EMBA: Efficient Memory Bandwidth Allocation to Improve Performance on Intel Commodity Processor

Yaocheng Xiang^{*†}
Peking University
yaocheng_x@pku.edu.cn

Chencheng Ye[‡]
Huazhong University of Science and
Technology
yechencheng@gmail.com

Xiaolin Wang^{*†}
Peking University
wxl@pku.edu.cn

Yingwei Luo^{*†}
Peking University
lyw@pku.edu.cn

Zhenlin Wang
Michigan Technological University
zlwang@mtu.edu

ABSTRACT

On multi-core processors, contention on shared resources such as the last level cache (LLC) and memory bandwidth may cause serious performance degradation, which makes efficient resource allocation a critical issue in data centers. Intel recently introduces Memory Bandwidth Allocation (MBA) technology on its Xeon scalable processors, which makes it possible to allocate memory bandwidth in a real system. However, how to make the most of MBA to improve system performance remains an open question. In this work, (1) we formulate a quantitative relationship between a program's performance and its LLC occupancy and memory request rate on commodity processors. (2) Guided by the performance formula, we propose a heuristic bound-aware throttling algorithm to improve system performance and (3) we further develop a hierarchical clustering method to improve the algorithm's efficiency. (4) We implement these algorithms in EMBA, a low-overhead dynamic memory bandwidth scheduling system to improve performance on Intel commodity processors. The results show that, when multiple programs run simultaneously on a multi-core processor whose memory bandwidth is saturated, the programs with high memory bandwidth demand usually use bandwidth inefficiently compared with programs with medium memory bandwidth demand from the perspective of CPU performance. By slightly throttling the former's bandwidth, we can significantly improve the performance of the latter. On average, we improve system performance by 36.9% at the expense of 8.6% bandwidth utilization rate.

^{*}Also with Shenzhen Key Lab for Cloud Computing Technology & Applications, SECE, Peking University, Shenzhen, China

[†]Also with Pengcheng Laboratory, Shenzhen, China

[‡]Also with National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6295-5/19/08...\$15.00

<https://doi.org/10.1145/3337821.3337863>

KEYWORDS

Memory bandwidth allocation; Multi-core architectures

ACM Reference Format:

Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. 2019. EMBA: Efficient Memory Bandwidth Allocation to Improve Performance on Intel Commodity Processor. In *48th International Conference on Parallel Processing (ICPP 2019)*, August 5–8, 2019, Kyoto, Japan. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3337821.3337863>

1 INTRODUCTION

Multi-core processors are widely used in data centers. Contention on shared resources such as the last level cache (LLC) and memory bandwidth may cause serious performance degradation and unfairness, which makes efficient source allocation a critical issue in data centers. A number of methods have been proposed to address LLC contention in real systems in order to improve system throughput [8, 36, 37] or mitigate unfairness [8, 14, 29, 37]. In the recent decade, memory bandwidth scheduling has been extensively studied using simulation [7, 12, 13, 15–17, 21, 24, 25, 28, 43]. These studies focus on controlling bandwidth at the source (i.e., CPU) by throttling request rate [7, 12, 15, 17, 21, 43], or allocating bandwidth at the target (i.e., memory controller) by prioritizing queued requests [16, 24, 25, 28], or combining these two technologies [13]. However, none of these studies can be applied into real systems directly. An indirect solution was to manage the shared LLC efficiently to protect cache-sensitive workloads from the cache-poisonous ones [8, 36, 37]. They analyze the locality of programs to make memory accesses hit CPU cache as many as possible and thus reduce avoidable memory interference. Although this method mitigates bandwidth contention in some scenarios, it cannot allocate memory bandwidth according to demands.

Intel recently introduces Memory Bandwidth Allocation (MBA) technology on its Xeon scalable processors [1], which provide hardware support for throttling memory access rates for the cores over-utilizing bandwidth. Our goal is to improve system performance via dynamic memory bandwidth throttling. In this work, we mainly make the following contributions:

(1) We explore the relationship between a program's performance and its LLC occupancy and memory request rate on Intel commodity processors and formulate it as a quantitative model to

guide the application of Intel Cache Allocation Technology (CAT) and MBA.

(2) Using the performance model, we propose a heuristic bound-aware bandwidth-throttling algorithm to dynamically explore the performance upper bound of co-running programs without *a priori* knowledge.

(3) We further introduce a hierarchical clustering method to improve the efficiency of our throttling algorithm.

(4) We implement these algorithms in EMBA, a low-overhead dynamic memory bandwidth scheduling system to improve performance on Intel commodity processors.

(5) We find that, when multiple programs are running simultaneously on an Intel multi-core processor, the programs with high memory bandwidth demand usually use bandwidth inefficiently compared with the programs with medium memory bandwidth demand from the perspective of CPU performance. By slightly throttling the former's bandwidth, we can significantly improve the performance of the latter.

The rest of this paper is organized as follows. In Section 2, we introduce the background about Intel MBA. Section 3 explains our performance model. Section 4 introduces our bound-aware throttling and hierarchical clustering algorithms in detail. We evaluate our method in Section 5 and survey the related work in Section 6. Finally, Section 7 concludes this work.

2 BACKGROUND

Intel MBA is a new feature of Intel Resource Director Technology (RDT) introduced to the Intel Xeon scalable processor family, which provides indirect and approximate control over the available memory bandwidth of each core [1].

Intel introduces a programmable request rate controller between the cores and the high-speed interconnect to control applications' memory accesses. Intel MBA mainly has the following characteristics:

- The basic resource management object is Class of Services (CLOSs). We allocate the LLC or memory bandwidth resources for each CLOS and assign applications to these CLOSs. Applications in the same CLOS share the same resources. The number of CLOS supported for the MBA feature may be smaller than those of other resources such as L3 CAT. In our system (Table 2), L3 CAT supports 16 CLOSs while MBA supports 8 CLOSs.
- The memory bandwidth throttling value is input as a percentage and the upper-bound and granularity of MBA is machine related. In our system, the following eight values are acceptable: 10%, 20%, 30%, 40%, 50%, 60%, 90%, 100%.
- Intel MBA throttles memory accesses in an indirect way. It actually throttles all the memory accesses that miss an L2 cache including those accesses that may hit LLC. MBA can thus control well the bandwidth-intensive applications that do not use the LLC effectively.

Intel warns that *the throttling values exposed by MBA are approximate, and are calibrated to specific traffic patterns. As workload characteristics vary, the throttling values provided may affect each workload differently. In cases where precise control is needed, the*

Memory Bandwidth Monitoring (MBM) feature can be used as input to a software controller which makes decisions about the MBA throttling level to apply. According to a recent study [27] and our evaluation, the upper bound and granularity of MBA's throttling value vary dramatically under different circumstances and are very hard to predict. It is necessary to adopt a dynamic heuristic scheduling strategy according to real time performance and bandwidth occupation of programs.

3 PERFORMANCE MODEL

Previous research using simulation finds that a program's performance is related to the rate at which it accesses the memory subsystem. MISE [32] observes that the performance of a memory-bound application is proportional to its memory accesses rate. ASM [31] further observes that the same relationship exists between an application's performance and its cache accesses rate. Zhao et al. [42] propose a regression approach to predict performance interference between co-running applications. In this work, we build a performance model to guide the use of MBA in real systems.

In order to schedule memory bandwidth effectively, we first explore the relationship between a program's performance and its available memory resources (i.e., LLC and memory bandwidth). We run each of the 23 SPEC CPUrate2017 benchmarks alone under every legal LLC and memory bandwidth allocation scheme, i.e. 11 (LLC ways) $\times$ 8 (memory bandwidth throttling levels) for our machine. In the exhaustive evaluation of these tests, we find that when given a certain amount of LLC way, a program's instruction per cycle (IPC), i.e. performance, is proportional to its memory bandwidth while # LLC way determines the slope of the proportional relationship.

Here we take two typical programs as examples, cache non-sensitive 503.bwaves_r and cache sensitive 510.parest_r. In Figures 1a and 1b, vertical axis represents the performance of a program and horizontal axis represents the memory bandwidth that it occupies. Points with different colors represent different # LLC ways we allocate to the program.

For each program, when the memory bandwidth it occupies increases, its performance will improve proportionally. The difference is that for cache sensitive 510.parest_r, the more # LLC ways we allocate to it, the less memory bandwidth it will demand, while cache non-sensitive 503.bwaves_r benefits little from a larger cache partition. We try to explore the quantitative relationship between a program's IPC and two factors, #LLC way and memory bandwidth. According to our observation,

$$IPC = f(\#LLC\ way) \times bandwidth \quad (1)$$

We assume f may be related to the program's locality characteristics, i.e. miss ratio curve (MRC). Thus, we further measure every program's MRC (Figures 1c and 1d) with CAT and substitute the value into Figures 1a and 1b. As a result, we get the proportional relationship between $IPC \times L3MissRatio$ and memory bandwidth (Figures 1e and 1f). Equation 1 can be converted into:

$$IPC = \frac{K}{MR(\#LLC\ way)} \times bandwidth, K \text{ is a constant} \quad (2)$$

An intuitive explanation is that, memory system can be abstracted as a queue where the number of requests entering the

Figure 1: Exhaustive evaluation of 503.bwaves_r and 510.parest_r under every legal LLC and memory bandwidth allocation scheme

queue (issued from core) is equal to the number of requests leaving the queue (served by DRAM) during a long enough period of time. Since the number of memory accesses in 1 second is much larger than the length of the queue, we take 1 second case as an example. In 1 second,

$$\text{Input data size} = 1 \times \text{CPU Frequency} \times \text{IPC} \times \text{API}_{llc} \times \text{MR}(\#LLC \text{ way}) \times k \quad (3)$$

$$\text{Output data size} = \frac{\text{Memory Bandwidth}}{\text{size of cache line}} = \frac{\text{Memory Bandwidth}}{S} \quad (4)$$

Thus,

$$\text{IPC} = \frac{\text{Memory Bandwidth}}{\text{CPU Frequency} \times \text{API}_{llc} \times \text{MR}(\#LLC \text{ way}) \times S \times k} \quad (5)$$

API_{llc} represents #accesses to the LLC per instruction which is program related and CPU Frequency is machine related. $\text{MR}()$ is related to MRCs, which can be profiled both online [8, 33, 37] and offline. k represents a correction to the formula according to memory level parallelism (MLP), such as outstanding misses caused by a same memory block. All these parameters except k can be measured in advance. With these parameters of all the co-run

programs profiled, at any moment we can measure the current value of IPC , memory bandwidth and LLC size for each program and then derive the value of k . Further, we can predict each program's IPC under different combinations of LLC size and memory bandwidth .

Suppose there are currently N programs running in the system. Each program occupies one physical core exclusively. We use speedup in IPC to quantify the improvement of program i 's performance:

$$\text{speedup}_i = \frac{\text{IPC}_i}{\text{IPC}_{base_i}} = \frac{\text{BW}_i}{\text{BW}_{base_i}} \times \frac{\text{MR}(\#LLC \text{ way}_{base_i})}{\text{MR}(\#LLC \text{ way}_i)} \quad (6)$$

IPC_i , BW_i and $\#LLC \text{ way}_i$ represent the IPC , bandwidth, and $\#LLC \text{ way}$ of program i in the current co-run case while IPC_{base_i} , BW_{base_i} and $\#LLC \text{ way}_{base_i}$ are those in the co-run case where all programs compete for all system resources without any intervention such as cache partitioning and bandwidth allocation. $\text{MR}()$ represents the MRC of program i , which is execution independent.

We measure system performance using Weighted Speedup (WS) [8, 9, 30]:

$$\text{WS} = \frac{1}{N} \sum_{i=1}^N \text{speedup}_i = \frac{1}{N} \sum_{i=1}^N \frac{\text{BW}_i}{\text{BW}_{base_i}} \times \frac{\text{MR}(\#LLC \text{ way}_{base_i})}{\text{MR}(\#LLC \text{ way}_i)} \quad (7)$$

According to our performance model, we can figure out the optimal LLC and memory bandwidth allocation under these restrictions:

$$\begin{cases} \sum_{i=1}^N \#LLC \text{ way}_i = \text{Total } \#LLC \text{ way} \\ \sum_{i=1}^N \text{BW}_i \leq \text{Total BW} \\ 0 \leq \text{BW}_i \leq \text{BW}_{solo_i} \end{cases} \quad (8)$$

BW_{solo_i} represents the memory bandwidth occupied by the i -th program when it runs alone.

There are still some challenges though the optimal solution can be predicted according to our equations. The first is the granularity of resource allocation. The granularity that CAT and MBA directly support may not match the optimal scheme of LLC and memory bandwidth. Previous research about cache partitioning with CAT proposes some methods for reducing partitioning granularity [8, 29, 37] while, to the best of our knowledge, MBA remains unexplored. Second, MBA itself doesn't support precise bandwidth allocation as we have discussed in Section 2. Thus, we need to develop a well-designed dynamic throttling strategy to reduce the granularity of bandwidth allocation and try to maximize system performance, which is our motivation.

In this work, we focus on the optimization of memory bandwidth allocation and assume that the shared LLC has been well partitioned. We leave each core free to compete for the whole LLC. Previous work that focuses on cache allocation with CAT adopts a partial sharing method in order to reduce the granularity of partitioning [37]. Their cache allocation scheme can be divided into multiple full-sharing subsets. Our online bandwidth scheduling strategy can be applied to these subsets directly.

With the LLC already allocated, all variables in Equation 7 are known except BW_i . We define the product of all these known variables as Bandwidth Efficiency (BE). For program i :

$$BE_i = \frac{MR(\#LLC\ way_{base_i})}{BW_{base_i} \times MR(\#LLC\ way_i)}$$

Given a certain amount of memory bandwidth, the higher the BE of a program is, the greater its contribution to system throughput will be. Then, bandwidth allocation can be transformed into a standard Linear Programming (LP) problem:

$$\begin{aligned} \text{Max} \quad & BE_1 BW_1 + \dots + BE_N BW_N \\ \text{s.t.} \quad & BW_1 + \dots + BW_N \leq \text{Total BW} \\ & 0 \leq BW_1 \leq BW_{solo_1} \\ & \vdots \\ & 0 \leq BW_N \leq BW_{solo_N} \end{aligned} \quad (9)$$

We sort BE_i from large to small, and it is easy to prove that we can achieve the theoretical maximum system performance by assigning BW_i its maximum value in the order of BE_i .

Correctness condition. In the subsequent experiments, we find that under the effect of MBA, the overall bandwidth utilization of the system is almost always reduced. Programs that occupy more bandwidth tend to utilize bandwidth better. Bandwidth loss caused by throttling these programs is almost always larger than the extra bandwidth that other programs can utilize. Limiting their bandwidth may slightly reduce the bandwidth utilization of the system, but can significantly increase the efficiency of CPU. In an actual multi-program co-running environment, every target program has a bandwidth upper-bound. When this bound is not reached, throttling another memory intensive program can allocate more bandwidth to the target program and improve its system. When the bound is reached, there will be no performance improvement. When the upper bounds of most programs are reached, the system bandwidth utilization can become lower due to throttling.

4 METHODOLOGY

In order to explore the bandwidth upper bounds of co-running programs and maximize the overall system performance, we adopt a heuristic approach to find the optimal solution without any prior application knowledge, which avoids the overhead caused by offline or online profiling in advance. There are mainly three reasons why we choose a heuristic method to find a near-optimal solution:

a. Inaccuracy of MBA. As discussed in Section 2, Intel implements memory bandwidth allocation by throttling the memory accesses caused by L2 misses of each core and the throttling values are approximate and related to specific workload characteristics. Based on our exhaustive evaluation of each benchmark under every valid throttling value, we find MBA has different effects on different programs, and the programs with higher memory bandwidth intensity are affected more significantly. Figure 2 shows the impact of MBA on two programs with different memory intensity. The vertical axis represents program's actual bandwidth. Lines of different colors represent different MBA parameters, and 100% indicates program's maximum bandwidth when no restrictions are imposed.

It can be seen that MBA is not exact and has different effects on different programs.

Figure 2: Actual memory bandwidth of 500.perlbenc_r and 519.lbm_r under every valid throttling value in their first 100 second execution

b. Unpredictability of bandwidth bound. Section 3 develops a relationship between a program's IPC and the memory bandwidth it occupies. According to our performance model, during a stable period, a program's IPC will increase proportionally with the increase of available memory bandwidth until it reaches its bandwidth upper bound. We actually transform the complicated prediction of IPC into relatively intuitive prediction of memory bandwidth. However, there are many factors involved in memory bandwidth contention. A DRAM access due to a low level cache miss has to participate in multi-level queue management, such as MSHR and SuperQueue. In addition, the specific scheduling strategy of Intel integrated memory controller (IMC) is also a black box. Besides, different programs have different data dependency and memory access patterns. When multi-programs co-running, they act differently with respect to the parallelism of channel and bank as well as the locality of row buffer. Therefore, when a program's bandwidth is throttling, it is difficult to predict how much other programs' bandwidth will increase.

c. Monotonicity of optimization direction. Although the bandwidth upper-bound is hard to predict, when a program's bandwidth is throttled, other programs can certainly benefit from it. According to Equation 9 derived from our performance prediction model, in order to achieve the maximum WS, we must throttle the programs with lower BE, and make the higher ones as close as possible to their bandwidth upper-bound. The direction to find the optimal solution is monotonic.

4.1 Bound-aware Throttling

According to our performance model, bandwidth should be allocated to the programs with larger BE values. Since MBA is a throttling-based technology, we need to throttle the bandwidth of the programs with smaller BE values to benefit the rest. As discussed before, the bandwidth bound of a certain program is hard to predict in a multi-program co-running environment. Therefore, we develop a bound-aware throttling algorithm to heuristically explore the bandwidth upper bounds of the programs to be optimized and maximize system performance.

Figure 3: State transition diagram of binary search

Algorithm 1 Bound-aware_Throttling

Input: N : the number of cores/programs, W : set of all programs,
 $IPC_{base}[]$: baseline IPC for each program,
 $Cluster_0$: workloads to be throttled,
 $Cluster_1$: workloads to be protected

Output: WS' : maximum average weighted speedup

```

1: foreach  $w_i \in W$  do
2:   Measure current  $BW_i$  and  $IPC_i$  for  $w_i$ 
3:    $BW[i] = BW_i$ ,  $IPC[i] = IPC_i$ 
4: end for
5:  $WS = \frac{1}{N} \sum_{i=0}^N \frac{IPC[i]}{IPC_{base}[i]}$ 
6: /* bound-aware throttling */
7: foreach  $w_i \in Cluster_0$  do
8:    $throttle\_value = 40\%$ 
9:   repeat
10:     $Throttle(w_i, throttle\_value)$ 
11:    foreach  $w_i \in W$  do
12:      Measure current  $BW_i$  and  $IPC_i$  for  $w_i$ 
13:       $BW'[i] = BW_i$ ,  $IPC'[i] = IPC_i$ 
14:    end for
15:     $WS' = \frac{1}{N} \sum_{i=0}^N \frac{IPC'[i]}{IPC_{base}[i]}$ 
16:     $progress = WS' - WS$ ,  $WS = WS'$ 
17:     $throttle\_value = binary\_search(throttle\_value, progress)$ 
18:  until  $throttle\_value \in \{10\%, 30\%, 50\%, 90\%\}$ 
19: end for
20:  $Retreat()$  //Retreat to the best throttle_value if needed

```

The details of the bandwidth throttling algorithm are shown in pseudocode Algorithm 1.

First of all, we need to determine the target of throttling. A straightforward method is to throttle programs one by one in reverse order of their bandwidth occupancy in the case of free competition.

When the target of throttling determined, we need to figure out other programs' bandwidth upper bounds. A simple method is to gradually throttle the target program from 90% to 10% at interval of 10%. In each step, we measure current system performance and compare it with the previous one. We will backtrack to the previous step when performance degradation occurs. The advantage of sequential search is that it does not miss any potential optimal solution, while the downside is that each target program needs to try all 8 possible throttling parameters, which consumes a lot of time to search. For programs with high phase switching frequency, spending too much time searching for potential solution

would miss a lot of optimization opportunities (More details about phase detection will be discussed in Section 4.3). Besides, when the performance difference between two adjacent parameters is small, the performance comparison may mislead the optimization direction due to measurement error. An alternative method is binary search (Figure 3), which ensures that each target program finds near-optimal solution within three steps.

There are some tradeoffs in our algorithm. We select 5 groups of application mixes (Table 1) for sensitivity study. We make sure that their bandwidth demands are as wide as possible to represent different scenarios. The bandwidth demand is derived from their aggregated bandwidth in solo-run case divided by the total bandwidth of the system. It can be seen from Figure 4a that binary search has a significant performance improvement over sequential search. Compared with sequential search, the performance improvement of binary search is increased by 7.07% on average and up to 14.08% in the case of w5.

Table 1: Application mixes used in sensitivity study

ID	Application Mixes	B/W(%)
w1	gcc_r, cactu_r, lbm_r $\times 2$, omnet_r $\times 2$, xz_r $\times 2$	74%
w2	perl_r, bwaves_r, lbm_r $\times 2$, omnet_r, xz_r $\times 2$	83%
w3	perl_r, mcf_r, cactu_r, lbm_r, omnet_r, fotonik3d_r, roms_r, xz_r	104%
w4	cactu_r $\times 2$, lbm_r, omnet_r, fotonik3d_r $\times 2$, xz_r	121%
w5	perl_r, cactu_r, lbm_r, omnet_r, fotonik3d_r $\times 3$, xz_r	155%

(a) Sequential search vs Binary search

(b) No-clustering vs Clustering

Figure 4: Performance improvement of different designs compared with baseline

Binary search performs better than sequential search for two main reasons. (1) When the programs with high memory intensity and low intensity corun, it is necessary to set an aggressive throttling value for a high intensity program, generally at least 30%. In this case, the binary search can be faster to find the optimal solution. (2) When multiple intensive programs exist at the same time, it is very inefficient to throttle a single program as it yields little impact. The extra bandwidth generated by throttling is often not enough to improve other memory-intensive programs and the overall performance of the system cannot be significantly improved. For instance, in w5, the most intensive program lbm is the first to be throttled, but if we set throttling value as 90%, the performance of another memory intensive program, i.e., fotonik3d is to improve, while other programs with lower intensity cannot benefit from it.

This phenomenon is also reflected in the cases we will talk about later. Although binary search can solve this problem with a larger throttling value difference in two adjacent explorations, it is still not an efficient solution. We use clustering to further optimize it.

4.2 Clustering

Clustering algorithms have been applied into cache allocation in some previous work [8, 29] in order to classify the workloads efficiently. In this work we adopt divisive hierarchical clustering [18] to classify the workloads based on their real-time memory intensity without any prior knowledge.

Why clustering? Since the direction of optimization is monotonous, it seems that we can get the maximum *WS* by limiting programs' bandwidth one by one in the order of their *BE* as we have done above. Why do we need clustering? There are two main reasons. The first is that when multiple programs with high and low memory intensity run simultaneously, if only the program with the highest memory intensity is throttled, the extra bandwidth brought by competition mitigation will be mostly occupied by other programs with high memory intensity rather than the lowest ones. In this case, the overall performance of the system may not be obviously improved, which will even mislead the direction of our optimization due to measurement error. Secondly, if we just throttle one program at a time, we need to try many times to find the best throttling scheme for all programs. Since it takes a certain amount of time to wait for the hardware to enforce a throttling command, the whole scheduling process will take too long to adapt to phase changes of workloads when their phases switch frequently.

Here we give a simple example. We run four instances of 519.lbm_r and four 520.omnetpp_r simultaneously on 8 different cores. The 8 instances should obviously be divided into two clusters, and the four instances of 519.lbm_r, due to their higher memory intensity, should be throttled according to our performance model. As shown in Figure 5, p_i represents the instance on the i -th core. s_j represents the non-clustering operation that limits the most memory intensive core's bandwidth to $j\%$. ck represents the clustering operation that limits the bandwidth of all cores in the memory intensive cluster to $k\%$ simultaneously. Compared with baseline where no restrictions are imposed, $s30$, $s20$ and $s10$ improve system weighted speedup by 1.69%, 3.13% and 5.49%, respectively, while $c30$, $c20$ and $c10$ improve system performance by 14.19%, 31.25% and 41.93%, respectively. When memory bandwidth of the most memory intensive $p0$ is throttled, we expect to see speedup from $p4$ to $p7$. However, other memory intensive workloads, i.e. $p1$ to $p3$, will occupy a large amount of resources if they are not throttled, leading to low performance improvement. Since the optimization direction in each step of our heuristic algorithm is based on the performance improvement of the previous step and measurement error of hardware performance counters on a real machine may misjudge a slight performance improvement as a degradation, so the method of non-clustering not only reduces the efficiency of each search step but also misleads the search direction for an optimal solution.

A standard divisive hierarchical clustering starts by classifying all the samples into one class and then gradually divide them into smaller units until each sample is a self-contained class [18]. When

Figure 5: A case study about the difference between clustering and no-clustering

(a) Proportion of different cluster numbers (b) Performance improvement numbers

Figure 6: The proportion of different cluster numbers during the entire scheduling process and their real-time performance improvement

the distance is less than a threshold, this division process will terminate.

In this work, we classify programs according to their memory intensity. For each classification, we divide all the programs to be throttled into two clusters with relatively higher or lower memory intensity. We collectively throttle the memory bandwidth of higher cluster and determine whether the system performance can be improved. If true, we retain the previous throttling scheme and continue to divide the unthrottled lower-intensity cluster into two clusters. Otherwise, we stop the process. We will repeat this process until there is no more performance improvement. As shown in Algorithm 2, we combine clustering and bound-aware throttling to dynamically schedule memory bandwidth in a heuristic way. Figure 4b shows that the raw bound-aware throttling with binary search can improve system performance by 28.47% on average, and the clustering algorithm improves it by another 6.57%.

Intuitively, the more layers our hierarchical clustering method has, the more performance gain system can achieve, but it is not true. Figure 6a shows that, throughout the entire scheduling process, EMBA achieves the best system performance with only one clustering layer in 74% of cases and two layers in 22% of cases. Figure 6b shows that more clusters do not bring more performance improvement. On average, the performance improvement of 3 clusters is the largest, reaching 52.63% while 2 clusters and more than three clusters increase by 47.42% and 48.30%, respectively. (Note that performance gains here refer to the real-time system performance gain when EMBA makes decisions, i.e., ws' in line 16 of Algorithm 1 subtracted by 1. They are not the same as the overall system performance across the whole execution as reported in Figure 4b.) An insight is that bandwidth pollution mainly comes

Algorithm 2 Heuristic_Hierarchical_Clustering

Input: $W = \{w_1, w_2, \dots, w_N\}$: workload of N programs,
 N : the number of cores/programs

Output: WS : maximum average weighted speedup

```

1:  $W' = W$ 
2: foreach  $w_i \in W$  do
3:   Measure current  $BW_i$  and  $IPC_i$  for  $w_i$ 
4:    $BW_{base}[i] = BW_i$ ,  $IPC_{base}[i] = IPC_i$ 
5: end for
6:  $Cluster_0 = \Phi$ ,  $Cluster_1 = \Phi$ ,  $WS = 1$ ,  $WS' = 1$ 
7: repeat
8:    $WS = WS'$ 
9:   /* initial cluster centers */
10:   $c_0, c'_0 = \max_{w_i \in W} (BW_{base}[i])$ 
11:   $c_1, c'_1 = \min_{w_i \in W} (BW_{base}[i])$ 
12:  /* start clustering */
13:  repeat
14:     $c_0 = c'_0$ ,  $c_1 = c'_1$ 
15:    foreach  $w_i \in W$  do
16:      if  $BW_{base}[i] > (c_0 + c_1)/2$  then
17:         $Cluster_0 = Cluster_0 \cup \{w_i\}$ 
18:      else
19:         $Cluster_1 = Cluster_1 \cup \{w_i\}$ 
20:      end if
21:    end for
22:     $c'_0 = \frac{1}{|Cluster_0|} \sum_{w_i \in Cluster_0} BW_{base}[i]$ 
23:     $c'_1 = \frac{1}{|Cluster_1|} \sum_{w_i \in Cluster_1} BW_{base}[i]$ 
24:    /* stop when centers unchanged */
25:  until  $c'_0 == c_0 \ \&\& \ c'_1 == c_1$ 
26:   $WS' = \text{Bound\_aware\_Throttling}(Cluster_0, Cluster_1,$ 
     $N, W', IPC_{base}[])$ 
27:   $progress = WS' - WS$ 
28:   $W = W - Cluster_0$ 
29: until  $progress \leq 0$ 
30: return  $WS$ 

```

from the programs with the highest memory intensity. We can improve system performance by throttling them while programs with medium memory intensity do not need to be restricted. Actually, in the subsequent experiments, we find that these medium intensive programs are the main source of system performance improvement.

4.3 Phase Detection Frequency vs Throttling Accuracy

A program's memory access pattern usually changes in different phases, which requires us to spend as little time as possible in a scheduling process to match the phase change of programs. However, it usually takes a certain amount of time to guarantee the accuracy of a scheduling system. Thus, there's a tradeoff between its efficiency and accuracy.

In each step of our heuristic algorithm we decide the next clustering and throttling strategy based on the feedback of the previous step. Thus, we must wait for the throttling command of the previous step to take effect and collect performance information under the new MBA scheme. The time interval from calling MBA API

Figure 7: Sensitivity studies about some parameters in phase detection

to the target throttling scheme actually taking effect is about 6.2 milliseconds on average. The time can be completely covered by calling API itself, which takes 12 ms in our implementation. When the throttling command takes effect, we still need to spend a certain amount of time on refreshing the performance counters to record current *memory bandwidth* and *IPC*. In our implementation it takes 5 ms to refresh the counters. In order to find the optimal solution, the worst-case search needs to traverse all possible throttling schemes. In the most extreme case, each program is divided into a self-contained cluster and each cluster exhausts all 7 legal throttling schemes, which will take $17 * 7 * 7 = 833ms$ to find the optimal solution. In the case where the phases of programs change frequently, it is likely that a heuristic exploration has not ended yet, but the phase of a program has changed.

Fortunately, the optimization methods mentioned in Section 4.1 and 4.2 greatly reduce our scheduling duration, which allows us to study and select a wider range of parameters for EMBA. Figure 7a and 7b show our sensitivity study about the threshold and interval of phase detection. During a period of execution, we consider a workload's behavior changes greatly if more than one program's IPC change beyond a certain range compared with the previous stage. We try several potential parameters and choose the best 50% and 1 second as our final IPC threshold and scheduling interval.

4.4 Putting All Together

EMBA is an efficient memory bandwidth throttling approach. It doesn't need any prior knowledge about workloads and thus avoids expensive profiling step. All we need is the real-time memory bandwidth occupied by each core, which can be collected via hardware performance counter with extremely low overhead. We implement EMBA in userspace and use an interface [2] provided by Intel to collect *IPC*, *bandwidth* and some other information we need. EMBA runs as a userspace daemon, monitoring the real-time *IPC* and bandwidth occupancy of each core every 1 second. When the *IPC* variation of at least one core exceeds 50%, a subsequent clustering and throttling process will be activated. EMBA will return to the monitoring state when the optimal MBA scheme is found and enforced. Figure 8 shows the workflow of EMBA.

5 EVALUATION

5.1 Experimental Setup

We implement and evaluate our EMBA on an Intel Xeon Silver 4110 machine with 8 cores and 11MB LLC, whose specifications are

Figure 8: Workflow of EMBA

summarized in Table 2. In order to verify the impact of different memory bandwidth resources on performance, we configure memory with 1 and 2 memory channels, which provide total bandwidth of 15.8GB/s and 31.6GB/s, respectively.

Table 2: Specification of the experimental system

CPU	8 Intel Xeon Silver 4110 cores @ 2.1GHz
L1 Cache	32KB per core, split D/I, 8-way set-associative
L2 Cache	1MB per core, 8-way set-associative
L3 Cache	11MB, 11-way set-associative, non-inclusive
Memory	(1) 32GB (1× 32GB DIMMs), DDR4 2133MT/s (2) 64GB (2× 32GB DIMMs), DDR4 2133MT/s
OS	CentOS 7, Linux kernel version 4.12.1

5.2 Benchmarks

Figure 9: Bandwidth demand of selected workloads

We use SPEC2017rate [3] as benchmarks to evaluate EMBA. We borrow the classification method of a previous study [7] to distinguish benchmarks’ memory intensity. We allocate 1 LLC way to each program in case of solo-run and measure their LLC Misses per 1K Instructions (MPKI) during a complete execution process. We refer to a benchmark as high-intensive if its MPKI is greater than ten, as medium-intensive if the MPKI value is greater than one but less than ten, as low-intensive if the MPKI value is less than one. Some benchmarks’ MPKIs are very low and their memory accesses are negligible, such as 548.exchange, 511.povray and 538.imagick. We don’t choose them for our workload mixes. The reason why we classify programs based on MPKI rather than their bandwidth occupancy is that a program’s bandwidth occupancy is significantly affected by hardware configuration such as the number of memory channels. Table 3 depicts each benchmark’s IPC, average bandwidth and MPKI. According to the MPKI metric, 520.omnetpp and

Table 3: Workloads’ characteristic and classification

Bench	IPC	Average B/W (MB/s)	MPKI	Memory Intensity
505.mcf	0.69	3393	39.04	High (H)
519.lbm	1.75	7434	33.17	
549.fotonik3d	1.73	7271	34.03	
510.parest	1.55	3272	17.41	
503.bwaves	2.32	2475	15.26	
520.omnetpp	0.40	704	15.03	
523.xalancbmk	1.07	997	14.94	
554.roms	1.93	3531	14.89	
507.cactuBSSN	0.76	867	8.78	Medium (M)
502.gcc	0.78	662	8.05	
557.xz	0.46	331	6.51	
521.wrf	1.11	755	5.23	
527.cam4	1.44	719	4.00	
526.blender	1.13	303	2.30	
525.x264	2.66	244	0.67	Low (L)
500.perlbench	1.58	188	0.93	
508.namd	2.53	184	0.59	
544.nab	1.37	94	0.57	
541.leela	1.22	62	0.32	
531.deepsjeng	1.09	60	0.43	
538.imagick	3.17	22	0.11	
511.povray	2.37	0.27	0.001	
548.exchange2	2.50	0.002	$3e^{-6}$	

523.xalancbmk are classified into the high-intensive category. However, in the terms of solo-run bandwidth demand, they behave more closely to medium-intensive programs which occupies bandwidth arranging from 300 MB/s to 1000 MB/s.

In order to comprehensively analyze system performance and the application scenarios of our method, we select programs with different memory intensity in different proportions to construct 100 8-program workloads, so that the workloads can cover as many bandwidth pressure situations as possible. We randomly select 4, 5, 6, 7, and 8 programs from the high memory intensity programs respectively, and then randomly select the remaining from the medium and low intensity programs to build 8 program workloads, i.e. 4H, 5H, 6H, 7H, 8H categories. Each category contains 20 different application mixes. Figure 9 shows the bandwidth demand of each selected workload. In all of our experiments, we bind each program to a dedicated core and repeatedly run it for 20 minutes.

5.3 Overhead

Table 4 reports the overhead of our method for workload w1 (see Table 1). We measure the overhead of EMBA on an 8-programs workload co-running in the 1-channel and 2-channel configurations, respectively. *free* indicates the program execution time when no bandwidth restrictions are imposed. MBA_{off} represents the execution time of programs when EMBA is turned on but MBA unenforced. MBA_{on} indicates the execution time of the program when EMBA is fully turned on. On average, EMBA introduces 0.10% and 0.41% overhead for w1 in 1 channel and 2 channels, respectively. Our experiments show that the overhead for other workloads is similarly low, which is 0.07% in 1 channel and 0.18% in 2 channels on average.

Table 4: Overhead analysis of EMBA for w1

Bench	2 channels				1 channel			
	free(s)	MBA _{off} (s)	overhead(%)	MBA _{on} (s)	free(s)	MBA _{off} (s)	overhead(%)	MBA _{on} (s)
lbm×2	364	364	0	835	622	623	0.16	1103
xz×2	299	299	0	223	509	509	0	282
omnet×2	1120	1135	1.34	783	2124	2129	0.24	1185
gcc	91	91	0	75	130	130	0	107
cactu	619	619	0	491	943	943	0	693
avg	539	543	0.41	539	954	955	0.10	749

5.4 Performance and Bandwidth Utilization

Metric of performance. We evaluate system performance using Weighted Speedup (WS) [8, 9, 30]:

$$\text{WeightedSpeedup} = \sum \frac{IPC_i}{IPC_{Base_i}} / \#program \quad (10)$$

The speedup of program i is defined as IPC_i / IPC_{Base_i} , where IPC_{Base_i} represents the IPC of program i when it co-executes with other programs without any resource control, and IPC_i is its IPC in co-run situation.

Figure 10: Performance improvement and memory bandwidth utilization compared with baseline

We evaluate all 100 workloads in the memory configurations of 1 channel and 2 channels, respectively. Figure 10a shows the average performance improvement of EMBA compared with baseline on different workload categories. In the 1 channel configuration, EMBA improves system performance by up to 73.7% and by 36.9% on average. In the 2-channel configuration, EMBA improves system performance by up to 37.7% and 10.1% on average. Since the CPU has only 8 physical cores, the bandwidth in the 2-channel environment is unsaturated, so are some cases in the 1-channel environment. Figure 9 shows that few workloads have their aggregated bandwidth demands reach the theoretical maximum bandwidth in the 2-channel environment.

Specifically, performance improvement of 4H is the lowest, which is 28% for 1 channel and 4% for 2 channels, respectively. Since the bandwidth pressure of the system is unsaturated, the benefit from bandwidth throttling is limited. The bandwidth of 5H is slightly higher than 4H, and its average performance improvement is 31% for 1 channel and 5% for 2 channels. 6H gets better improvement with 1 channel increased by 42% and 2 channels 10%, respectively. 7H is the most improved group of all categories, whose performance increases by 51% in 1 channel and 18% in 2 channels. In the case of 8H, the bandwidth pressure reached the maximum, but their performance improvement are slightly reduced compared with 7H, with 1 channel increased by 33% and 2 channels 13%.

Figure 10b shows the bandwidth utilization rate of different application mixes. In the 1-channel configuration, EMBA reduces memory bandwidth utilization rate by 8.61% compared to baseline on average and up to 11.97%. In the 2-channel configuration, the reduction in bandwidth utilization increases to 14.66% on average and up to 15.79%. The smaller the bandwidth pressure is, the greater the bandwidth utilization loss will be.

Figure 11: Memory bandwidth demand and performance gain of each application mix

Figure 11 shows the bandwidth demand and performance gain of each application mix. When the aggregated bandwidth demand of an application mix is between 0 and 100%, their average performance gain is 31.34%. When the demand is between 100% and 150%, the average performance improvement is 41.30%. When the demand is between 150% and 200%, their performance is improved by 29.68% on average. EMBA can efficiently improve system performance under medium bandwidth pressure. When the pressure is too high or too low, EMBA does not function that well.

5.4.1 Case Study. We take one workload each from the 4H, 5H, 6H, 7H and 8H mixes as examples to break down the performance gain.

Figure 12 depicts IPC and average bandwidth occupancy of individual programs with EMBA on or off. We classified programs according to their MPKI, thus 520.omnetpp and 523.xalancbmk were considered as high-intensive category. However, the solo-run bandwidth demand suggests that they behave more closely to medium-intensive programs, which occupies bandwidth ranging from 300 MB/s to 1000 MB/s. Thus, we consider these two as medium-intensive programs in this case study.

In all cases, the programs with low memory intensity are barely affected in terms of bandwidth occupancy and performance, the programs with high memory intensity are throttled to benefit medium memory intensive programs, resulting in overall performance improvement. In the 4H case, EMBA improves overall performance by 19.76%, the medium memory intensive programs contributes

Figure 12: Case study of different workload mix scenarios

most of the performance gain, specifically, 520.omnetpp_r is improved by 109.43%, 557.xz_r 75.66%, 507.cactuBSSN_r 51.66%, and

526.blender_r by 19.39%. On the other hand, the high memory intensive programs suffer from significant performance degradation, which ranges from 40.99% to 61.91%. The low memory intensive programs shows little to no performance change.

In the 5H case, overall performance is improved by 26.25%. Specifically, the most significant performance degradation of individual programs is 52.76%, coming from the high-intensive 519.lbm_r while the most significant performance improvement is 155.66% from the medium-intensive 520.omnetpp_r.

In the remaining cases, the same phenomenon is observed regardless of the system bandwidth utilization that high-intensive programs experience performance degradation while medium-intensive programs are significantly improved and the IPC change of a program is proportional to the change in its memory bandwidth. As a result, EMBA improves the overall system performance.

Figure 13 shows the average speedup of every benchmark across all the multi-programmed workloads. The programs with high bandwidth demand experience significant performance degradation, notably a 38.1% reduction in lbm, a decrease of 18.57% in fotonik3d, and a 9% reduction in bwaves. The programs with medium bandwidth demand are improved significantly. For instance, xalancbmk is improved by 98%, omnetpp 126%, and xz 98%. At last, the programs with low bandwidth show no significant changes.

Insights. We have three interesting observations: (1) When the bandwidth pressure of a system is neither too high nor too low, EMBA can bring the best optimization. (2) EMBA essentially suppresses the memory bandwidth utilization of programs with high memory intensity, and encourages the programs with medium memory intensity. (3) The optimization maximizes the system performance without increasing the demand on the bandwidth utilization, furthermore, it slightly lowers the utilization as demonstrated by Figure 10.

5.5 Discussion

EMBA could cause significant performance loss for an individual program while improving the overall performance of an application mix, the loss might be unacceptable in specific scenario. A possible solution extends EMBA to be QoS-aware, such that the performance of any individual program is guaranteed to be not less than certain percentage of the baseline, for example, when the performance is measured in bandwidth, 20% QoS means the bandwidth reduction of any individual program in an application mix should not exceed 20% of its bandwidth in baseline. A similar QoS metric was used by Ye et al. [39] for cache partition.

We implement QoS guarantee by introducing constraints to each step of throttling of Algorithm 1. Figure 14 depicts the QoS-aware EMBA optimization of the 7H case in Figure 12, the IPC slowdown of 519.lbm_r, which experiences the largest performance loss with original EMBA, is reduced from 46.86% to 19.73%, while the overall performance improvement is reduced from 32.95% to 18.20%. We leave the fine-grained QoS guarantee to future work.

6 RELATED WORK

Resource allocation has been well studied in the past, most of them focus on computation resource, including CPU and GPU [6, 19, 23, 40], some of the rest focus on storage resources, including cache or

Figure 13: Average speedup of each benchmark in all of the co-running application mixes

Figure 14: Case study about QoS guarantee

memory [4, 10, 11]. Previous studies in allocating memory bandwidth either are conducted using simulation [20] or rely on a specifically designed kernel scheduler [41]. To the best of our knowledge, no prior work has improved performance via memory bandwidth allocation on commercial hardware.

MemGuard [41] relies on a specifically designed kernel scheduler, it allows each application to be associated with a memory bandwidth reservation, then maximizes the overall performance by allocating the spare bandwidth after satisfying each application's reservation. Merlin [34] manages memory bandwidth by migrating pages between NUMA nodes. Lee et al. [20] design a SDRAM controller specific to multimedia platform SOCs. A most recent scheme proposed by Jinsu et al. [26] allocates memory and LLC for fairness.

One problem of hardware-based resource allocation is that the coarse granularity could limit the potential of performance improvement. KPart [8] sidesteps the limitations of Intel CAT by grouping applications based on cache-sensitivity and then sharing a partition across the programs within a group. Brock [5] reports that each partition sharing could be equivalent to a partition namely natural cache partition. Other approaches co-locate applications without allocating the bandwidth [22, 37, 38].

Our allocation strategy is based on the performance model of memory bandwidth, which has been well studied both in simulation [21, 31, 32] and in a real machine [26]. All of them result in a similar conclusion, i.e., the performance is linearly correlated to the memory bandwidth. Fang et al. [21] report the interaction of cache partitioning and bandwidth partitioning, and conclude that the optimal bandwidth partitioning can be derived theoretically.

They conduct the experiments in a simulator and simulates only 250 millions instructions, while the benchmarks we are using execute 2.5 trillions instructions on average. DraMon [35] uses LLC miss ratio and a set of hardware parameters to predict the memory bandwidth, they report a runtime prediction overhead of 0.5% on average.

7 CONCLUSION

In this work, we explore the relationship between program's performance and its LLC occupancy and memory request rate on commodity processors and formulate a quantitative model to guide the application of Intel Cache Allocation Technology (CAT) and MBA. With the performance model, we propose a heuristic bound-aware throttling algorithm to dynamically explore the upper bounds of programs' memory bandwidth without any prior knowledge. We further introduce a hierarchical clustering method to improve the efficiency of our throttling algorithm. Finally, we implement these algorithms in EMBA, a low-overhead dynamic memory bandwidth scheduling system to improve performance on Intel commodity processors. On average, we improve system performance by 36.9% at the expense of 8.6% bandwidth utilization rate.

Our model indicates that cache partitioning and bandwidth throttling simultaneously affect system performance. We leave LLC and memory bandwidth co-scheduling as our future work.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Leonel Sousa, and anonymous reviewers for their constructive comments and suggestions. We also thank Dong Tong and Yehong Zhong for fruitful discussions. The research is supported in part by the National Key R&D Program of China under Grant No. 2018YFB1003505, National Science Foundation of China (No. 61472008, 61672053 and U1611461), Shenzhen Key Research Project No: JCYJ20170412150946024, National Science Foundation CSR-1618384 and Beijing Municipal Science and Technology Program under Grant Z181100008918015.

REFERENCES

- [1] Intel 64 and ia-32 architectures software developer's manual. <https://software.intel.com/>.
- [2] Intel-cmt-cat. <https://github.com/intel/intel-cmt-cat/tree/master/pqos/>.
- [3] Spec cpu 2017. <https://www.spec.org/cpu2017/>.
- [4] BERGER, D. S., BERG, B., ZHU, T., SEN, S., AND HARCHOL-BALTER, M. Robinhood: Tail latency aware caching—dynamic reallocation from cache-rich to cache-poor.

- In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 195–212.
- [5] BROCK, J., YE, C., DING, C., LI, Y., WANG, X., AND LUO, Y. Optimal cache partition-sharing. In *2015 44th International Conference on Parallel Processing* (2015), IEEE, pp. 749–758.
 - [6] CHERKASOVA, L., GUPTA, D., AND VAHDAT, A. Comparison of the three cpu schedulers in xen. *SIGMETRICS Performance Evaluation Review* 35, 2 (2007), 42–51.
 - [7] EBRAHIMI, E., LEE, C. J., MUTLU, O., AND PATT, Y. N. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010* (2010), pp. 335–346.
 - [8] EL-SAYED, N., MUKKARA, A., TSAI, P., KASTURE, H., MA, X., AND SÁNCHEZ, D. Kpart: A hybrid cache partitioning-sharing technique for commodity multicore. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018* (2018), pp. 104–117.
 - [9] EYERMAN, S., AND ECKHOUT, L. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3 (2008), 42–53.
 - [10] FUNARO, L., BEN-YEHUDA, O. A., AND SCHUSTER, A. Ginseng: Market-driven {LLC} allocation. In *2016 {USENIX} Annual Technical Conference ({USENIX}) {ATC} 16* (2016), pp. 295–308.
 - [11] HAMEED, A., KHOSHKBARFOROUSHHA, A., RANJAN, R., JAYARAMAN, P. P., KOLODZIEJ, J., BALAJI, P., ZEADALLY, S., MALLUHI, Q. M., TZIRITAS, N., VISHNU, A., ET AL. A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems. *Computing* 98, 7 (2016), 751–774.
 - [12] HERDRICH, A., ILLIKAL, R., IYER, R. R., NEWELL, D., CHADHA, V., AND MOSES, J. Rate-based qos techniques for cache/memory in CMP platforms. In *Proceedings of the 23rd international conference on Supercomputing, 2009, Yorktown Heights, NY, USA, June 8-12, 2009* (2009), pp. 479–488.
 - [13] HOWER, D. R., CAIN, H. W., AND WALDSPURGER, C. A. PABST: proportionally allocated bandwidth at the source and target. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017* (2017), pp. 505–516.
 - [14] HU, X., WANG, X., LI, Y., LUO, Y., DING, C., AND WANG, Z. Optimal symbiosis and fair scheduling in shared cache. *IEEE Trans. Parallel Distrib. Syst.* 28, 4 (2017), 1134–1148.
 - [15] ILLIKAL, R., CHADHA, V., HERDRICH, A., IYER, R., AND NEWELL, D. PIRATE: qos and performance management in CMP architectures. *SIGMETRICS Performance Evaluation Review* 37, 4 (2010), 3–10.
 - [16] IPEK, E., MUTLU, O., MARTÍNEZ, J. F., AND CARUANA, R. Self-optimizing memory controllers: A reinforcement learning approach. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China* (2008), pp. 39–50.
 - [17] JAHRE, M., AND NATVIG, L. A light-weight fairness mechanism for chip multiprocessor memory systems. In *Proceedings of the 6th Conference on Computing Frontiers, 2009, Ischia, Italy, May 18-20, 2009* (2009), pp. 1–10.
 - [18] KAUFMAN, L., AND ROUSSEUW, P. J. *Finding groups in data: an introduction to cluster analysis*, vol. 344. John Wiley & Sons, 2009.
 - [19] LE, T. N., SUN, X., CHOWDHURY, M., AND LIU, Z. AlloX: Allocation across computing resources for hybrid cpu/gpu clusters. *ACM SIGMETRICS Performance Evaluation Review* 46, 2 (2019), 87–88.
 - [20] LEE, K.-B., LIN, T.-C., AND JEN, C.-W. An efficient quality-aware memory controller for multimedia platform soc. *IEEE transactions on circuits and systems for video technology* 15, 5 (2005), 620–633.
 - [21] LIU, F., JIANG, X., AND SOLIHIN, Y. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India* (2010), pp. 1–12.
 - [22] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015* (2015), pp. 450–462.
 - [23] LU, Q., YAO, J., QI, Z., HE, B., ET AL. Fairness-efficiency allocation of cpu-gpu heterogeneous resources. *IEEE Transactions on Services Computing* (2016).
 - [24] MUTLU, O., AND MOSCIBRODA, T. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA* (2007), pp. 146–160.
 - [25] NESBIT, K. J., AGGARWAL, N., LAUDON, J., AND SMITH, J. E. Fair queuing memory systems. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA* (2006), pp. 208–222.
 - [26] PARK, J., PARK, S., AND BAEK, W. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), ACM, p. 10.
 - [27] PARK, J., PARK, S., HAN, M., HYUN, J., AND BAEK, W. Hypart: a hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus, November 01-04, 2018* (2018), pp. 5:1–5:14.
 - [28] RAFIQUE, N., LIM, W., AND THOTTETHODI, M. Effective management of DRAM bandwidth in multicore processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), Brasov, Romania, September 15-19, 2007* (2007), pp. 245–258.
 - [29] SELFA, V., SAHUQUILLO, J., ECKHOUT, L., PETIT, S., AND GÓMEZ, M. E. Application clustering policies to address system fairness with intel’s cache allocation technology. In *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017* (2017), pp. 194–205.
 - [30] SNAVELY, A., AND TULLSEN, D. M. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000* (2000), pp. 234–244.
 - [31] SUBRAMANIAN, L., SESHADRI, V., GHOSH, A., KHAN, S. M., AND MUTLU, O. The application slowdown model: quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015* (2015), pp. 62–75.
 - [32] SUBRAMANIAN, L., SESHADRI, V., KIM, Y., JAIYEN, B., AND MUTLU, O. MISE: providing performance predictability and improving fairness in shared main memory systems. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013* (2013), pp. 639–650.
 - [33] TAM, D. K., AZIMI, R., SOARES, L., AND STUMM, M. Rapidmrc: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009* (2009), pp. 121–132.
 - [34] TEMBEY, P., GAVRILOVSKA, A., AND SCHWAN, K. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–14.
 - [35] WANG, W., DEY, T., DAVIDSON, J. W., AND SOFFA, M. L. Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (2014), IEEE, pp. 380–391.
 - [36] WANG, X., CHEN, S., SETTER, J., AND MARTÍNEZ, J. F. SWAP: effective fine-grain management of shared last-level caches with minimum hardware support. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017* (2017), pp. 121–132.
 - [37] XIANG, Y., WANG, X., HUANG, Z., WANG, Z., LUO, Y., AND WANG, Z. DCAPS: dynamic cache allocation with partial sharing. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018* (2018), pp. 13:1–13:15.
 - [38] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (New York, NY, USA, 2013), ISCA ’13, ACM*, pp. 607–618.
 - [39] YE, C., BROCK, J., DING, C., AND JIN, H. Rochester elastic cache utility (recu): Unequal cache sharing is good economics. *Int. J. Parallel Program.* 45, 1 (Feb. 2017), 30–44.
 - [40] YU, Y., ZHANG, J., AND LETAIEF, K. B. Joint subcarrier and cpu time allocation for mobile edge computing. In *2016 IEEE Global Communications Conference (GLOBECOM)* (2016), IEEE, pp. 1–6.
 - [41] YUN, H., YAO, G., PELLIZZONI, R., CACCAMO, M., AND SHA, L. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2013), IEEE, pp. 55–64.
 - [42] ZHAO, J., FENG, X., CUI, H., YAN, Y., XUE, J., AND YANG, W. An empirical model for predicting cross-core performance interference on multicore processors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (Sep. 2013), pp. 201–212.
 - [43] ZHOU, Y., AND WENTZLAFF, D. MITTS: memory inter-arrival time traffic shaping. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016* (2016), pp. 532–544.