actual performance (the first set of bars) increased by 33% on Ranger and by 35% on Longhorn.

Microfission also improves intrachip/intranode scaling. Comparing the optimized code at 4 cores per chip to the base code at 1 core per chip, we see a 33% increase in scalability – on Ranger/Barcelona, the optimized code reaches 78% efficiency at 4 cores per chip and actually manages to benefit from 3 cores per chip. On Longhorn/Nehalem, the optimized version has 10% better intrachip scaling properties over the unoptimized version. However, comparing the optimized version at 4 cores per chip over the unoptimized base at 1 core per chip, the efficiency in-

creases to 58%, a 42% improvement in intrachip scalability that is able to benefit from all 4 cores.

The computational pattern that enables microfission, i.e., multiple terms in an equation discretized on the same grid, is common across regular HPC applications so that microfission should enhance the performance and scalability of many such applications. The detailed measurement and analysis of the performance of HOMME on multicore chips and the success of microfission suggests some guidelines for coding regular applications for multicore chips: (i) employ structures of arrays rather than arrays of structures, (ii) group all computations on a logical data structure in the same loop as much as possible, (iii) minimize the number of temporary arrays that are declared and used, and (iv) adjust array/loop block sizes to fit in the caches that are characteristic of multicore chips.

5 Related Research

The literature is full of work on optimizing specific scientific kernels for multicore CPUs, including stencil computations [5], sparse matrix-vector multiplication [25], and even a Lattice-Boltzmann computation [20]. Development of the STREAM benchmark suite [15, 16], which demonstrates highly optimized memory performance, was achieved based on detailed architectural knowledge. Mytkowicz et al. analyzed SPEC benchmarks using PAPI to read hardware counters and noted that measurement error due to measurement perturbation, OS context, and compiler flags can be so significant and unpredictable that correct performance conclusions cannot be easily drawn [18]. In fact, our study of a full-scale application in the context of a modern multicore supercomputer showed almost an order of magnitude greater perturbation. Their optimizations are measurement instead of code based – they recommend running tests over a wide range of OS parameters and validating any performance conclusions by conducting detailed tests to confirm hypotheses. Our research goes into more depth about measurement errors and how to avoid them or at least limit their impact.

Dongarra et al. focus attention on the impact of multicore architectures on scientific applications [8]. For example, they observed that multiple cores on a chip cannot be treated as a traditional SMP due to shared on-chip resources, and, as a result, new scientific code would become much more complex because it would have to take increasingly varying architectures into account. Whereas they further identify potential difficulties in programming and compilation for multicore architectures, they do not focus on issues with performance measurement.

Jayaseelan et al. investigate various performance characteristics of three compilers on the integer SPEC benchmarks, focusing on PAPI hardware counters as primary metrics [12]. They observed that different compilers do better in different functions, and that judging a compiler only by total program performance can be a mistake. They also note compiler differences in cache miss rates and fundamental instruction counts and recommend some optimizations. Our research focuses more on memory and less on CPU optimizations. We note the impact of the compiler on memory access patterns in the multicore regime and recommend fundamental but simple changes to the way compilers optimize code for multicore processors.

The previous research that most directly addresses bottleneck analysis and optimization approaches specific to multicore chips is the Roofline model from Berkeley [26]. This model defines a number of key performance metrics and uses microbenchmarks to estimate realistic values for a given hardware platform. Comparing the actual performance of several HPC proxy kernels (the original seven Berkeley dwarfs) then provides insight into multicore bottlenecks and code optimization techniques on a half dozen single-chip multicore systems. While insightful and useful, the Roofline paper focuses on a much coarser level, examines a different scale of programs, and targets more traditional optimization techniques like blocking and CPU-centric approaches such as successful vectorization and balancing multiplies with adds. It does not employ hardware counters, nor does it focus on the difficulties in making accurate measurements or interpretations of those measurements. Roofline does not consider the on-chip memory hierarchy or the complications of DRAM pages and access patterns along with their performance impact on high-level compiler optimizations.

6 Summary and Conclusion

This paper details an experimental study of intrachip/intranode scalability and the factors that determine this scalability for HPC applications running on systems with multicore chips and multichip nodes. Our study focuses on a deep analysis of HOMME, an at-scale application used for production climate modeling.

Our results demonstrate that effective measurement, analysis, and optimization of memory performance bottlenecks intrinsic to multicore nodes require a different and more complex approach than memory bottleneck detection and alleviation in unicore nodes. We show that multicore systems exhibit qualitative and quantitative differences in performance bottlenecks and metrics, in experimental and measurement issues, and in optimization strategies.

For example, we show that accurate memory performance measurements in multicore environments must account for the delayed effects of memory references and the nondeterministic interactions among the cores on a chip and/or node, which are immaterial or absent in unicore nodes. We further show the temporal context of the measurements to be critical, and obtaining sufficient measurement accuracy may require using multiple tools. We demonstrate a range of measurement techniques to overcome these complications. We also describe a structured process for effectively measuring the performance metrics critical to multicore chip and multichip node performance, including methods for interpreting these metrics to obtain an accurate definition of the causes of multicore-related memory performance bottlenecks.

Using this process, we have identified three key multicore memory performance bottlenecks: shared L3 cache capacity, shared off-chip bandwidth, and DRAM page conflicts. Driven by these bottlenecks, we developed a source-code loop-level optimization called microfission that, when applied to HOMME,

reduces the L3 cache miss rate by almost 50%, more than doubles the DRAM page hit rate, reduces compiler overhead instructions by a third, and increases intrachip scalability by up to 42% and absolute performance by up to 35%. We anticipate that microfission will be applicable to a wide range of multicore applications and that our insights into multicore bottlenecks will inspire additional optimizations specifically aimed at multicore execution.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful feedback. We would also like to thank Lars Koesterke for his expert help with Fortran95 issues as well as his experience with HPC optimizations, and Victor Eijkhout for his experiences with sparse and unstructured HPC applications. This work was supported in part by the National Science Foundation under award CCF-0916745 and OCI award 0622780.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, April 2010.

[2] BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, Rev 3.48. `http://developer.amd.com/documentation/guides/pages/default.aspx`, April 2010.

[3] D. F. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. In *ACM Computing Surveys*, volume 26, pages 345–420, December 1994.

[4] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC-10)*, November 2010.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC-08)*, November 2008.

[6] J. Diamond, B.-D. Kim, M. Burtscher, S. Keckler, K. Pingali, and J. Browne. Multicore Optimization for Ranger. In *2009 TeraGrid Conference*, June 2009.

[7] J. Diamond, J. D. McCalpin, M. Burtscher, B.-D. Kim, S. W. Keckler, and J. C. Browne. Making Sense of Performance Counter Measurements on Supercomputing Applications. Technical Report TR-10-25, University of Texas at Austin, Department of Computer Science, 2010.

[8] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The Impact of Multicore on Computational Science Software. *CTWatch Quarterly*, 3:3–10, February 2007.

[9] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 120–126, June 1982.

[10] J. Hack, B. Boville, B. Briegleb, J. Kiehl, P. Rasch, and D. Williamson. Description of the NCAR Community Climate Model (CCM2). Technical Report TN-382, NCAR, Boulder, Colorado, 1993.

[11] Intel 64 and IA-32 Software Developer's Manual, Volume 3B: System Programming Guide, Part 2. `http://www.intel.com/assets/pdf/manual/253669.pdf`, January 2011.

[12] R. Jayaseelan, A. Bhowmik, and R. D. C. Ju. Investigating the Impact of Code Generation on Performance Characteristics of Integer Programs. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architecture (INTERACT)*, pages 1–8, March 2010.

[13] Longhorn User's Guide. `http://services.tacc.utexas.edu/index.php/longhorn-user-guide`.

[14] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, F. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[15] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `http://www.cs.virginia.edu/stream/`, 1991-2010.

[16] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[17] MPI Profiler. `http://mpip.sourceforge.net/`.

[18] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. We Have It Easy, But Do We Have It Right? In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7, April 2008.

[19] NSF 0605: The High-Performance Computing Challenge Benchmarks, version 2.0. `http://www.nsf.gov/publications/pub_summ.jsp?ods_key=nsf0605`, November 2005.

[20] L. Oliker, J. Shalf, and K. Yelick. Optimization of a Lattice Boltzmann Computation on State-of-the-art Multicore Platforms. *Journal of Parallel and Distributed Computing*, 69(9):762–777, September 2009.

[21] PAPI: Performance Application Programming Interface. `http://icl.cs.utk.edu/papi`.

[22] Linux Performance Counter Kernel API. `http://user.it.uu.se/~mikpe/linux/perfctr/`.

[23] Ranger User's Guide. `http://www.tacc.utexas.edu/services/userguides/ranger`.

[24] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[25] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. *Parallel Computing*, 35(3):178–194, March 2009.

[26] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, April 2009.