Quantitative Characterization of Memory Contention

David Eklov, Nikos Nikoleris, David Black-Schaffer and Erik Hagersten
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{david.eklov, nikos.nikoleris, david.black-schaffer, eh}@it.uu.se

ABSTRACT

On multicore processors, co-executing applications compete for shared resources, such as cache capacity and memory bandwidth. This leads to suboptimal resource allocation and can cause substantial performance loss, which makes it important to effectively manage these shared resources. This, however, requires insights into how the applications are impacted by such resource sharing.

While there are several methods to analyze the performance impact of cache contention, less attention has been paid to general, quantitative methods for analyzing the impact of contention for memory bandwidth. To this end we introduce the *Bandwidth Bandit*, a general, quantitative, profiling method for analyzing the performance impact of contention for memory bandwidth on multicore machines.

The profiling data captured by the Bandwidth Bandit is presented in a *bandwidth graph*. This graph accurately captures the measured application's performance as a function of its available memory bandwidth, and enables us to determine how much the application suffers when its available bandwidth is reduced. To demonstrate the value of this data, we present a case study in which we use the bandwidth graph to analyze the performance impact of memory contention when co-running multiple instances of single threaded application.

1. INTRODUCTION

Prior research has shown that contention for shared resources, such as cache capacity and off-chip memory bandwidth, can have a large negative impact on application performance [7, 23]. Current trends of increasing core counts, without a corresponding growth in off-chip bandwidth, indicate that the pressure on shared memory resources will only increase in the future [18]. Methods and tools to aid the analysis of applications' performance sensitivities to resources sharing are therefore becoming increasingly important, both for application developers and system architects.

In the case of cache capacity, the Miss Ratio Curve (MRC) [16] is a quantitative tool for analyzing applications' sensitivity to contention. MRCs present applications' miss ratios as a function of their allotted cache capacity and can answer questions such as, how much an application suffers, in terms of cache misses, when its cache capacity is reduced. MRCs have been the foundation for many techniques to manage shared cache capacity [17, 21, 22]. Several other tools, such as Cache Pirating [9] and Stressmark [24], have been proposed that in a similar fashion plot various application performance metrics as a function of cache capacity. The left graph in Figure 1 shows data obtained using Cache Pirating for OMNet++. It presents OMNet++'s Cycles Per Instruction (CPI) as a function of its available cache capacity. This data has been used to predict and explain how cache contention impacts throughput in multiprogrammed environments on contemporary multicore architectures [9].

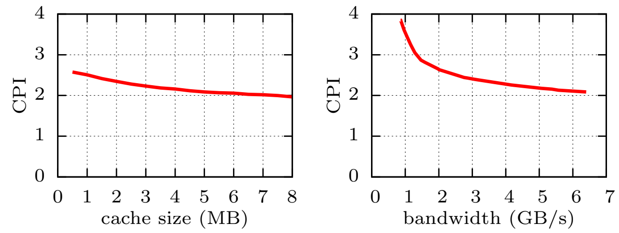


Figure 1: CPI as a function of cache (left) and bandwidth (right) for OMNet++ on an Intel Nehalem system.

While there are several general methods to analyze the performance impact of cache contention, less attention has been paid to general, quantitative methods for analyzing the impact of contention for off-chip memory bandwidth. The fact that contention for memory bandwidth can impact an application's performance, either by increasing its memory access latencies or reducing its available off-chip memory bandwidth, is widely understood. However, it is not always obvious when and by much how these factors impact application performance. To this end we introduce the *Bandwidth Bandit*, a general, quantitative, profiling method for analyzing the performance impact of contention for shared off-chip memory resources, and determining the applicationÕs degree of latency- and bandwidth-sensitivity.

The right graph in Figure 1 shows data obtained using the Bandwidth Bandit for OMNet++ on an Intel Ne-

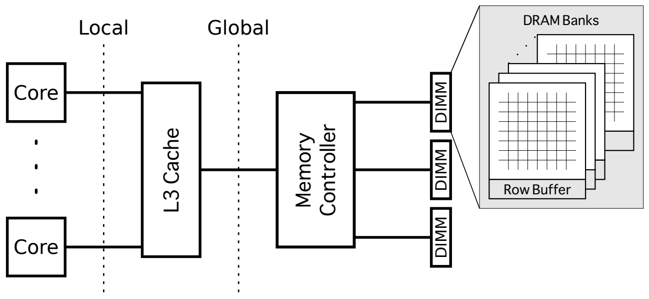


Figure 2: The memory hierarchy.

halem system. It presents OMNet++'s CPI as a function its available bandwidth and quantitatively shows how much OMNet++ suffers when its share of the available bandwidth is reduced. As such, this data enables a new dimension of resource contention analysis by enabling existing cache contention analyses (e.g., [9]) to be performed for bandwidth contention as well. Section 8 presents an example, showing how it can be used to explain how contention for memory bandwidth limits the scalability of multiprogrammed environments.

The design of the Bandwidth Bandit is inspired by Cache Pirating. It co-runs the application whose performance we want to measure (the Target) with a Bandit application that "steals" memory bandwidth. Varying the amount of bandwidth stolen by the Bandit, while measuring the Target's CPI, allows us to plot the Target's CPI as a function its available bandwidth. As we want to analyze applications' sensitivities to contention for memory bandwidth it is important that the Bandit does not steal shared resources other than bandwidth. For example, if the Bandit consumes large amounts of shared cache capacity, it might inadvertently cause the Target to slowdown and perturb the measurements.

2. BACKGROUND

2.1 Memory Hierarchy Organization

The memory hierarchy considered in this paper is that of the Intel Nehalem processor, shown in Figure 2. If a memory access cannot be serviced by the cores' private caches (not shown in the figure), it is first sent to the shared L3 cache. If the requested data is not found in the L3 cache, it is sent to the integrated Memory Controller (MC). The MC has three independent memory channels over which it communicates with the DRAM modules. Each channel consists of an address and a data bus. Memory requests are typically 64 bytes (one cache-line) and require multiple transfers over the data bus. Each DRAM module consists of several independent memory banks, which can be accessed in parallel, as long as there are no conflicts on the address and data buses. The combination of independent channels and memory banks provides for a large degree of available parallelism in the off-chip memory hierarchy.

The DRAM memory banks are organized into rows (also called pages) and columns. To address a word of data the MC has to specify the channel, bank, row and column of the data. To read or write an address, the whole row is first copied into the bank's row buffer. This single-entry buffer (also known as a page cache) caches the row until a different row in the same bank is accessed.

On a read or write access three events can occur: A *page-hit* when the accessed row is already in the row buffer and the data can be read/written directly; a *page-empty* when the row buffer is empty and the accessed row has to be copied from the bank before it can be read/written[1]; or a *page-miss* when a row other than the one accessed is cached in the row buffer. In the case of a page-miss, the cached row has to first be written back to the memory bank before the newly accessed row is copied into the row buffer. These three events have different latencies, with a page-hit having the shortest latency, and a page-miss having the longest.

2.2 Memory Hierarchy Performance

From a performance point of view the memory hierarchy can be described by two metrics: its latency and bandwidth. These two metrics are intimately related. Using Little's law [14], the average bandwidth achieved by an application can be expressed as follows:

$$bandwidth = transfer\_size \times \frac{MLP}{latency}, \qquad (1)$$

where MLP is the application's average Memory Level Parallelism, that is, the average number of concurrent memory requests it has in-flight, and *latency* is the average time to complete the application's memory accesses.

The above equation clearly illustrates that the bandwidth achieved by an application is determined by both its memory access *latency* and its memory *parallelism*. However, these parameters vary throughout the memory hierarchy, and from application to application. For example, at the bank level, the parallelism is limited by the number of banks. However, MCs typically queue requests to busy banks. From the higher-level perspective of the MC, the parallelism, or number of in-flight requests, will include the requests in these queues, and appear larger. The latency will also appear different, since the time spend in the queues has to be considered. The above equation will therefore have different values for latency and MLP depending on where it is applied in the memory hierarchy.

3. EXPERIMENTAL SETUP

The experiments presented in this paper have been

[1]Page-empties occur when the MC preemptively closes a page that hasn't been accessed recently to optimistically turn a page-miss into a page-empty.

run on a quad core Intel Xeon E5520 (Nehalem). Its cache configuration is detailed in the following table:

L1 cache	32k/32k, 8/4 way, inst./data, private
L2 cache	256k, 8 way, unified, private
L3 cache	8MB, 16 way, inclusive, shared

In this system, an L2 cache miss is sent to a Global Queue (GQ) [3] which tracks the in-flight L2 misses. The GQ has three queues for in-flight accesses: a 32-entry queue for loads, a 16-entry queue for stores, and a 12-entry queue for requests to the QuickPath Interconnect (QPI). In a single socket system, upon receiving a request for a cache line from one of the four cores, the GQ first sends a request to the shared L3 cache. If the cache line is not present in the L3 cache, it then sends the request to the MC.

The MC for this system has three memory channels. Our baseline setup uses one dual-ranked 4GB DDR3-1333 DIMM. For experimenting with different numbers of active memory channels we used up to three dual-ranked 2GB DDR3-1333 DIMMs. All DIMMs have 16 memory banks (8 per rank) and 8kB page caches.

Using a small micro-benchmark, we measured the access latencies of page-hits (82 cycles), page-empties (160 cycles) and page-misses (177 cycles). This micro-benchmark traverses a linked list such that each memory access is data dependent on the previous memory access. Its execution time is therefore limited by the memory access latencies which allow us to measure its access latencies. By carefully staging the linked list's layout in memory we ensure that the memory accesses results in the desired event.

4. SOURCES OF MEMORY CONTENTION

4.1 Limited Memory Parallelism

According to Eq.1, it appears that an application's bandwidth is strictly proportional to its number of parallel memory requests. However, the memory hierarchy cannot always accept as many parallel requests as the application can generate. Such limitations can result in contention and appear at both the *local level* (limitations on the number of requests individual cores can have in-flight) and the *global level* (limitations on the total number of in-flight memory requests in the shared memory hierarchy).

Local bottlenecks: Figure 3 shows the results of co-executing multiple instances of a small micro-benchmark whose MLP we can vary. The data shows the aggregate bandwidth for one to four instances of the micro-benchmark and for one, two and three active memory channels. For the case of a single instance (lower red line), regardless of the number of memory channels, the bandwidth increases (almost) linearly with the memory parallelism until it reaches an MLP of 10, at which point it levels off. This suggests that there is a limit of 10 in-flight memory requests for a single core.

By examining the data for two instances (green line), for two and three active memory channels (Figures 3(b) and 3(c)), we can see that this is indeed the *local* per-core limit. For two instances (two cores) the bandwidth increases linearly until the memory parallelism reaches 20 (10 per instance). This indicates that the system can readily reach a total memory parallelism of 20, and that the limit of 10 is the local limit for each core.

Global bottlenecks: The effects of global bottlenecks can be seen in the data for one active memory channel (Figure 3(a)). For a single channel, two or more instances causes the bandwidth to level off at about 7.5GB/s. This occurs when the per-instance memory parallelism is 8 (for 2 instances), 5 (for 3 instances), and 4 (for 4 instances). In all three cases this represents a combined memory parallelism of 16. We can therefore conclude that with one memory channel active the memory hierarchy can keep only 16 parallel memory requests in-flight at a given time. Repeating the analysis for two and three memory channels (Figures 3(b) and 3(c)) shows that the bandwidth levels off at 13.2GB/s and 15.9GB/s, respectively. This occurs when the memory parallelism reaches a total of 32, indicating that there is a *global* limit of 32 parallel memory requests with two or more memory channels active.

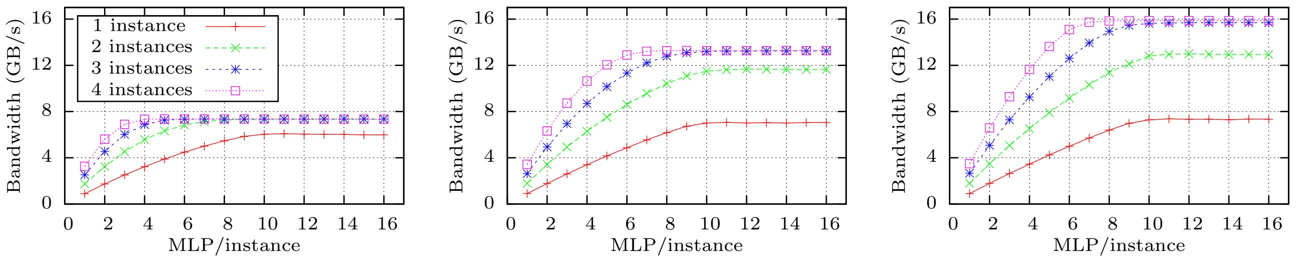
In the case of two or more active memory channels, we suspect that the limit of 32 memory requests is due to the 32-entry queue for loads in the GQ in our Nehalem system. In the case of one active channel, the limit may be due to the fact that there are only 16 memory banks per channel. However, from an application's point of view this distinction is unimportant. In both cases the application's parallel memory accesses will be queued somewhere in the memory hierarchy, and the finite length of these queues will impose a limit on the maximum MLP.

4.2 Reduced Access Latencies

According to Eq. 1 the bandwidth should increase linearly with the memory parallelism until any of the MLP limits are hit. However, Figure 3 clearly shows that this is not the case as the bandwidth rolls off smoothly. This is because the aggregate bandwidth increases when the MLP is increased. Increased bandwidth increases memory contention, which in turn can cause an increase number of page-misses and bank contention, ultimately resulting in increased access latencies[2].

Page-Misses: Memory contention can turn page-hits into page-misses and thereby increase access latencies. To cause a page-miss, it is enough that one thread accesses a bank whose page cache holds a row for another thread, forcing the cached row to be replaced. As

[2]There is a third way in which memory contention can increase access latencies: contention for the address and data buses. This, however, is much less significant.



(a) One memory channel (b) Two memory channels (c) Three memory channels

Figure 3: Aggregate bandwidth as a function of MLP for different numbers of memory channels.

the access latency in the case of a page-miss is about twice that of a page-hit, this can have large impact on application performance (see Section 3).

Bank contention occurs when two or more threads try to access the same bank at the same time. When this happens, only one of the requests can be issued to the bank and the other(s) must be queued in the MC until the bank is available, causing their latencies to increase.

5. THE BANDWIDTH BANDIT

The Bandwidth Bandit method enables us to measure how an application's performance is affected by contention for shared off-chip memory resources. It works by co-running the application whose performance we want to measure (the *Target*) with a *Bandit* application that generates contention for the shared off-chip memory resources. To accomplish this, the Bandit accesses memory at a specified rate and in a controlled pattern that ensures it generates the desired amount and type of contention. By measuring the Target's performance while varying the amount of contention the Bandit generates, we can obtain the Target's performance as function of contention for the memory system.

5.1 Requirements

Since we want to isolate the performance impact due to memory contention, it is important that the Bandit does not fight with the Target for any other shared resources. In particular, the Bandit must avoid using a significant amount of the shared cache, as the Target's performance may be sensitive to its shared cache allocation.

As we saw in Section 4, contention for shared off-chip memory resources can result in both reduced bandwidth and increased latencies, at different points in the memory hierarchy. These effects are due to 1) reduced memory parallelism, 2) increased bus and bank contention, and 3) an increased number of page-misses. To generate realistic memory contention, the Bandit must be able to cause all of the above.

1) Reduced memory parallelism occurs when co-running applications generate memory request at such a rate that they start to compete for the limited number of GQ entries.

2) Bus and bank contention arises when multiple applications accesses the same bank. However, to access the bank the applications must first generate memory accesses, have them queued in the GQ, and then gain access to the address and data buses for the access. If the rate at which the application access the bank is increased, the contention for that bank will increase, but this will also cause both more GQ entries to be allocated and more bus contention.

3) Increased page-misses is a function of both the co-running applications' relative access rates and their page locality, i.e. how many times they access a given page without intervening accesses to different pages. In general, applications with higher access rates are more likely to cause page-misses for other applications.

5.2 Implementation

In order to generate a specific amount of realistic memory contention the Bandit application has to be able to generate a specific amount of parallel memory accesses and access a set of banks at a given rate. To expose the impact of request reordering in the MC, the Bandit has to be able to vary the page locality within the access stream. To accomplish this we first need a mechanism to access individual memory banks.

In order access individual banks we allocate 32 large (2MB) pages. With only one memory channel active, one large page span across all (16) memory banks. (We discuss the case of more active memory channels below.) This allows us to access all memory banks from within a single large page. We initialize the large pages with 16 independent linked lists. Each list has one element in every large page that all reside in the same memory bank. (These elements are necessarily in different rows). Therefore, when traversing one of these linked lists the Bandit generates 32 memory requests to different rows within the same bank. Furthermore, the elements in a list are laid out such that they all map into the same cache-set. Traversing all 16 lists will therefore only thrash 16 cache-sets[3]. As the associativity of the

[3]While one could completely avoid thrashing the shared cache by using non-cacheable memory [2], the maximum access rate to this type of memory is too low to generate significant amounts of contention.

last-level cache on our system is 16 and the lists have 32 elements each, all accesses will result in cache misses.

To control the amount of row locality (i.e. the number of consecutive accesses to the same row), we can insert additional elements into the linked list that are allocated at addresses immediately following the original elements. To ensure that all access to the elements result in cache misses they are 64B aligned, which guarantees that they are on different cache lines. For example, to generate contention with a locality of four (e.g., every fourth access causes a page-miss), we insert three elements after each original element. However, for each additional element we will use one extra cache-set, limiting the amount of locality we can generate without consuming too much of the shared cache capacity. On our machine, a locality of eight uses 1.5% of the cache-sets.

When more than one memory channel is active (i.e. populated with DIMMs) the MC spreads the physical address space across the channels with a 64B granularity in a round robin fashion. In the case of two (three) channels, two (three) consecutive large pages are required to span all 32 (48) memory banks. However, in user space we have no control over whether the (virtual) pages we allocate are backed by physically consecutive pages or not. To work around this, we wrote a small kernel module that we can query for the physical address of a virtual page. This allows us to ensure that our allocated pages span the correct channels.

To place the elements in the linked list such that they reside in the same memory bank, we need to know how the MC maps physical addresses to banks, rows and columns. This information has been partially documented by Intel [1]. Guided by this information we were able to experimentally find the complete address mappings.

The Bandit application allocates linked lists as discussed above and traverses them at a rate that generates the desired amount of memory contention. As each core is limited to 10 in-flight memory requests, we run three parallel instances of the Bandit application to be able to generate high levels of contention Since we have different linked list for the different memory banks, we can control how much contention we generate for each bank individually.

6. RESULTS

6.1 Methodology

In this section we present data obtained using the Bandwidth Bandit method on a set of applications from the SPEC2006 [11] and PARSEC [5] benchmarks suites. We selected eight benchmarks (six from SPEC and two from PARSEC) that have large bandwidth demands; as such applications are believed to be more sensitive to memory contention. All benchmarks were run to completion with their reference input sets. Our goal is to investigate how sensitive individual threads are to memory contention and therefore we ran the PARSEC benchmarks with a single thread.

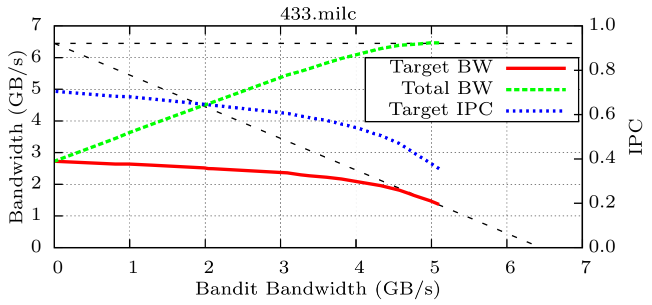


Figure 4: Bandwidth Bandit data: Target's bandwidth (left, red) and IPC (right, blue); and total system bandwidth (left, green), as a function of the bandwidth stolen by the Bandit.

To obtain the Bandit data we co-executed three instances of the Bandit application with benchmark application multiple times, each time increasing the bandwidth demand of the Bandit[4]. For every 100M instructions executed by the Target application[5], we recorded both the Target's and the Bandit's time stamp counter and number off-chip fetches from the hardware performance counters. All data presented throughout the rest of the paper represent one 100M instruction window of the most representative memory behavior. To find such windows, we used Cache Pirating [9] to measure fetch ratio curves (i.e. fetch ratio as a function of cache size) of each 100M window. These curves captures the memory behavior of the windows. To find the most representative window we applied a simple clustering algorithm which groups windows with similar fetch ratio curves and selected one window from the group with the most windows.

6.2 Bandwidth Bandit Data

Figure 4 shows an example of the raw data obtained using the Bandwidth Bandit for milc on our Nehalem system. The graphs show milc's bandwidth and IPC, and the total bandwidth (Target plus Bandit), as a function of the bandwidth stolen by the Bandit. When the Bandit does not steal any bandwidth, milc's *baseline bandwidth* is about 2.7GB/s and its *baseline IPC* is about 0.70. However, when the Bandit steals only 2GB/s, milc's bandwidth and IPC have dropped to 2.5GB/s and 0.65, respectively. At this point the total bandwidth is 4.5GB/s (2.5GB/s Target + 2GB/s Bandit). At increased Bandit bandwidths (moving to the right

[4]This overhead be avoided by dynamically changing the Bandit's bandwidth demand during execution, as has been successfully demonstrated for stealing cache space [9].
[5]This window size is commonly used for program phase detection [20] since programs typically have stable behaviours across many different metrics on this time scale.

on the x-axis), the total bandwidth levels out at a Bandit bandwidth of about 4.6GB/s, at which point we have reached the *saturation bandwidth*.

The saturation bandwidth (the horizontal dashed line in Figure 4) is the largest bandwidth that can be achieved for milc and three instances of the Bandit application, and is about 6.4GB/s. Therefore, when we increase the Bandit's bandwidth beyond 4.6GB/s, milc's bandwidth drops by the same amount, and it starts to follow the 45° dashed line, which indicates the points where the total bandwidth equals the saturation bandwidth.

Based on this data, we can make the following two observations. First, the bandwidth saturates well below the system peak bandwidth of 10.7GB/s. Second, the performance of milc starts to drop long before the saturation bandwidth is reached. This demonstrates that we cannot assume that co-running applications will not be impacted by bandwidth contention as long as the total bandwidth is below the system peak bandwidth.

6.3 Bandit Access Pattern

To investigate how the applications are affected by the Bandit's access pattern, we ran the experiment described in Section 6.1 with the Bandit's locality set to one, four and eight. (E.g., accessing one, four or eight cache lines from each row before moving on to the next row.) This causes the Bandit to generate contention that is more similar to that generated by an application with a sequential access pattern. The results for milc and soplex are shown in Figure 5. The other applications displayed similar behaviors and are not shown.

The figure shows both the Bandit data collected with a page locality of one (e.g., random access pattern) and four (e.g., sequential). Increasing the locality to eight had a negligible further impact. The first thing to note is that the saturation bandwidth is greater (increasing from 6.4GB/s to 7.3GB/s for milc). This is because the bank access times for the Bandit are reduced (due to the increased number of page-hits), and the throughput of the memory banks therefore increases.

Furthermore, if we look at the shape of the Target's bandwidth and CPI curves, the curves for a locality of four appear to be stretched. Indeed, if the data is presented as a percentage of the saturation bandwidth, the two curves match almost perfectly. (Figure 6) This suggest that applications' relative sensitivity to contention for bandwidth does not change significantly with its co-runners' access patterns. As a result, plotting the bandwidth stolen by the Bandit (the x-axis of the bandwidth graph) as percentage of the saturation bandwidth, factors out the impact of the access patterns.

6.4 Bandwidth Graphs

Figure 7 shows Bandwidth Bandit data for the eight benchmarks. In order to factor out the impact of the Bandits access pattern, bandwidths are express as a percentage of the saturation bandwidth. The resulting *bandwidth graphs* (BWGs) present a general, quantitative description of the applications' sensitivities to bandwidth contention, and allow us to determine how much an application suffers when its available bandwidth is reduced. In the following section, we use these BWGs to investigate the benchmark applications' sensitivities to contention for memory bandwidth.

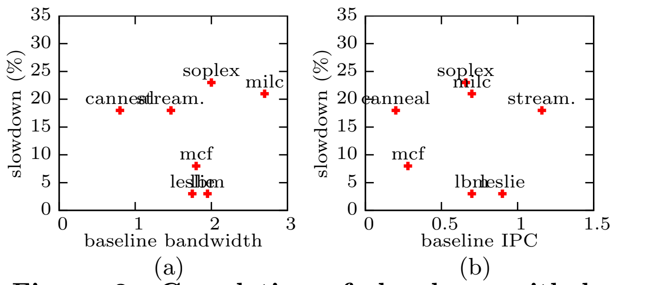


Figure 8: Correlation of slowdown with baseline bandwidth (a) and IPC (b) when the total bandwidth is 90% of the saturation bandwidth.

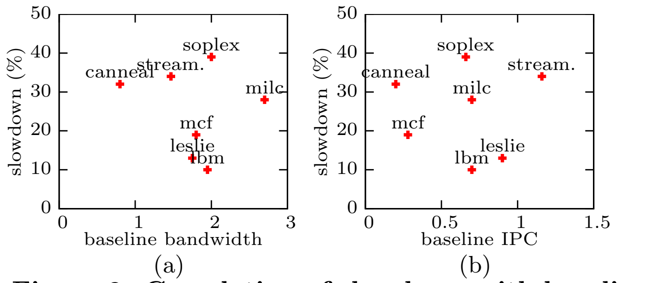
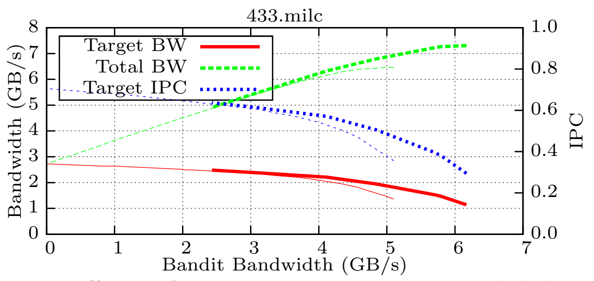


Figure 9: Correlation of slowdown with baseline bandwidth (a) and IPC (b) when the total bandwidth just reached the saturation bandwidth.

7. SENSITIVITY TO CONTENTION

Since applications with large baseline bandwidths have high demand for off-chip memory bandwidth, it is commonly believed that they are more sensitive to contention for off-chip bandwidth. To investigate this we plot the correlation between application slowdown due to memory contention and baseline bandwidth in Figure 8(a) and Figure 9(a). The baseline bandwidths are the applications' bandwidth demand when run alone on the machine. Figure 8(a) and Figure 9(a), show slowdown due to contention (baseline IPC relative to IPC under contention) at the point when the total bandwidth (Target plus Bandit) reaches 90% and 100% of the saturation bandwidth, respectively. This allows us to investigate the applications' sensitivities both before and after the total bandwidth saturates.

The graphs in Figure 8(a) and Figure 9(a) show hardly any correlation between slowdown and the baseline bandwidth. For example, in Figure 8(a), canneal and streamcluster have the lowest (but different) baseline bandwidths, but large (and very similar) slowdowns, while lbm and soplex have virtually the same baseline bandwidth, but very different slowdowns. Figure 9(a) shows



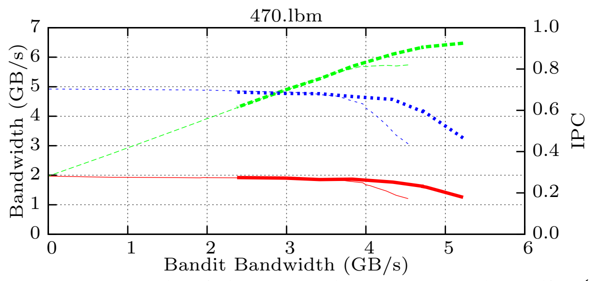
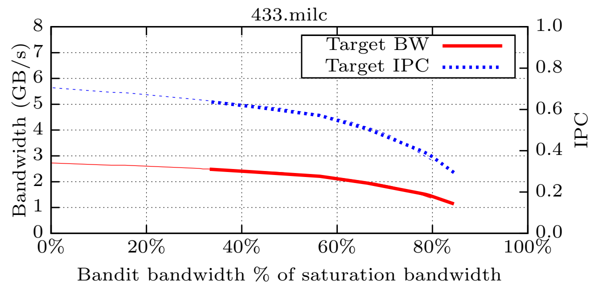


Figure 5: Effect of access patterns on sensitivity. Data captured with a random-access bandit (thin lines) and a sequential-access bandit (thick lines).



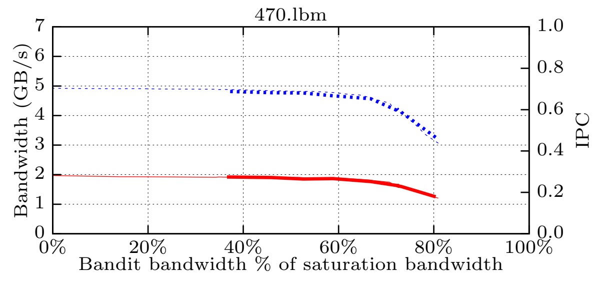


Figure 6: Bandit bandwidth as percentage of the saturation bandwidth. Data captured with a random-access bandit (thin lines) and a sequential-access bandit (thick lines).

a similar lack of correlation. Furthermore, Figure 8(b) shows the correlation between slowdown and the corresponding baseline IPCs. This graph shows even less correlation. These graphs indicate that neither the baseline bandwidth nor the baseline IPC are good indicators of an application's sensitivity to memory contention. To quantitatively analyze this lack of correlation we introduce the concepts of *latency-* and *bandwidth sensitivity*.

7.1 Latency vs. Bandwidth Sensitivity

As we saw in Section 4, memory contention can cause both increased access latency and/or reduced the available memory parallelism. Roughly, when the total bandwidth is below the saturation bandwidth contention can cause increased latencies. When the total bandwidth reaches the saturation bandwidth, increased contention causes both increased latencies and reduced memory parallelism. Furthermore, as shown in Figure 8 and Figure 9, different applications exhibit different sensitivities to memory contention. Some applications experience large slowdowns as the latency increases but the total bandwidth is still below the saturation bandwidth. These applications are *latency sensitive*. Other applications experience significant slowdowns only when the total bandwidth saturates. These applications are *bandwidth sensitive*.

Latency Sensitivity: Milc and soplex are latency sensitive applications. They suffer the highest slowdowns before the bandwidth saturates (see Figure 8(a)). Before the bandwidth saturates contention mainly results in increased access latencies. The large slowdowns experienced by milc and soplex therefore suggests that they are sensitive to increased access latencies. This can be seen in their BWGs (Figure 7) where their IPCs drop significantly long before the total bandwidth saturates.

Bandwidth Sensitivity: Lbm and leslie3d are bandwidth sensitive applications. They experience only modest slowdowns before the bandwidth saturates (see Figure 8(a)). At the point where the total bandwidth saturates, the memory parallelism becomes the bottleneck. At this point, if the Bandit (or other co-runners) steals additional bandwidth, it will use more of the available memory parallelism (e.g., GQ entries and memory banks). Hence, the Target gets to use less, and its bandwidth is therefore reduced by the same amount stolen by the Bandit. For example, lbm performs a stencil computation and has a very regular access pattern. The prefetchers in modern processors will easily detect this pattern and prefetch the data in advance, thereby hiding its access latency. The performance of lbm is therefore not directly affected by increased access latencies as long as there is enough memory parallelism available and the prefetcher can fetch far enough ahead. This can be seen in lbm's and leslie3d's BWGs (Figure 7) where their IPC virtually flat until the point where the bandwidth saturates.

8. CASE STUDY

In this case study we show how to use BWGs obtained with the Bandwidth Bandit to analyze how contention for memory bandwidth impacts throughput as we increase the number of co-running instances of OMNet++.

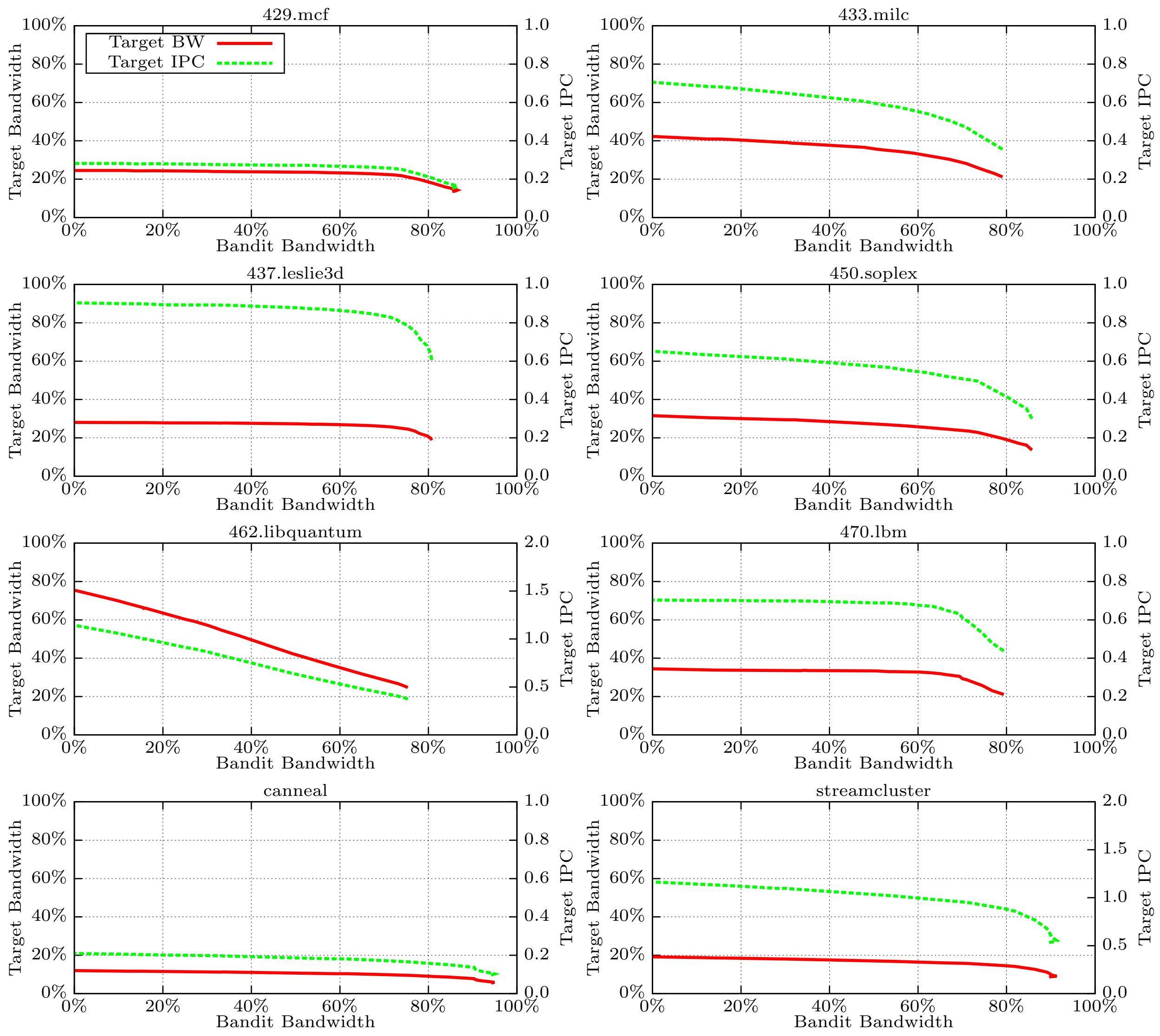


Figure 7: BWGs: Target's bandwidth (left, red) and IPC (right, blue), as a function of bandwidth stolen by the Bandit (x-axis). Bandwidths are presented as a percentage of the saturation bandwidth.

8.1 Experimental Setup

In this study we are concerned with the impact of contention for memory bandwidth. Therefore, to eliminate the impact of cache contention, we use cache coloring to partition the 8MB shared L3 cache of our Nehalem machine into four 2MB partitions, effectively emulating a processor with four 2MB private L3 caches. To achieve this we use an adaption Lin et al.'s cache coloring patch [13] for Linux.

To obtain the reference throughputs, we co-ran one to four instances of OMNet++ on the above setup, pinned to different cores and cache partitions, and measured the throughput using hardware performance counters.

8.2 Estimating Throughput

Given the BWG of an application, finding the throughput of a given number of co-running instances is a two-step process. First, we use the bandwidth graphs to find the bandwidths of the co-running instances. Then, we use these bandwidths to find the co-running instances' individual IPCs, which gives us the overall throughput.

Bandwidth: When co-running multiple instances of the same application, all instances get an equal amount of bandwidth. Therefore, finding how much bandwidth one of the co-running instances get amounts to finding the (x, y)-point on its bandwidth graph where $y = x$, $y = 2x$ and $y = 3x$ for two, three and four co-running instances, respectively. These equations can be easily solved using standard fixed-point methods. The solutions of are marked with circles in Figure 10.

Throughput: To find the IPC of one of the co-running instances, we first compute the total bandwidth

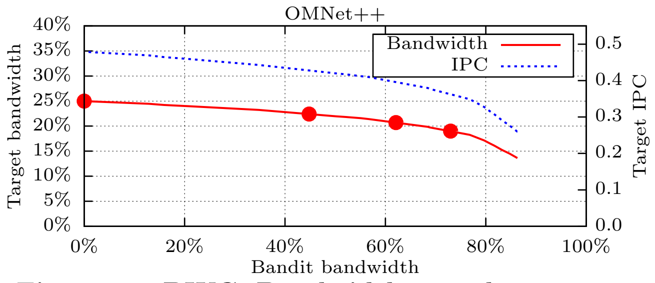


Figure 10: BWG. Bandwidths are shown as percentage of the saturation bandwidth.

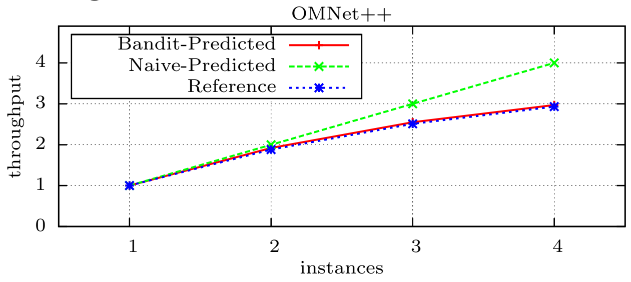


Figure 11: Normalized Throughput

of its co-runners, T (the sum of their individual bandwidths). Then, we use the first instance's IPC curve to find its IPC when its co-runners have a total bandwidth of T. Since we are co-running multiple instances of the same application, all co-runners have the same IPC, and the overall throughput is therefore the sum of the co-running applications' individual IPCs.

8.3 Results

Figure 11 show the throughput predicted using the Bandwidth Bandit ("Bandit-Predicted") and the reference throughput ("Reference"). It also show a "naive" throughput prediction ("Naive-Predicted"). Because the baseline bandwidth demand of OMNet++ is slightly less than 25% of the saturation bandwidth, which is less than the system peak bandwidth, we would not expect four instances to saturate the bandwidth. Therefore, without any additional information, our best (although naive) prediction would be that the throughput scales linearly.

The curve labeled "Bandit-Predicted" is the throughput predicted using the Bandwidth Bandit. This prediction almost perfectly match the reference throughput. This almost perfect match between the "Bandit-Predicted" and the "Reference", and the large discrepancy between the "Naive-Predicted" and the "Reference" emphasize the importance of using the contention aware data provided by the Bandwidth Bandit.

9. RELATED WORK

9.1 Cache Contention

Several methods have been proposed to reduce negative impacts of contention by actively manage both shared cache capacity and memory bandwidth. While some of these methods use fairly simple measures for contention (e.g. cache miss ratios [10]) their management desicions are based on how much applications benefit/suffer when their allocation of the managed resource is increased/reduced. These methods use different mechanisms (e.g. partitioning [13] or scheduling [25]) and have different objectives (e.g. maximizing throughput [21, 17] or fairness [12, 26]). In the case of partitioning, the managed resource is partitioned and distributed among the co-running applications/threads, effectively avoiding contention. In the case of scheduling the contention is minimized by selecting which applications to co-run. To make this selection, both the amount of contention applications generate, as well as their sensitivity to contention from others, have to be considered.

Contention for shared cache capacity has been well studied and understood. How much an application benefits from the amount of cache capacity it receives can be quantified by its Miss Ratio Curve (MRC). Many methods have been proposed to collect MRCs [6, 4, 22, 19]. However, miss ratio, i.e. cache misses per instruction, does not necessarily correlate well with performance. Eklov et al. [9] presented Cache Pirating, a method to measure performance, such as CPI and off-chip bandwidth, as a function of shared cache space.

9.2 Bandwidth Contention

Dey et al. [7] and Tang et al. [23] present studies on how contention for both cache capacity and memory bandwidth impact the performance of multithreaded applications. To evaluate the effects of contention they vary the assignment of applications to cores to adjust cache and bandwidth sharing.In terms of memory contention they conclude that higher bandwidth applications generate more contention and are more sensitive to it.

Mars et al. [15] present the Bubble-Up method, similar to Doucette et al. [8]'s base vectors, which measures both applications' sensitivities and contentiousness by co-running them with a set of micro benchmarks. Furthermore, they propose a resources aware scheduling algorithm that leverages the Bubble-Up data. This work has many similarities to the approach taken in this paper. However, the key difference, is that their micro benchmarks stress all the resource in the memory hierarchy at the same time, while the Bandwidth Bandit goes to great length to only stress ("steal") memory bandwidth, which makes it more suitable for general analysing of applications' sensitivity for bandwidth contention.

Xu et al. [25] found that the common scheduling policy of co-scheduling applications to reach a total bandwidth close the system's peak degrades performance more than expected. They further found that bandwidth demands are typically bursty, and that using averages to estimate the total bandwidth is therefore inaccurate. Our results suggests that their observations might in addition be due to the latency sensitivity of their applications.

Rogers et al. [18] studied the impact off-chip bandwidth will have on future multicore scaling. They found that the performance of future multi-cores will be severely limited by the lack of bandwidth scaling with conventional techniques, thereby highlighting the importance of understanding the impact of off-chip bandwidth contention, for present and future chip-multi processors.

10. CONCLUSIONS

The goal of this work is to develop a method that enables us to quantitatively understand how contention for off-chip memory bandwidth affects applications' performance. Using the Bandwidth Bandit method introduced in this paper we were able to quantitatively analyze application latency sensitivity and bandwidth sensitivity. We see that latency sensitive applications are likely to experience large slowdowns (up to 25%) before the memory hierarchy is saturated, while the performance of applications that are not latency sensitive is not significantly affected until the memory hierarchy becomes saturated. This new quantitative method provides the data needed to explain and understand the impact of sharing off-chip bandwidth on modern hardware.

11. REFERENCES

[1] Intel Xeon processor 5500 series datasheet volume 2, Apr. 2009.

[2] Intel 64 and IA-32 architectures developer's manual: Vol. 3A, May 2011.

[3] Intel 64 and IA-32 architectures optimization reference manual, June 2011.

[4] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *SIGMETRICS*, 2005.

[5] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Jan. 2011.

[6] D. Eklov and E. Hagersten. Statstack: Efficient modeling of LRU caches. In *Proc. of ISPASS*, 2010.

[7] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *ISPASS*, 2011.

[8] D. Doucette and A. Fedorova. Base Vectors: A Potential Technique for Microarchitectural Classification of Applications. In *Proc. of WIOSCA*, 2007.

[9] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *Proc. of ICPP*, 2011.

[10] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proc. of USENIX*, 2005.

[11] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34, 2006.

[12] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proc. of PACT*, 2006.

[13] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. of HPCA*, 2008.

[14] J. D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9, 1961.

[15] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible colocations. In *Proc of MICRO*, 2011.

[16] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9, 1970.

[17] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of MICRO*, 2006.

[18] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In *Proc. of ISCA*, 2009.

[19] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality Approximation Using Time. *SIGPLAN Not.*, 42(1), 2007.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proc. of PACT*, 2001.

[21] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. of HPCA*, 2002.

[22] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proc. of ASPLOS*, 2009.

[23] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proc. of ISCA*, 2011.

[24] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *Proc. of ISPASS*, 2010.

[25] D. Xu, C. Wu, and P. C. Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Proc. of PACT*, 2010.

[26] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proc. of ASPLOS*, 2010.