View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UPCommons. Portal del coneixement obert de la UPC

Active Measurement of Memory Resource Consumption

Marc Casas
Barcelona Supercomputing Center
Jordi Girona 29, Nexus II Building
08034 Barcelona

Greg Bronevetsky
Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA, 94550

***Abstract*—**

Hierarchical memory is a cornerstone of modern hardware design because it provides high memory performance and capacity at a low cost. However, the use of multiple levels of memory and complex cache management policies makes it very difficult to optimize the performance of applications running on hierarchical memories. As the number of compute cores per chip continues to rise faster than the total amount of available memory, applications will become increasingly starved for memory storage capacity and bandwidth, making the problem of performance optimization even more critical.

We propose a new methodology for measuring and modeling the performance of hierarchical memories in terms of the application's utilization of the key memory resources: capacity of a given memory level and bandwidth between two levels. This is done by actively interfering with the application's use of these resources. The application's sensitivity to reduced resource availability is measured by observing the effect of interference on application performance. The resulting resource-oriented model of performance both greatly simplifies application performance analysis and makes it possible to predict an application's performance when running with various resource constraints. This is useful to predict performance for future memory-constrained architectures.

I. Introduction

Hierarchical memory (registers, caches and main memory) is a critical driver of modern systems' high performance because it combines small amounts of fast but expensive memory and large amounts of slower, cheaper memory to provide an excellent balance of low cost, high performance and high capacity. However, its complexity makes it very difficult to achieve high performance and energy efficiency for real applications, a problem that has motivated significant research on cache-friendly algorithms [8], [5] and performance analysis tools to simplify this task [19], [16], [12]. Unfortunately, even after decades of work, the goal of easy-to-use memory optimization techniques is still far from reach.

Modern architectural designs provide increasing improvements in computation capability while maintaining a constant power utilization by increasing the number of cores on each chip. Since the power-efficiency and cost-efficiency memory designs are not improving at the same rate, the amount of memory per compute core is dropping [13]. This is especially true for High Performance Computing (HPC) systems, where hard limits on power costs will mean that next-generation Exascale systems may provide one or two orders of magnitude less memory capacity and bandwidth per core than today's systems [13]. These limitations will force application designers to fundamentally rethink how their algorithms utilize the memory system and will make effective memory optimization methodologies critical for maintaining application performance on future systems.

Ensuring that applications use the memory hierarchy optimally or restructuring algorithms to leverage hierarchies that are deeper (more levels) and thinner (fewer resources per core) requires a detailed analysis of how an application uses memory. Although there exists a wide range of tools to help with this task, they have key limitations. Simulation-based tools such as cachegrind [17] and gem5 [3] can analyze the application's behavior in great detail and can predict the performance of any collection of applications running on any hardware configuration. However, such tools run hundreds or thousands times slower than native execution and cannot simulate the commercial architectures on which almost all applications run because simulator developers have no access to their proprietary details. These limitations have motivated work on tools based on monitoring hardware performance counters. These tools report metrics such cache miss rates or instructions per cycle for various code regions [19], conduct complex statistical analyses of such counter data [12] or connect them to other aspects of the application, such as data structures [16]. Although these tools are efficient and precisely capture the state of the hardware and how it is utilized by the application, this information is not actionable in most cases. First, the metrics reported by these tools are so low-level that they can only be interpreted by the most hardware-savvy developers. Further, this information is not useful for predicting how the application may behave in alternate scenarios, such as

The research leading to these results has received funding from the European Research Council under the European Union's 7th FP (FP/2007-2013) / ERC GA n. 321253. Work partially supported by the Spanish Ministry of Science and Innovation (TIN2012-34557). This article has been authored in part by Lawrence Livermore National Security, LLC under Contract DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. This work was partially supported by the Department of Energy Office of Science (Advanced Scientific Computing Research) Early Career Grant, award number NA27344.

if its available resources are reduced by the execution of other software or because the application runs on a new platform. The limitations of today's techniques motivate the need for a new approach that combines the predictive capability of simulation-based tools with the high performance of counter-based methods for real commercial proprietary hardware.

This paper presents a new performance analysis technique that addresses this need by capturing the application's effective use of the storage capacity of different levels of the memory hierarchy as well as the bandwidth between adjacent levels. Our approach models various memory components as resources and measures how much of each resource the application uses *from the application's own perspective*. To the application a given amount of a resource is "used" if not having this amount will degrade the application's performance. This is in contrast to the hardware-centric perspective that considers "use" as any hardware action that utilizes the resource, even if it has no effect on performance. For instance, while from the hardware perspective cache storage capacity is "used" when live data is stored in it, to the application it is "used" only if the data is part of the application's active working set. This paper specifically focuses on measuring storage capacity in caches that are shared by multiple cores and the bandwidth between them and higher levels of the memory hierarchy. In addition to measuring use, we also quantify the application's sensitivity to being provided less of the resource than it needs to run without suffering performance degradations. This predicts how well the application would run in scenarios where less of the resource is provided, such as the memory hierarchy of a future system (e.g. a node of an Exascale system).

We measure the application's use of memory resources via the proactive methodology illustrated in Figure 1. While the application runs, it uses a given fraction of each resource at a given level of a memory hierarchy (denoted "level X cache"). If this level X cache is shared among multiple cores it is possible to measure this use by running on another core an interference thread that utilizes a known amount of cache capacity or bandwidth, where the use of a separate core limits the thread's effects to just the chosen resource. The algorithm increases the amount of resource used until the main application's performance is observed to degrade. The difference between the total amount of the resource and the amount used by the interference thread at that point is the amount actively used by the application. If the application's performance is not sensitive to the interference then it either primarily uses a higher level of the memory hierarchy (little use of level X) or doesn't fit in level X and is thus not sensitive to reductions in its capabilities. The two cases can be differentiated by observing the application's miss rates for level X cache.

Overall, this paper's contributions are:

- A methodology to actively measure the application's resource use in terms of the effect of availability of this resource on its performance.
- A novel interference-based mechanism to simulate a reduction in available memory storage and bandwidth on a given real hardware architecture.
- A validation technique for quantifying the real effects of our interference mechanisms on the application and bounding the effects of a given source on interference on unrelated resources.
- A method to predict how the application's performance will degrade on alternative, less capable memory hierarchies.

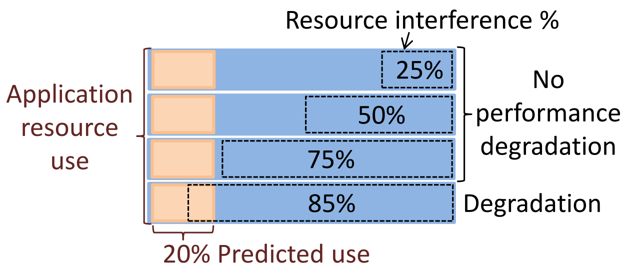

Fig. 1. The application's resource use is measured by interfering with increasing fractions of the resource until this affects application performance

Section II presents our interference measurement methodology. This approach is validated in Section III using microbenchmarks with well-characterized memory behavior. Section IV demonstrates how our measurements can be applied to real parallel applications by measuring the cache storage and bandwidth requirements of MCB [2] and Lulesh [1]. Our evaluation quantifies the performance degradation that results from providing applications with different amounts of memory resources, enabling more intelligent work scheduling and architecture design planning.

II. Measurement Methodology

Each level of the memory hierarchy provides two resources: storage capacity and communication bandwidth to the (larger and slower) level below it. Our Active Measurement methodology measures the application's use of these resources by comparing its performance when running on a given memory system to its performance when less of a given memory system resource is available due to the execution of a special interference thread. Specifically, the techniques presented in this paper focus on portions of the memory hierarchy that are shared among multiple cores. This restriction enables more precise measurements by making it easier to isolate the effects of the interference thread to just the specific resource it is designed to target because the interference threads run on different cores that share the resource. This section details the design of these threads and Section III experimentally validates that each thread is effective at using up the resource that it targets and uses few other resources.

Our experiments focus on the following architecture: 2-socket nodes with 8-core Intel Xeon E5-2670 processors. The L1 and L2 caches are private to each core, while the L3 cache is shared among all the cores on a socket. The L3 on the Intel Xeon E5-2670 is 20MB in size and this architecture is thus denoted Xeon20MB (details in Table I).

	Cache	Capacity	Line Size	Associativity
Private	L1 I	32KB	64 bytes	8-way
	L1 D	32KB	64 bytes	8-way
	L2	256KB	64 bytes	8-way
Shared	L3	20MB	64 bytes	20-way

TABLE I
8-CORE INTEL XEON E5-2670 MEMORY HIERARCHY

```
long long int* buf_0 =
     malloc(sizeof(long long int)*bufSize);
...
long long int* buf_numBufs =
     malloc(sizeof(long long int)*bufSize);

for(int i=0; 1; i++) {
    buf_0[identity(largePrime*i)%bufSize]++;
    ...
    buf_numBufs[identity(largePrime*i)%bufSize]++;

}
```

Fig. 2. Pseudo-code of the bandwidth interference thread BWThr

A. Memory Bandwidth Interference

The pseudo-code of the bandwidth interference thread, denoted BWThr is shown in Figure 2. This code attempts to transfer as much data as possible between one memory hierarchy level and the next by issuing a large number of memory accesses that will miss in the first level. We induce frequent memory misses by allocating a buffer and iterating it with a stride that is a large prime number. The use of a prime keeps the number of iterations between adjacent accesses to the same location large and the constant stride makes it possible for the hardware prefetcher to help use up more bandwidth. Further, to ensure that the compiler cannot perform any optimizations based on the simple access pattern, the computation of the strided index `largePrime*i` is wrapped inside a call to an `identity` function that is located in a different file and thus not available at compile time. One side-effect of this is that the compiler can no longer transform the loop to issue the maximum number of simultaneous memory accesses that the underlying hardware can support. We overcome this problem by simultaneously performing this procedure for many buffers at the same time (our experiments use 44, which we discovered to be sufficient), maximizing the concurrent memory traffic.

B. Cache Storage Interference

Figure 3 shows the pseudo-code of the storage interference thread, denoted CSThr. It allocates a buffer of a given size and then randomly touches elements in this buffer. If the buffer fits inside the high-level private caches of CSThr's processor this iteration has little effect on the cache shared by it and the application. This is because once the data is fetched into this cache during the initial iterations of the inner loop, there will be no additional cache misses. However, once the array grows larger than the private caches, the random order of the memory accesses ensures that many of the buffer accesses will miss in the private cache and will always hit in the lower-level shared cache. A random memory access pattern ensures a more intense perturbation than linear access patterns because the probability of consecutively accessing two addresses of the same cache line is very low, meaning that almost every access misses in the L1 and L2 and hits in the L3. Further, the use of random access ensures that the hardware pre-fetcher will not recognize the access pattern and thus will not fetch in additional addresses outside the target buffer. Because CSThr spends all of its time passing over the buffer, the application threads have little chance to use the cache space assigned to the buffer before CSThr accesses this line again and pulls this resource away from the application. The design of CSThr ensures that it predictably utilizes a fixed fraction of the target shared cache and prevents the application from making any productive use of it.

```
int* buf = malloc(sizeof(int)*bufSize);
while(1) buf[random_position]++;
```

Fig. 3. Pseudo-code of the storage interference thread CSThr

III. VALIDATION

In this section we evaluate the amount of storage and bandwidth resources the interference threads utilize and validate our results. We also demonstrate that each interference thread only utilizes its target resource, meaning that they affect application behavior orthogonally.

A. Memory Bandwidth Interference

The bandwidth used by BWThr is computed based on the number of L3 cache misses it incurs, which is obtained by reading the hardware performance counter available in the Xeon20MB machine. Since each miss causes a full cache line to be transferred from main memory to the L3 of its core, its bandwidth use is computed as:

$$BW = \frac{cache\_line\_size \cdot \#cache\_misses}{ExecutionTime} \quad (1)$$

This is simply the total amount of memory transferred between the given memory hierarchy level and the next, divided by the application's execution time.

Our measurements indicate that using a 520KB buffer a single BWThr utilizes 2.8GB/s per core in Xeon20MB. Since Xeon20MB provides 17GB/s of bandwidth between the L3 cache and memory according to the STREAM benchmark [15], 7 BWThr running on 7 different cores would consume approximately 100% of the available bandwidth.

B. Cache Storage Interference

The CSThr utilizes cache storage by repeatedly accessing a range of memory addresses, attempting to deny the application their use. However, because there is a time window between adjacent touches by CSThr of the same cache location it is possible for the application to bring its own data into this location and make productive use of it before it is evicted by CSThr. As such, the total amount of storage utilized by CSThr cannot be computed directly and must be computed based on its effects on representative applications. We evaluate this in two ways. First, as described in Section III-C we create several synthetic benchmarks with different well-known

```
int* buf = malloc(sizeof(int)*bufSize);
for (int i=0; i<N_ACCESSES; i++) {
        int value = buf[X()];
        // Some computation involving value
}
```

Fig. 4. Probabilistic Memory Access Algorithm. $X()$ is a random variable that has a probability distribution function f associated.

memory access patterns based on probability distributions. We then use the knowledge of a each benchmark's distribution to derive the L3 cache miss rate that the benchmark would observe when running on a fully associative cache of a given size. Given the L3 miss rate observed when each benchmark runs concurrently with CSThr we can then compute the effective cache size that is available to the benchmark at a given level of CSThr interference. Our experiments show that in most cases CSThr has a consistent effect for all the benchmarks and identify the narrow range of applications and interference levels for which the effects of CSThr cannot be accurately quantified.

C. Validation Based on Synthetic Benchmarks

1) Benchmark Design: Figure 4 shows the skeleton of the synthetic benchmarks we use to validate CSThr. It loops over a buffer N_ACCESS times and in each iteration reads the value at a buffer index chosen randomly from some probability distribution and performs some number of computations on this location. Every time that a new index is generated, an access to the memory must be performed to get the data. In our experiments we created several different variants of this benchmark for a range of probability distributions and different degrees of memory access patterns. Table II lists all the distributions considered and represents both a wide range of access patterns as well as degrees of spatial locality (depends on the standard deviation, which is varied widely). Since in the computation in 4 $value$ is in a register, average time between two memory accesses is varied by setting the computation performed after each read to be 1, 10 or 100 integer additions. By doing that, we consider not only the memory access pattern itself, but also several degrees of memory access frequency.

The first step in our analysis is to derive the expected L3 cache miss rate for each benchmark given the amount of cache storage that is available. This is computed via the following formula, which calculates the probability that a given randomly sampled index is in the cache and uses it to compute the Expected Hit Rate (EHR):

$$EHR = \sum_{i \,\in\, \text{buffer}} P(i \text{ is accessed}) \cdot P(i \in \text{cache}) \qquad (2)$$

$P(i$ is accessed$)$ is equal to the probability mass function $f(i)$ of the distribution. The probability of i being in the cache, if the cache capacity is smaller than the buffer size, is equal to $\frac{Cache\_capacity}{Buffer\_size}$, multiplied by the factor $f(i) \cdot Buffer\_size$. The first rate comes from just applying the classic definition of probability of number of favorable outcomes divided by the total number of possible outcomes. The second factor is the

Pattern Name	Statistical Distribution	Distribution Parameters	Standard Deviation
Norm 4	Normal	μ=n/2 σ=n/4	n/4
Norm 6	Normal	μ=n/2 σ=n/6	n/6
Norm 8	Normal	μ=n/2 σ=n/8	n/8
Exp 4	Exponential	λ=4/n	n/4
Exp 6	Exponential	λ=6/n	n/6
Exp 8	Exponential	λ=8/n	n/8
Tri 1	Triangular	a=0 b=0.4n c=n	$n^2/18$
Tri 2	Triangular	a=0 b=0.6n c=n	$n^2/18$
Tri 3	Triangular	a=0 b=0.8n c=n	$n^2/18$
Uni	Uniform	a=0 b=n	$n^2/12$

TABLE II
MEMORY ACCESS PATTERNS CONSIDERED. n IS THE SIZE OF THE BUFFER.

probability that one of the last $Buffer\_size$ accesses is to index i, depending on the mass function f. Thus we have:

$$EHR = \sum_{i \,\in\, \text{the buffer}} f(i) \cdot \frac{Cache\_capacity}{Buffer\_size} \cdot \frac{f(i)}{\frac{1}{Buffer\_size}} \qquad (3)$$

$$= Cache\_capacity \cdot \sum_{i \,\in\, \text{buffer}} f(i)^2 \qquad (4)$$

This formula makes three assumptions. The probability of accessing any buffer element must be non-zero and the size of the buffer must be larger than the size of the cache. These assumptions are satisfied by the distributions in Table II and we run the benchmarks with sufficiently large buffers. Finally, the formula applies to steady state execution, after the algorithm has warmed up the cache by loading its contents based on its probability distribution. We satisfy this assumption by setting N_ACCESS to be much larger than the buffer sizes.

One limitation of this model is that it assumes that the cache is fully associative, which is not true in general. As such, despite the fact that the model is in general accurate, as it is demonstrated in section III-C2, it slightly underestimates the number of cache misses because set-associative miss ratios are in general larger than fully-associative ones.

The dependence of the cache miss rate on cache size has been studied for generic applications in the past by using statistical models empirically derived [9]. Our model offers more insight, as it is not empirical, but it is restricted to a particular set of memory access patterns.

2) Validation of Probabilistic Model: The variety of distributions used in our synthetic benchmarks induce a wide range of memory access patterns and L3 cache miss rates from below 10% to above 80%. The probability distributions with larger standard deviations access with a higher probability wider sets of data, which decreases the chances of accessing two elements of the same cache line in a short period of time. This fact induces higher miss rates due to worse spatial memory locality. Further, cache miss rates rise as the buffer size increases since more memory is available for selection.

Equation 4 computes the benchmark's miss rate based on its available storage capacity. We validated the equation by running the synthetic benchmarks on Xeon20MB and

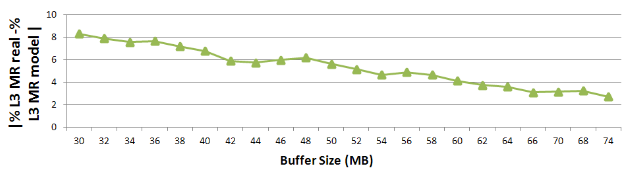

Fig. 5. Model Evaluation.

compared the miss rates it predicts based on the 20MB of L3 cache known to be available with the real cache miss rates measured by hardware counters. Figure 5 shows the absolute differences between these two numbers, averaged over all the distributions in Table II. The average absolute distance between the measured cache miss rates and the predicted ones is always less than 10% and the average plus one standard deviation is 15% or less. Error is higher for small buffer sizes because the model assumes that caches are fully associative. This under-predicts the real cache miss rate when the cache is not heavily used and there are few cache misses. However, as the cache becomes fully utilized by larger buffer sizes and most memory accesses become misses (above 50% miss rate), the details of cache associativity become unimportant and the model's error drops to under 5%. In summary, the data shows that the simple analytic model of the behavior of the synthetic benchmarks is accurate in general, especially for large buffer sizes. Exhaustive studies on the relationship between cache associativity and miss rate have been done [10] and are consistent with the accuracy of our analytical model.

3) Measuring Cache Storage Use: Having shown that Equation 4 accurately predicts the L3 miss rate of the synthetic benchmarks based on the available cache storage capacity we can now use it to predict the effective storage available to these benchmarks when CSThr interferes with a portion of it. This is done by running experiments where CSThr interferes with a given synthetic benchmark's use of cache storage, and measuring the resulting L3 miss rate. We then invert the formula in Equation 4 and given the observed miss rate compute the effective amount of available cache storage.

We conducted this evaluation on the Xeon20MB architecture, with 0 to 5 CSThrs, each using 4MB buffers. The synthetic benchmarks were parameterized with 10 different probability distributions (Table II), 3 different degrees of memory access frequency (1, 10 and 100 integer additions per load) and 22 different buffer sizes from 30MB to 74MB for a total of 660 different configurations. The L3 cache miss rates of these variants range from less than 10% to more than 80%, with a similar variation on L3 cache memory access frequencies. This breadth of coverage makes our validation representative of a wide range of real applications, as it is pointed out in [20], where it is shown that L3 miss rates in SPEC 2006 applications range from 20% to 100% in commodity hardware, which is equivalent to the range covered by our 660 benchmark configurations.

The goal of this experiment is to determine whether the effects of different numbers of CSThrs are consistent across this wide range of memory access patterns. Figure 6 shows the results of our experiment. The charts from top to bottom show results with different degrees of memory access frequency, least frequency on top and most on the bottom. The charts from left to right show results with different numbers of CSThrs, with no interference on the left and then 1 through 5 CSThrs on the right.

Each chart shows on the y axis the amount of cache storage that is available to the benchmarks, as computed by our formula and the x-axis shows the size of the buffer used by the benchmark (concrete buffer sizes omitted to improve readability and are the same as in Figure 5). In addition to the average predicted storage (across all the 10 probability distributions), denoted by the thick horizontal line, the charts show the region that includes the average plus and minus the standard deviation. The length of these intervals represents the dispersion of the measurements across all the distributions and is smaller for more consistent predictions of cache storage capacity. As such, the lower the length is, the more consistent is the estimation of the equivalent cache capacity.

The data displayed in the column tagged as "No Interference" shows results computed without any CSThr. As such, the predicted cache capacity should match the real cache capacity of the machine, and they do specially when buffer sizes close to 74MB are used. The buffer sizes close to 30MB show lower values because, as explained in Section III-C1, the model under-estimates the cache miss rates due to its assumption of fully associative caches. Thus, given the higher miss rates measured in real experiments, the inverse of Equation 4 under-estimates the available amount of cache storage. The error of the predictions drops as the buffer size is increased, as seen in Figure 5, until the predictions reach the correct capacity of 20MB.

The outcome of the experiments with no interference is the same for all degrees of memory access frequency (1, 10 or 100 operations between loads) since in these experiments the benchmarks do not compete for the cache. When 1 CSThr is used, the model predicts an effective cache capacity of approximately 15MB for all degrees of memory access frequency and 12MB with 2 CSThrs. When we use 3, 4, or 5 CSThrs the effective cache capacity is approximately 7, 5, and 2.5MB respectively.

One interesting phenomenon is that as the frequency of memory accesses rises so does the standard deviation of the predictions. Although this effect is weak for 1 CSThr, it becomes stronger as more CSThrs are added. Our experiments show that for low access frequencies the effects of CSThrs are erratic to a point where with 5 CSThrs interfere with either as much as the entire cache or as little as just half of it.

Overall, this validation quantifies the accuracy of the validation methodology and identifies the properties of applications for which it has high error: high degrees of interference and memory access frequency. Fortunately 100 arithmetic operations per load represent a very low memory access frequency. In particular, in scientific applications, which are typically more regular than most other application domains, 1-10 operations per load is considered the most common range.

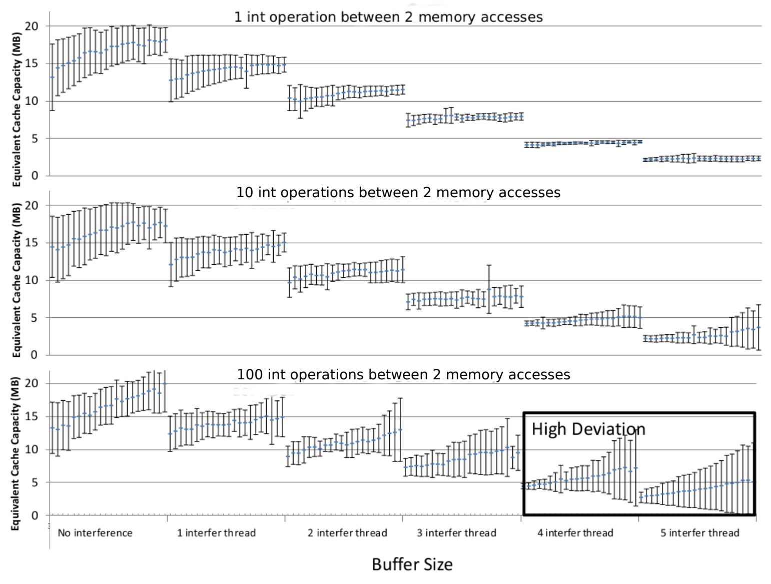

Fig. 6. Evaluation of the cache capacity interference. In the top, the benchmarks compute 1 integer sum between two consecutive memory accesses. In the middle, they perform 10 integer sums and at the bottom 100 integer sums are computed between memory accesses.

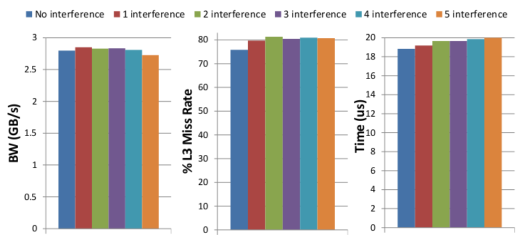

Fig. 7. Memory Bandwidth use, L3 cache miss rate and time to do 10^7 iterations over its main loop of the BWThr when running concurrently with between 0 and 5 CSThrs.

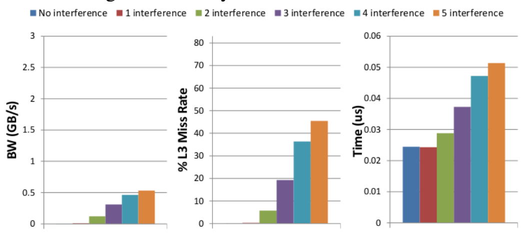

Fig. 8. Memory Bandwidth use, L3 cache miss rate and time to perform a read, an arithmetic addition and a write of the CSThrs when running concurrently with between 0 and 5 BWThr.

Our experiments thus show that our validation methodology is accurate for most real-world workloads.

D. Orthogonality of Cache Storage and Bandwidth Interference

Sections III-A and III-B have demonstrated that BWThrs and CSThrs consistently utilize a given amount of cache bandwidth and storage, respectively. However, since both techniques use the memory hierarchy there is a risk that they utilize additional resources besides the ones they target. If so, they would interfere with the application in multiple ways, making each measurement reflect the use of a complex combination of system resources that has little intuitive meaning to the application developer. It is thus necessary to establish that these threads have orthogonal effects: that each thread type almost exclusively affects the resource it targets and no other. By running the interference threads on separate cores we have ensured that no resources private to the application's cores are used. In this section we quantify the degree to which BWThrs utilize cache storage and CSThrs utilize cache bandwidth.

To measure this we executed the BWThrs (520KB buffers) and CSThrs (4MB buffers) simultaneously on different cores of the same Xeon20MB socket to measure each others' resource use. Figure 7 shows the effect on the execution of a single BWThr of concurrently running between 0 and 5 CSThrs, measured in terms of (i) the amount of memory bandwidth effectively used by the BWThr, (ii) the measured L3 cache miss rate of the BWThr, and (iii) the total time required to iterate 10^7 times over the BWThr's main loop, shown in Figure 2. The data shows that the BWThr behaves the same regardless of the number of CSThrs that are running concurrently with it. That implies that we can run up to 5 CSThrs without significantly impacting the memory bandwidth.

Figure 8 shows the opposite experiment: 1 CSThr running concurrently with between 0 and 5 BWThrs. The bandwidth plot shows the bandwidth consumed by the CSThr and execution time plot shows the average time the CSThr takes to perform a read, an arithmetic addition and a write operation. The data shows that a single BWThr has no impact on the CSThr's performance and 2 BWThrs have a small effect. However, the CSThr is impacted significantly by the execution of 3, 4 and 5 BWThrs, which implies that 3 or more BWThrs utilize significant amounts of cache capacity. This induces L3 misses in the CSThr, which cause it to slow down and use more bandwidth. As discussed in Section II-A, each BWThr utilizes 2.8GB/s of bandwidth, which means that up to 5.6GB/s can be stolen without impacting cache capacity. Since the total memory bandwidth measured in the Xeon20MB architecture is near 17GB/s, that means we can impact on 32% of the total, a significant portion of it, and keep the independence of the interference threads. Another important measurement provided in the left hand side of figure 8 is the memory bandwidth utilization of the CSThr. A single CSThr without additional interference utilizes very little memory bandwidth, which fully validates the orthogonality of CSThr and BWThr.

These results demonstrate that our measurements are indeed highly focused for most of their dynamic range. Further, because their effects are orthogonal they can be used to represent the application's overall memory behavior as a simple 2-dimensional linear space where BWThr and CSThr identify the basis vectors: the application's utilization of storage and bandwidth. This projection enables application developers and architects to reason about an application's memory use in the same terms as if they were using a cache simulator without paying the big burden in terms of computing time that architectural simulation has.

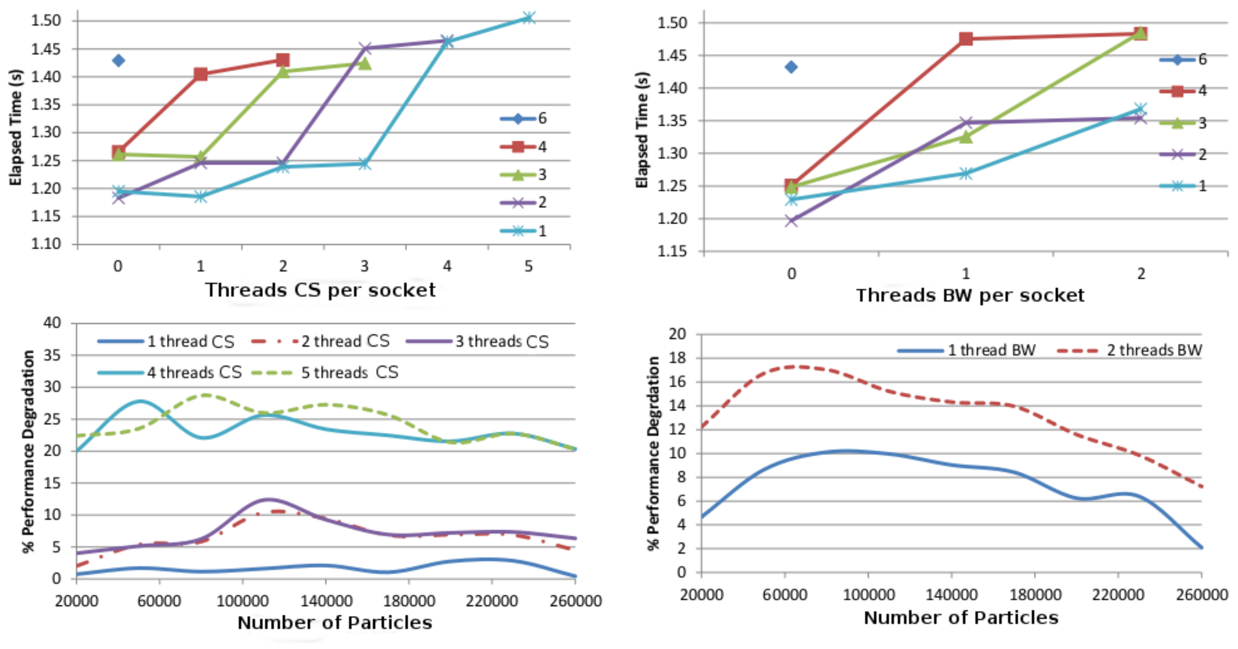

Fig. 9. Performance Degradations of MCB on 24 MPI tasks. The two top figures show results obtained considering several MPI mappings and using a 20,000 particles domain. The two figures at the bottom show results obtained when we map 1 MPI tasks per processor. Numbers of particles between 20,000 and 260,000 are considered.

IV. Parallel Application Studies

Having presented and validated our basic measurement methodology we now demonstrate how it can be applied to parallel applications to gain insight into how they utilize the memory hierarchy. Our evaluation focuses on the Lulesh and MCB scientific benchmark codes. Lulesh solves a Shock Hydrodynamics Challenge Problem that simulates large deformations in materials using a finite differences scheme. MCB simulates the fuel assemblies in a nuclear reactor by simulating the flow of neutrons through it using the Monte Carlo method. We consider the execution of these applications on a wide range of input sizes on a cluster of nodes with the Xeon20MB architecture, each of which has 32GB of RAM and are connected via InfiniBand QDR (QLogic) interconnect (40Gb/sec bandwidth). Our experiments focus on two aspects of these applications' performance:

- Measuring the amount of L3 storage and L3$\leftrightarrow$Main Memory bandwidth used by these applications, and
- Characterizing their sensitivity to being provided less of either of these resources.

Our measurements are performed by running the benchmarks on one or more nodes, allocating some cores to the application and executing BWThrs or CSThrs on some or all of the remaining cores. In bandwidth experiments we run 1 or 2 BWThrs with a buffer size of 520KB each to interfere with up to 32% of the total memory bandwidth. In storage experiments we run between 1 and 5 CSThrs with a buffer size of 4MB each to utilize upto 87% of the total cache capacity (17.5MB out of 20MB). In our experiments, MCB is run with 24 MPI processes and Lulesh with 64 MPI processes.

In each experiment we increased the amount of storage or bandwidth utilized by the BWThrs and CSThrs to observe both the point where the application's execution time degraded (indicated the amount of resource used by the application) as well as the relationship between resource availability and performance degradation. The degree to which the performance of a parallel application degrades is a combination of two major effects. First, when less memory bandwidth is available cache misses take longer to complete because less bandwidth is available to transfer the data. Further, when available memory storage is reduced, cache accesses that are normally hits become misses, which may also take longer than normal since the increased number of misses may saturate available memory bandwidth. These phenomena cause each individual application process to slow down. Importantly, the slowdown of each process is stochastic, with individual instructions on different processors affected very differently by interference. This non-deterministic slowdown of instructions introduces noise into the application's execution, which is a well-known source of slowdown for parallel applications [18], [11].

The top graphs of Figure 9 show in detail the performance degradations we have measured for several different mappings of the processes of MCB to compute nodes when the input set size is 20,000 particles. They show separate curves for cases where $p = 1$ through 6 processes are mapped to each Xeon20MB processor and 8-p cores are available on each processor (each processor has 8 cores). Since MCB uses a total of 24 processes and each node has 2 processors, when p processes run on one processor the overall application uses $24/(2*p)$ nodes. The x-axis corresponds to different numbers of CSThrs or BWThrs running on the available cores. The y-axis shows MCB's execution time in this mapping and interference level. Note that since different mappings leave different numbers of cores available, not all combinations of mapping and interference can be executed. The bottom graphs show the performance degradations measured when running MCB on problems with 20,000 to 260,000 particles. They correspond to MCB runs on 24 Xeon20MB processors (12 nodes), with 1 MCB process and 7 available cores per processor. The graphs on the left focus on storage interference and the graphs on the right present bandwidth interference results.

The top-left graph of Figure 9 shows MCB's performance degradation across different process mappings. We can see the consistency of the performance degradations across all the considered MPI mappings: the more processes mapped on each processor, the less cache capacity is available for each process and thus the same performance degradation is induced with fewer CSThrs. This representation suggests a simple way to calculate the average cache capacity utilization of MCB processes. For each process mapping, we consider the experiments with no performance degradation and pick the one that has the most CSThrs. We then consider the experiments with performance degradation and pick the one with the fewest CSThrs. Our prior analysis has determined that 1, 2, 3, 4, and 5 CSThrs with a 4MB buffer size leave 15, 12, 7, 4 and 3MB of cache capacity available to the application. We use this information to compute for each of the above configurations the ratio $\frac{Available\_cache\_capacity}{\#processes}$ to obtain the upper and lower bound on the amount of storage available to each application process. For MCB running on 20,000 particles (top-left graph of Figure 9) each MCB process needs between 3.75 and 5MB of L3 capacity when 4 processes are mapped on each processor, 3.5-6MB are required when 2 run on each processor and 4-7MB for 1 process per processor. Performing the same analysis for the memory bandwidth (from before, we have 17GB/s with no interference, 14.2GB/s with 1 BWThr and 11.4 with 2 BWThrs), we calculate that each MCB process' bandwidth utilization is 3.5-4.25GB/s, 3.80-4.7GB/s, 7.1-8.5GB/s and 11.4-14.2GB/s when we map 4, 3, 2 and 1 processes per processor.

Figure 10 shows these results more visually, clarifying the trends. These measurements confirm the intuition that mapping one process per processor consumes more memory bandwidth than other mappings because all the communications go through the memory bus. As more processes share the same L3 more communication can go through the L3 without incurring misses. In contrast, storage use does not change significantly as processes are spread out. The results are similar for other input sizes.

The bottom-left graph of Figure 9 shows MCB's performance degradation as 1 through 5 CSThrs are executed on the available cores, which is equivalent to an L3 cache size of 15MB, 12MB, 7MB, 4MB and 3MB, respectively, according to the measurements described in Section III-C3. The data shows that when MCB simulates 20,000 to 260,000 particles, there is little performance degradation with one, two or three

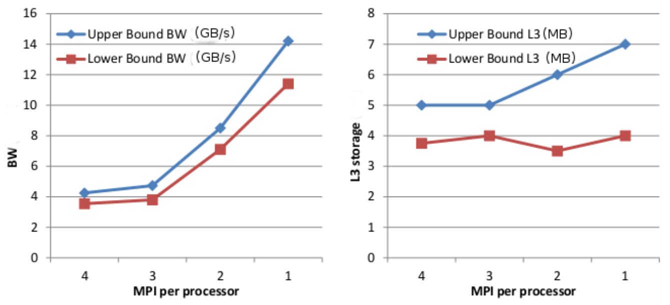

Fig. 10. MCB resource consumption depending on the MPI mapping. Number of particles is 20,000.

CSThrs and significant degradation of 20-25% with four or five CSThrs. This means that on this input range each MCB process uses between 4MB and 7MB of the L3 cache, the same behavior as we have shown in figure 10. The fact that even when given 2.5MB MCB's performance only suffers by less than 30% indicates that this application would not perform much worse even if the L3 cache was not available on this architecture.

The bottom-right graph of Figure 9 shows MCB's performance degradation when one or two BWThrs are executed to reduce the available bandwidth (as discussed above, running more than 2 BWThrs also uses up cache storage). A single BWThr reduces the available bandwidth by 16% of the total memory bandwidth (17GB/s) while two reduce it by 32%. The impact on MCB's performance grows as the number of particles increases from 20,000 to 90,000 because its communication and thus miss rate grows with increasing workloads. Above 90,000 particles the impact of bandwidth interference drops because the application spends more time computing and less time communicating, which reduces the pressure on the memory buses.

Figure 11 shows the performance degradation of Lulesh [1] as it runs with 64 MPI processes. The physical domains simulated by Lulesh are cubes of sizes from 22x22x22 to 36x36x36 units (the size of one dimension is reported on the x axis), using the same format as Figure 9. The graphs in the top show performance degradations we measure running Lulesh on 22x22x22 cube across different process mappings. Experiments with 4 processes per processor show that Lulesh overflows the L3 cache when any number of CSThrs run, meaning that for all inputs each Lulesh process uses more than 3.5MB of storage (15MB that one CSThr leaves available, divided by 4 Lulesh processes per processor).

The bottom graphs of figure 11 show runs on 64 Xeon20MB processors (32 nodes) and 1 process per processor. The bottom-left graph shows the performance degradation of Lulesh with increasing number of CSThrs for 1 process per processor. When domain size is 32x32x32 or smaller degradation is less than 5% for 1 and 2 CSThrs but more than 10% for 5 CSThrs, indicating that each Lulesh process uses between 2.5MB and 10MB of cache storage. For larger cubes Lulesh overflows the L3 cache with any amount of storage interference, suggesting that for these input sizes Lulesh

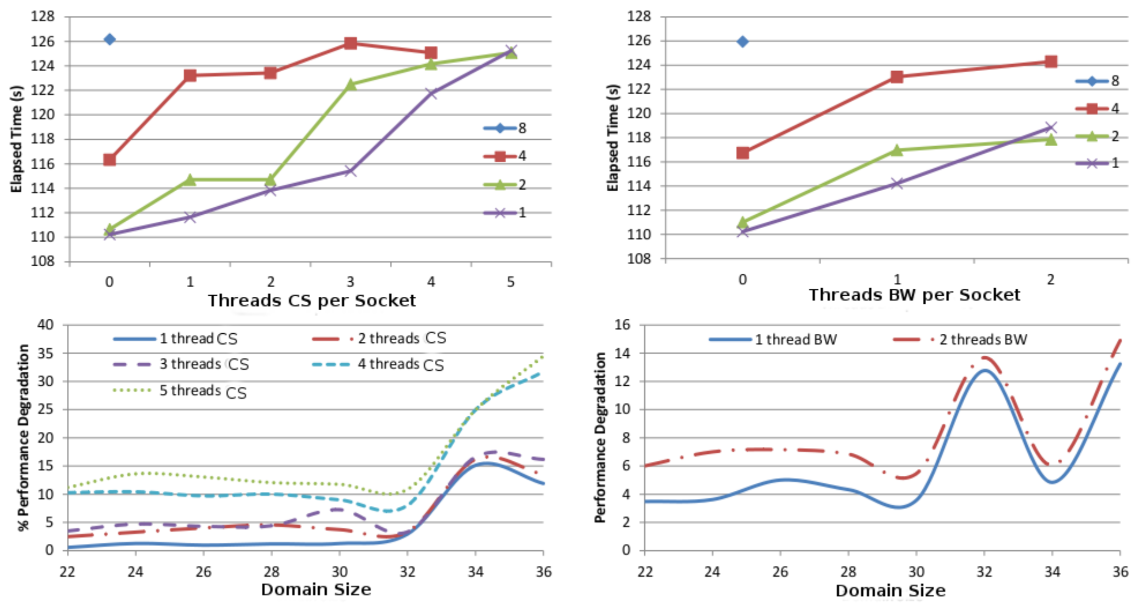

Fig. 11. Performance Degradations of Lulesh on 64 MPI tasks. The two top figures show results obtained considering several MPI mappings and using a 22x22x22 domain. The two figures at the bottom show results obtained when we map 1 MPI process per processor. The domains considered are between 22x22x22 and 36x36x36.

processes use more than 15MB of cache each. The bottom-right graph of Figure 11 shows that performance degradation is larger than 10% when domain sizes are 32 and 36 with one or two BWThrs. This is consistent with the fact the L3 cache memory is not large enough for sizes bigger than 30 so the application needs the memory bus regularly to fetch data into the cache memory.

The top graphs of figure 12 present the amount of bandwidth and storage used by each Lulesh process on the 22x22x22 cube input as the mapping of processes to processors is varied, using the data from the top graphs of Figure 11. The data shows that much like MCB, Lulesh uses more L3 bandwidth as processes are spread out across nodes and fewer processes share a common L3. Further, it shows that this application's storage use rises as processes are moved apart. This is likely because MPI communication buffers spend longer in the cache being transferred across sockets and nodes, making it less available for other data. The most interesting thing shown in this figure is that Lulesh processes do not always need the same amount of L3 storage. As such, we can see how the processes need between 3.5 and 7MB when the cube's size is 22x22x22 and between 7 and 20 MB when the size is 36x36x36.

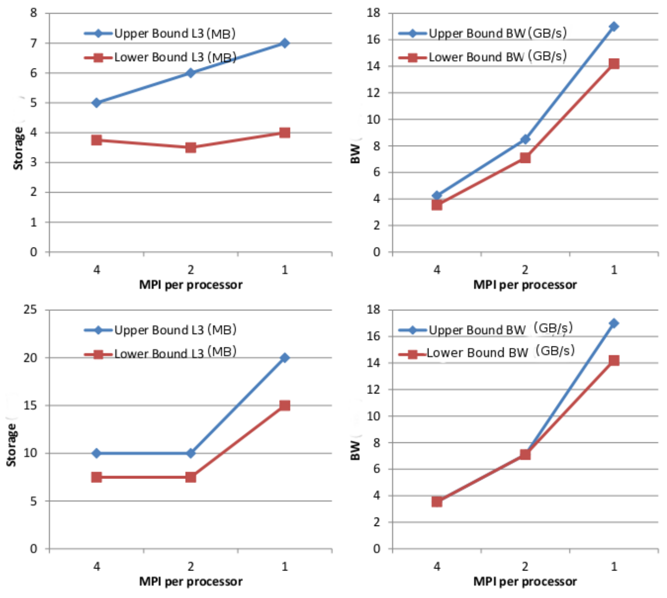

Fig. 12. Lulesh resource consumption depending on the MPI mapping. Two different cubes are considered: 22x22x22 in the top and 36x36x36 at the bottom.

V. Related Work

Some approaches [21] and [4] use interference threads to steal available cache storage from a targeted application. However, they do not control the working set size of their cache-stealing routines or validate the exact resources they utilize.

Eklov et al's work [6], [7] is a valuable contribution. Like us, Eklov et al develop interference workloads that utilize cache storage and bandwidth. However, their work falls short in a few critical areas. First, our methodology validates the independence between the BWThr and the cache interference (CSThr), while Eklov et al do not consider the possible impact of the Bandwidth Bandit on the cache storage capacity. As such, the performance slowdowns reported by Eklov et al can be misleading since the Bandwidth Bandit can erase some cache lines and thus increase applications cache miss rate. The resulting performance slowdowns are thus difficult to interpret since they are caused by interference in multiple resources. Second, while BWThr is much simpler than Eklov et al's Bandwidth Pirate, experiments show that it is at least as effective at reducing available memory bandwidth. As reported in the top of page 6 of [7], the Bandwidth Pirate method can steal up to 4.6 GB/s (43% of total system bandwidth) with an unknown impact on cache storage. Our method steals up to 32% of total bandwidth without significantly impacting cache storage. Third, our work proposes a method to precisely evaluate the effective reduction in cache capacity due to cache interference while Eklov et al use of a simple heuristic to estimate the point where their cache storage interference is inaccurate, what causes them to assign low accuracy to experiments where 66%-75% of cache capacity is stolen. Our more accurate validation methodology allows us to ensure accurate results when interfering with up to 87% of cache storage capacity.

In [14] a "bubble" kernel with tunable capacity and memory activity is presented. While the bubble is very useful on analyzing applications' performance degradation due to generic memory activity, it is not able to decompose such degradation into several factors, as our work does. As such, using the bubble we would be able to predict the performance interference between co-located applications, but not the particular resource that is exhausted. In [22] the approach is improved to provide accurate QoS control and maximized server utilization via interference measurement.

Finally, techniques to accurately measure the parameters of hardware components have been proposed [24], [23]. The authors argue that existing micro-benchmarks are inadequate, and present novel micro-benchmarks for determining the parameters of all levels of the memory hierarchy, including registers, all cache levels and the translation look-aside buffer. The experimental results show that the tool successfully determines memory hierarchy parameters on many current platforms.

VI. Conclusion

It is becoming critical to quantify how applications use the memory hierarchy to enable developers to identify performance bottlenecks. We present and validate the Active Measurement methodology, which quantifies an application's utilization of the memory hierarchy, specifically the storage and bandwidth of shared caches, and predicts the application's performance when the required memory resources are not available. Our approach is a significant improvement over the prior work. It provides information that today can only be derived by simulators but it is much faster than current simulation-based techniques and unlike simulation makes predictions for any architecture the application may run on with no information about its proprietary internal details. Further, it is much more actionable than performance counter analysis techniques because it can predict application performance when different amounts of key resources are available.

References

[1] Livermore unstructured lagrangian explicit shock hydrodynamics (lulesh). https://computation.llnl.gov/casc/ShockHydro/.

[2] Monte carlo benchmark. http://www.osti.gov/estsc/details.jsp?rcdid=4793.

[3] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. Accuracy evaluation of GEM5 simulator system. *IEEE International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, July 2012.

[4] X. Chen, C. Xu, R. P. Dick, and Z. M. Mao. Performance and power modeling in a multi-programmed multi-core environment. *Proc. of the Design Automation Conference (DAC)*, 2010.

[5] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 1999.

[6] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. *Proc. of the International Conference on Parallel Processing (ICPP)*, 2011.

[7] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: quantitative characterization of memory contention. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 457–458, New York, NY, USA, 2012. ACM.

[8] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.

[9] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma. On the nature of cache miss behavior: Is it 2? *J. Instruction-Level Parallelism*, 10, 2008.

[10] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, Dec. 1989.

[11] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. *Proceedings of the 2010 ACM/IEEE conference on Supercomputing (SC)*, 2010.

[12] K. A. Huck, A. D. Malony, S. Shende, and A. Morris. Scalable, automated performance analysis with tau and perfexplorer. *PARCO*, pages 629–636, 2007.

[13] P. Kogge. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, DARPA IPTO, 2008.

[14] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 248–259, New York, NY, USA, 2011. ACM.

[15] 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, 1995.

[16] J. Mellor-Crummey, R. Fowler, and D. Whalley. Tools for application-oriented performance tuning. *Proc. of the 15th International Conference on Supercomputing*, pages 154–165, 2001.

[17] N. Nethercote, R. Walsh, and J. Fitzhardinge. Building workload characterization tools with valgrind, October 2006.

[18] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance. *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC)*, 2003.

[19] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Open — speedshop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, pages 105 – 121, 2008.

[20] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2):46–55, July 2009.

[21] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. *Proc. of the Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[22] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. *SIGARCH Comput. Archit. News*, 41(3):607–618, June 2013.

[23] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. *Proceedings of the 2005 ACM SIGMETRICS international conference*, pages 181–192, 2005.

[24] K. Yotov, K. Pingali, and P. Stodghill. X-ray: A tool for automatic measurement of hardware parameters. *Proceedings of the 2nd Second International Conference on the Quantitative Evaluation of Systems*, pages 168–177, 2005.