Enforcing Last-level Cache Partitioning through Memory Virtual Channels

Jongwook Chung
Seoul National University
Seoul, Republic of Korea
jwchung@scale.snu.ac.kr

Yuhwan Ro
Samsung Electronics
Suwon, Republic of Korea
yuhwan.ro@samsung.com

Joonsung Kim
Seoul National University
Seoul, Republic of Korea
joonsung90@snu.ac.kr

Jaehyung Ahn
Samsung Electronics
Suwon, Republic of Korea
jh91.ahn@samsung.com

Jangwoo Kim
Seoul National University
Seoul, Republic of Korea
jangwoo@snu.ac.kr

John Kim
KAIST
Daejeon, Republic of Korea
jjk12@kaist.edu

Jae W. Lee
Seoul National University
Seoul, Republic of Korea
jaewlee@snu.ac.kr

Jung Ho Ahn
Seoul National University
Seoul, Republic of Korea
gajh@snu.ac.kr

***Abstract*—Ensuring fairness or providing isolation between multiple workloads with different characteristics that are co-located on a single, shared-memory system is a challenge. Recent multicore processors provide last-level cache (LLC) hardware partitioning to provide hardware support for isolation, with the cache partitioning often specified by the user. While more LLC capacity often results in higher performance, in this work we identify that a workload allocated *more* LLC capacity result in *worse* performance on real-machine experiments, which we refer to as MiW (more is worse). Through various controlled experiments, we identify that another workload with less LLC capacity causes more frequent LLC misses. The workload stresses the main-memory system shared by both workloads and degrades the performance of the former workload even if the LLC partitioning is used (a balloon effect).**

To resolve this problem, we propose virtualizing the datapath of main-memory controllers and dedicating the *memory virtual channels* (mVCs) to each group of applications, grouped for LLC partitioning. mVC can further fine-tune the performance of groups by differentiating buffer sizes among mVCs. It can reduce the total system cost by executing latency-critical and throughput-oriented workloads together on shared machines, of which performance criteria can be achieved only on dedicated machines if mVCs are not supported. Experiments on a simulated chip multiprocessor show that our proposals effectively eliminate the MiW phenomenon, hence providing additional opportunities for workload consolidation in a datacenter. Our case study demonstrates potential savings of machine count by 21.8% with mVC, which would otherwise violate a service level objective (SLO).

***Keywords*-Memory Virtual Channel, LLC Partitioning, Fairness, More is Worse**

I. Introduction

Modern chip multiprocessors (CMPs) consist of multiple cores sharing various resources, including shared last level cache (LLC), on-chip interconnect, and main memory [6], [32], [49]. CMPs are currently the most popular design choice for servers used in cloud environments, and such CMP-based servers consistently run a number of heterogeneous applications to satisfy the needs of diverse users. This trend is becoming more prevalent with the emergence of virtual machines and containers for cloud services.

When applications run simultaneously, contention and interference of shared resources in a system can cause performance degradation for some or all of the applications [9], [18], [30], [39], [47], [48], [49]. As a result, there has been a significant amount of prior work done to provide fairness and minimize interference from sharing the on-chip LLC capacity and main-memory bandwidth [13], [39], [47], [48], [49]. In particular, when multiple applications compete for a limited capacity of shared cache, high-priority applications that need quality-of-service (QoS) guaranteed (or real-time applications) can suffer from performance degradation due to excessive cache occupancy from other applications [6], [18], [32], [42]. To ensure the performance guarantee for QoS or real-time applications, modern CMPs provide cache partitioning (CP) [3], [8], [15] where different portions of LLC are allocated to different applications. Cache partitioning can allocate an isolated cache region to high-priority applications, which avoids contention and interference by preventing concurrently running applications from evicting high-priority application cache lines [15]. Many prior studies [47], [48] have investigated alternative CP to maximize overall performance. However, recently, CMPs [15] provide *user-specified* CP, and the previously proposed CP algorithms are not necessarily applicable. In this work, we propose a mechanism to enforce performance isolation in user-specified LLC partitioning.

When a CMP dedicates more LLC capacity to a process group through cache partitioning, the intuitive expectation is that performance improves [15]. However, we demonstrate that the *opposite* can occur as a process group can actually perform *worse* when it obtains more LLC capacity. We refer to this as **more-is-worse (MiW)** phenomenon and define MiW *degree* as the ratio of IPC when maximum LLC is allocated to a process group to the maximum IPC that can be obtained through CP. Our evaluations show that MiW degree can reach up to 39.5% with synthetic workloads, 14.4%

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00016

for SPEC CPU2006 [45], and 547.0% for TailBench [20] benchmarks, respectively, on Intel Broadwell-based [23] Xeon systems.

In this paper, we first provide an analysis of why this MiW phenomenon occurs. When a particular process (e.g., process A) receives more LLC capacity, another process in the system (e.g., process B) comes to receive a smaller fraction of the LLC capacity and experiences higher LLC Miss Per Kilo Instructions (MPKI). This increases main-memory bandwidth demand from process B (a *balloon effect*[1]) and results in higher main-memory access latency for all the processes. Even though the memory access patterns for process A and process B are different (i.e., accessing different banks or ranks), both processes share the same datapath to the main-memory system, including memory request buffers. As a result, requests from process B can monopolize the shared datapath resource in the memory system. This effectively results in process B "blocking" process A's requests and degrades the performance of process A.

To prevent this blocking in the datapath to the main memory, we propose to *virtualize* the datapath of memory controllers with **memory virtual channel** (mVC) where a separate memory request buffer is provided for each group of LLC. The overall memory request buffer storage is partitioned across the number of groups supported in the LLC, which is equivalent to the number of mVCs. DRAM commands from different buffers (or mVCs) are arbitrated and served independently – thus, each mVC has effectively a private datapath to the memory channel and avoids blocking. The memory controller requires a *mVC arbiter* that is responsible for arbitrating between the mVCs – the mVC that receives a grant from the mVC arbiter gains access to the memory channel. The grant is released only after a column-level DRAM (RD/WR) command is issued to avoid unnecessary DRAM row-buffer conflicts.

We discuss mVCs with four different buffer allocation policies, which are *static*, *proportional*, *inversely-proportional* (both based on its share of LLC ways), and *dynamic* partition. The observations show that static and proportional partitions are more effective in eliminating MiW than the other. Furthermore, we explore the design space by observing the performance of mVC on various ratios of buffer allocation. As a result, we show that it is possible to select an appropriate configuration satisfying the target performance for the group with more LLC capacity, and also maximizing the performance of the group with less LLC capacity. Our case study shows that when satisfying 90% of the standalone performance, with mVCs we can save 21.8% of machines by sharing the machines among applications in a distributed system.

In summary, this paper makes the following contributions:

- This is one of the first work to demonstrate the problem of MiW (more-is-worse) on a real machine, where allocating more LLC capacity to a workload leads to worse performance due to an increased degree of congestion (blocking) on the main memory shared by all the workloads (a balloon effect).
- We propose to virtualize the datapath of memory controllers to mitigate this blocking problem and explore the design space of memory request buffer allocation.
- We evaluate memory virtual channels (mVC) using a cycle-level simulator, which effectively eliminates MiW and recovers lost IPC due to the blocking.
- We perform a case study to demonstrate mVC can provide additional opportunities for workload consolidation to save the machine count by up to 21.8%, which would otherwise violate a service level objective (SLO).

II. Background: Cache Partitioning

To overcome the contention and interference on the shared resources, CMPs provide cache partitioning/allocation techniques [3], [15]. A cache partitioning (CP) divides shared LLC resource and dedicates each partitioned LLC to a group (class) of processes. CP allows the cache to be adequately allocated according to the working set size or cache sensitivity of a process group, alleviating contention and interference between processes [39], [47]. For example, AMD provides CP in Opteron [3], [8], and Intel introduced Cache Allocation Technology (CAT) starting from Haswell architecture [15].

CP techniques can be classified as way, set, or block (line) based partitioning [1], [7], [32], [33], [41], [42], [53]. *Way-based partitioning* [7] divides LLC by cache ways. Processes can replace the cache line only within the allocated cache ways. Way-based partitioning is relatively cheap to implement because the process can access all the cache sets regardless of the number of allocated ways. However, it is limited to the maximum number of ways in granularity, and the associativity of each partition can be greatly reduced depending on the allocated ways [42]. *Set-based partitioning* [1], [33] (or page coloring [53]) partitions LLC by sets instead of ways, and each process gets a number of sets from the cache. LLC is virtually divided so that the address of a requested data is mapped to a set in the virtual cache. The virtual set index is then mapped to the actual physical cache set index. This translation makes set-based partitioning more expensive than way-based partitioning, especially when resizing the partition. For finer-grained partitioning, *block-based partitioning* was also proposed to partition the cache by cache block (line) granularity [41] and provide more cache partitions. However, the complexity and overhead for managing and storing the mapping information identifying the owner of each cache line are high [32].

AMD Opteron [3] implements set-based cache partitioning. To minimize the amount of LLC data being evicted by

[1]We use the terminology balloon effect since changes in one area (i.e., cache partitioning) leads to an adverse effect in another area (i.e., memory bandwidth).

Table I: Hardware setup used in Section III.

Hardware Info	Settings
CPU Model	Intel Xeon E5-2698 v4
CPU Clock, # of cores	2.2GHz, 20
# of memory controllers per CPU	2
Per core:	
L1 I/D $ type/size/associativity	Private/32KB/8
L2 $ type/size/associativity	Private/256KB/8
L3 $ type/size/associativity	Shared/2.5MB/20
# of Hardware threads	2
Hardware prefetch	Off
Per 32GB DDR4-2400 memory controller:	
# of channels	2
(# of ranks, bandwidth) per channel	(2, 19.2GB/s)

a core that does not allocate the data, the Opteron processor can direct L2 victim traffic to a specified set of the LLC. However, the unit of partitioning is a quarter of the total LLC capacity, which is too coarse-grain. By contrast, Intel CAT [15] adopts way-based CP for the shared LLC. With CAT, each class of service (CLOS) consists of one or more applications. A bitmask (each bit representing a single cache way) is used to determine the amount of LLC allocation for each CLOS, and the bitmask can be changed dynamically at runtime. CLOS can be allocated exclusively (isolated mode), or allocated to overlap with other CLOS (overlapped mode). CAT has been supported since Haswell microarchitecture with 4 CLOS; more recent Broadwell and Skylake-based servers support up to 16 CLOS.

Herdrich et al. [15] demonstrated the performance improvement of up to 4.5× from CAT when running SPEC CPU2006 applications together as CAT significantly alleviated the performance degradation of an application from interference. With CP (e.g., CAT), more LLC capacity can be dedicated to a certain application to prioritize and improve its performance effectively. However, contrary to this intuitive expectation, we observed that *a group of processes could actually perform worse when they receive more LLC capacity.*

III. More-is-Worse Phenomenon

We first demonstrate and analyze how *the performance of a process group decreases as we allocate more LLC capacity with cache partitioning* on real machines. To the best of our knowledge, this un-intuitive phenomenon has not been reported on real machines.[2]

A. More LLC Leading to Performance Drop

We evaluated a system with a single socket Intel Xeon Broadwell server with 20 cores (40 hardware threads with HyperThreading), 50MB of shared LLC, and 76.8GB/s of peak main-memory bandwidth. The Intel machine has CAT (Cache Allocation Technology) for cache partitioning (CP). Details of the experimental setup are described in Table I. Our initial evaluation uses SPEC CPU2006 benchmarks [45] and executed SPEC rate of N, running N instances (processes) of a benchmark simultaneously. We bundled the cores that execute the same benchmark into one CLOS (class of service).

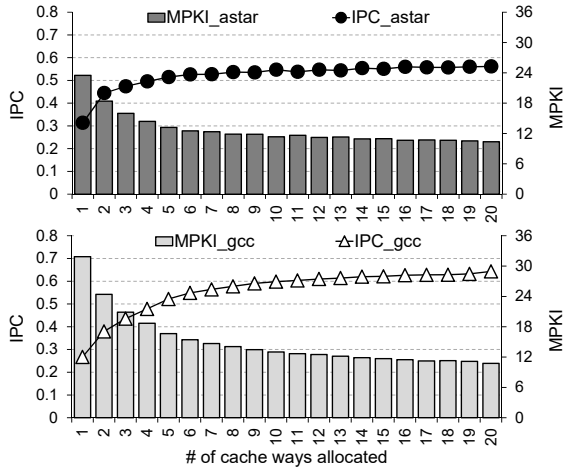
Figure 1(a) shows the IPC and LLC Misses Per Kilo Instructions (MPKI) variation as the number of allocated LLC ways increases when executing 473.astar and 403.gcc benchmarks alone with rate 20. Each core runs two instances, and thus, we use 10 out of the 20 cores. The evaluated Intel processor has 20 LLC ways per cache set, and thus, we swept the LLC ways from one to 20. The presented IPC is the mean IPCs from all the cores running the same application. The results are intuitive – as more LLC is allocated, MPKI decreases and performance (IPC) monotonically increases. Initially, more LLC results in a significant decrease in MPKI and correspondingly a significant performance improvement but afterward, the change in MPKI is limited as performance saturates [39].

We then executed the two benchmarks together with each running on 10 physical cores and each with a rate of 20. We dedicated varying numbers of LLC ways to the two application groups; N to one and $(20 - N)$ to the other. Figure 1(b) shows the normalized IPC and LLC MPKI when executing 473.astar and 403.gcc together, with the IPC and MPKI values of the two applications without CP in the rightmost column. Using CP improves the aggregate performance of the two application groups sharing the LLC. When we allocate nine LLC ways to 473.astar (11 for 403.gcc), its performance is the same as (2.7% better than) that without CP, showing CP is effective.

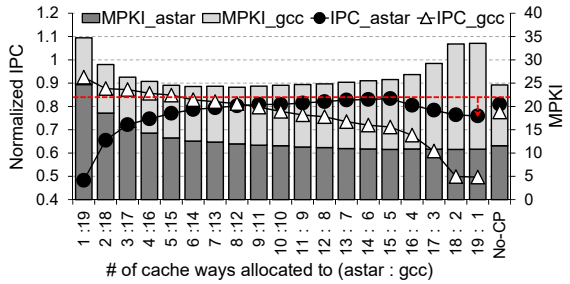
The expected behavior is a trade-off between LLC capacity and performance. As more LLC capacity is allocated to a workload, the performance is expected to continue to increase or saturate. However, our evaluation shows that performance can be actually *degraded* with more LLC capacity. For example, for 473.astar, performance first increases as LLC capacity increases, but beyond 15 LLC ways, the performance drops by up to 8.9%. This is seemingly counter-intuitive as the performance of both 403.gcc and 473.astar are degraded when 473.astar occupies more than 15 LLC ways. We call this **MiW (more-is-worse)** phenomenon.

In addition to 403.gcc and 473.astar, similar behaviors were also observed in other SPEC CPU2006 and SPEC CPU2017 [46] benchmarks. The degree of MiW, the ratio of IPC when maximum LLC is allocated to a process group to the maximum IPC that can be obtained through CP, for some of the SPEC benchmarks are summarized in Table II(a). We observe up to 14.4% performance degradation when the former benchmark of the pair occupies more LLC capacity over a certain threshold, respectively. Note that MiW does not happen always. For example, on the pair of

[2]We used the isolated mode because the overlapped mode can cause unnecessary contention between the benchmarks on LLC, making the analysis more complicated.



(a) IPC, MPKI of 473.astar and 403.gcc when executed alone.



(b) IPC, MPKI of 473.astar and 403.gcc when executed together.

Figure 1: IPC and LLC MPKI for 473.astar and 403.gcc (a) when executed alone as LLC capacity is increased and (b) two benchmarks are executed together, with the IPC normalized to when each runs alone and occupies the entire LLC capacity (20 ways). *IPC of 473.astar decreases by up to 8.9% after reaching the peak when it is allocated with 15 LLC ways.*

473.astar-473.astar, the performance of both groups increase monotonically as more LLC ways are allocated.

B. Synthetic Workload Evaluation

In this section, we evaluate the MiW phenomenon using synthetic workloads to better control workload's memory access characteristics and analyze performance degradation when allocating more LLC capacity. We use a pointer chasing synthetic workload, whose performance is sensitive to memory latency because of true dependency between each memory access. We controlled the degree of memory bandwidth pressure by varying the amount of data read per step of pointer chasing.

Without loss of generality, we call a group (class) of applications that are allocated more LLC capacity and expects higher performance 'group-A', and the other group that receives the remaining LLC capacity 'group-B'. To differentiate the characteristics of workload group-A and group-B, we set group-A to read only one cache line (64B) per pointer chasing step over 1GB of working set, which is 20× larger than the LLC capacity. Thus, group-A is less sensitive to changes in LLC capacity but more sensitive to changes in main-memory access latency. group-B reads 1KB of data per pointer chasing step over 5MB of working set, which is only one-tenth of the system's LLC capacity, to generate frequent LLC misses when smaller LLC capacity is allocated. We read 1KB of data per step to generate more bandwidth pressure to memory compared to group-A. We evaluated with the same system described earlier in Table I, except only a single memory channel instead of four channels is used to stress main-memory bandwidth.

Table II: The degrees of MiW (more-is-worse) over pairs of applications (App A/B) which divide up LLC. The MiW degree is measured by comparing (a) the aggregated IPC, and (b) the tail (95th percentile) latency of App A when it occupies the maximum share of LLC (numerator) with the one when it performs best over all possible LLC shares (denominator) through CP. For latency-critical workloads, 403.gcc has been used for App B.

(a) SPEC benchmarks

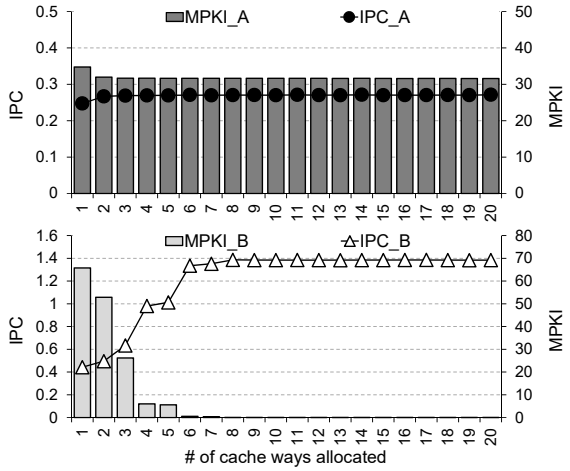
App A	App B	MiW
omnetpp	gcc	14.40%
astar	gcc	8.94%
sphinx	gcc	8.43%
gcc	gcc	6.01%
xz	xalancbmk	5.27%
mcf	blender	3.22%

(b) TailBench benchmarks

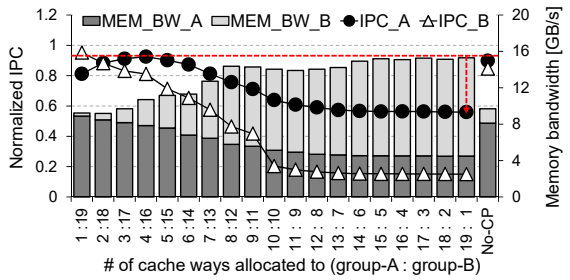
App A	MiW
moses	547.00%
masstree	142.83%
img-dnn	10.20%
specjbb	9.00%
xapian	8.51%
silo	8.39%

Figure 2(a) shows the IPC and LLC MPKI as the number of LLC ways allocated to group-A and group-B is varied. For group-A that uses 1GB of memory and much larger than LLC capacity, its performance is mostly insensitive to the change in the allocated LLC capacity, and the memory bandwidth usage is maintained at a constant level of 1.8GB/s. By contrast, group-B uses only 5MB of memory and allocating a large amount of LLC capacity leads to negligible LLC misses. However, when the allocated LLC capacity is small, there are LLC misses and memory access rates increase rapidly. Therefore, the IPC decreases by 68% and the memory bandwidth usage increases to 5.8GB/s.

The result when both group-A and group-B are executed is shown in Figure 2(b). When allocating more LLC capacity to group-A, we expect performance to increase or reach a steady-state, but performance actually decreases when 5 (25% of LLC capacity) or more LLC ways are allocated to group-A, reproducing MiW observed with SPEC benchmarks. Since group-A and group-B alone cannot fully utilize the system memory bandwidth, we executed group-A and group-B with rate four. The performance degradation (MiW) gets worse as more instances of the group-A and group-B are populated. The synthetic evaluations demonstrate that



(a) IPC, LLC MPKI of the synthetic workload when group-A and group-B run alone.



(b) IPC, memory bandwidth of group-A and group-B execute together.

Figure 2: Synthetic workload evaluation when (a) executing alone and (b) group-A and group-B executing together, with IPC normalized to when each workload runs alone with 20 LLC ways allocated. The IPC of group-A drops up to 39.5% after reaching the peak when it occupies 4 LLC ways.

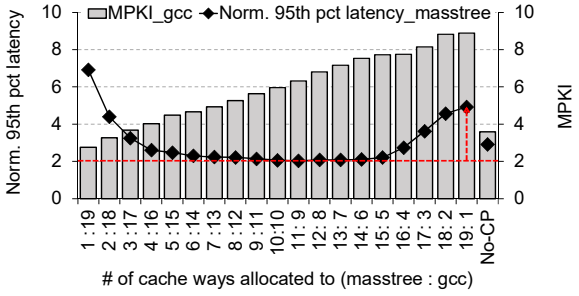


Figure 3: 95th percentile latency of masstree when executed with 403.gcc, where normalized to the tail latency executed alone occupying the entire LLC capacity.

MiW can be reproduced with a simple, synthetic workload but more interestingly, MiW can start even if an application group occupies only a smaller portion of the shared LLC resource.

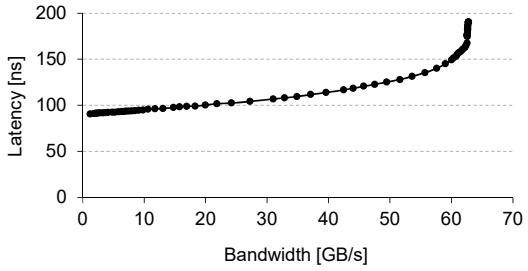


Figure 4: Load-latency values of the tested system (Table I) with 76.8GB/s of max main-memory bandwidth. Latency rises rapidly when system bandwidth gets closer to the peak.

C. Impact on Latency-critical Workloads

In addition to the SPEC benchmarks, we evaluate the impact of MiW on latency-critical (LC) workloads. In particular, it is well-known that LC applications, especially in datacenters, often require predictable and small tail latency [5], [10], [29]. However, as shown in Section III-A, MiW increases MPKI – thus, higher memory access latencies can significantly impact the tail latency problem [21]. Therefore, MiW can be even more critical for LC workloads.

To evaluate the impact of MiW on LC workloads, we used TailBench [20], [21] and executed each TailBench benchmark together with 403.gcc from SPEC CPU2006. Similar to the previous evaluations, we vary the number of LLC ways for the two benchmarks, but for the TailBench benchmarks, performance is measured in terms of tail latency. We used the single-node integrated configuration of TailBench, where a client and the corresponding LC application are integrated into a single process.

Figure 3 shows the normalized 95th percentile latency of masstree, where normalized to the tail latency when runs alone occupying the entire LLC. The result shows that the tail latency increases by up to 143%, as it occupies more LLC ways. Table II(b) summarizes the degree of MiW of other TailBench benchmarks. Moses and masstree have significantly higher MiW degrees compared to other benchmarks (as high as 547% with moses), due to higher LLC MPKI from these workloads, and thus, results in longer queuing time. Due to space constraints, additional results are not shown, but similar trends were observed in evaluation of Intel Skylake machines, with tail latency increasing by up to 210% due to MiW.

D. The Root Cause of the MiW Phenomenon

To understand the root cause of MiW, we first pay attention to the fact that MiW occurs when applications stress the main-memory bandwidth of a system. Figure 4 shows the relationship between the bandwidth load and the observed latency of a main-memory system with the peak bandwidth of 76.8GB/s specified in Table I. Main-memory access latency values increase slowly when the memory

system is lightly loaded, but they increase rapidly as the load gets closer to the theoretical peak bandwidth, similar to interconnection networks [9]. When a larger portion of LLC capacity is allocated to the synthetic workload group-A in Figure 2, the other workload group-B receives smaller LLC capacity, experiences higher LLC MPKI, stresses main-memory bandwidth that is shared between group-A and group-B, and hence increases memory access latency for both group-A and group-B. In other words, *when group-B stresses the main-memory bandwidth due to fewer LLC ways allocated, group-A also experiences high memory access latency breaking the performance isolation between the workload groups, which is the very intention of CP*. Therefore, the group with more LLC capacity (group-A) has higher memory access time for memory requests that miss LLC; this overhead can even outweigh the benefits of lower LLC MPKI due to larger LLC capacity, resulting at performance drop especially if group-A is highly sensitive to main-memory latency.

It might appear as if memory requests from different applications are heading to the same destination (a memory channel) and hence these requests cannot be isolated, leading to a surge in access latency values on all the requests; but in reality, they are likely headed to different destinations. When the requests from both processes access the same target in main memory (e.g., the same DRAM bank), they all should experience high loaded access latency due to the elevated degree of queuing delay. However, different processes mostly access different targets (e.g., different DRAM banks) as modern CMPs typically have dozens of DRAM banks per channel; so the chances that two requests from different processes access the same bank are meager.[3]

Then, the reason why a surge in LLC MPKI of one process (group-B) negatively affects the performance of the other (group-A) could be due to blocking of the datapath that a request handling an LLC miss experiences, a well-known problem in designing the flow control of interconnection networks when requests from different source-destination pairs share the same intermediate datapath (e.g., buffers) [9]. This blocking occurs when the oldest packet in an intermediate shared buffer cannot be transferred because the next node on the route for its destination is congested, the "younger" packets in the shared buffer are blocked, resulting in a performance drop. A solution for this blocking is to virtualize the datapath, such as virtual channels [9].

Moreover, requests from one process (group-B) can occupy a significant portion or even all of the shared intermediate datapath (memory request buffer), which is a valuable/scarce resource. This limits the memory controller's visibility of the processes (group-A and -B) with different access behaviors, and lead to a poor scheduling decision.

[3]Techniques to partition main-memory such that a bank is dedicated to a process (e.g., PALLOC [52] and [28]), can be used to ensure banks are not shared between processes.

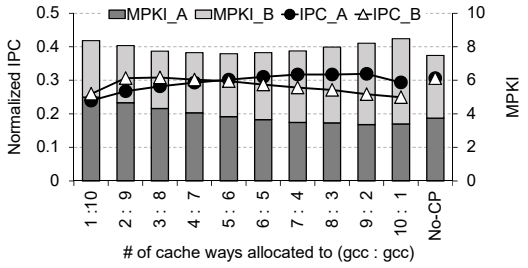


Figure 5: The impact of Memory Bandwidth Allocation (MBA) on 403.gcc-403.gcc. MBA cannot eliminate MiW.

Virtualizing the datapath can help to solve this problem.

We first show that *existing hardware does not have virtualized datapath in memory controllers.* We control main-memory bandwidth demands from two groups of processes (group-C and -D) such that group-C alone spends half the peak bandwidth of a system, and group-D alone spends the entire bandwidth. When we run group-C and group-D together, we observed that group-C burns $1/3$ of main-memory bandwidth, whereas group-D uses the other $2/3$. If the memory requests from group-C and group-D are through virtualized datapath, as group-C and group-D both have the same priority level, they should both utilize $1/2$ of main-memory bandwidth.

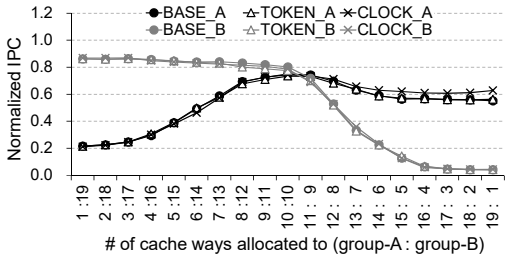
IV. Limitations of Existing Solutions

Before exploring the idea of virtualizing the datapath of memory controllers, we first assess if the ideas that are already implemented in hardware (main-memory bandwidth throttling [17]) or have been extensively studied before (memory scheduling considering fairness [27], [36]) can address MiW. Through experiments with the latest HW and simulation, we observe these existing solutions cannot eliminate MiW.

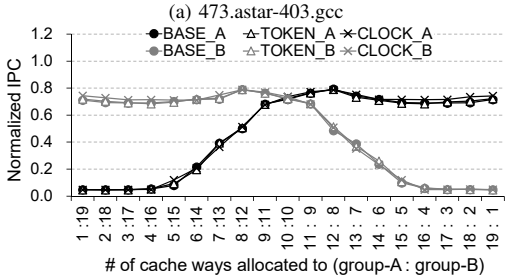
A. Memory Bandwidth Throttling

The latest Skylake-based [11] Xeon systems support a feature named Memory Bandwidth Allocation (MBA) [17], which limits the memory bandwidth dedicated to each group (class). We evaluated a system with a single socket Skylake server with 24 physical cores (HyperThreading enabled), 33MB of shared LLC with 11 ways, and 21.3GB/s of peak main-memory bandwidth. MBA limits memory bandwidth by the granularity of 10% (we used the linear mode [17]).

Figure 5 shows the normalized IPC and stacked-up MPKI values of a pair of 403.gcc and 403.gcc, similar to the experiments in Section III-A except that MBA is enabled here. We allocated 90% of bandwidth allocation (the higher, the more bandwidth allocated) to group-A and 10% to group-B. The result shows MiW is still observed for the machine with MBA. We tested different bandwidth allocation ratios (e.g., 10%/90% and 50%/50% to group-A/-B), but MiW persists.



(a) 473.astar-403.gcc



(b) 403.gcc-403.gcc

Figure 6: Simulation results of augmenting the default memory access scheduling (FR-FCFS [40], BASE) with token-bucket (TOKEN [27]) and virtual clock (CLOCK [36]) algorithms. These fairness-aware memory scheduling algorithms do not resolve MiW.

However, if we change the configurations such that main-memory is not bandwidth saturated by either decreasing SPEC rate, increasing peak main-memory bandwidth by populating more channels (Skylake supports up to 6 channels per socket), or lowering the bandwidth allocation values of MBA to all the application groups, MiW mostly disappears. *This also indicates that the blocking in congested memory controllers is a likely source of MiW.*

The memory bandwidth throttling looks like a plausible solution, but *MBA has a limitation in that it controls memory bandwidth indirectly and approximately* [17]. MBA places a programmable rate controller in L2 MSHR, a boundary between private L2 caches and share LLC. This enables per-core rate control (source throttling) without introducing virtualized datapath. However, as L2 misses are then filtered through LLC (whose miss rates are hard to predict as it is shared among many cores), this indirect bandwidth control is inevitably approximate. Therefore, MBA must conservatively limit memory bandwidth to prevent the blocking (over-throttling), and hence the performance of all the application groups would be sub-optimal due to this main-memory bandwidth underutilization.

B. Fairness-aware Memory Scheduling

Among the proposals of providing fairness on top of memory access scheduling (the control part of a memory controller), we selected two representative ones and tested if they can address MiW. First, we chose the token bucket algorithm (TOKEN), which was originally introduced as an arbitration method for interconnection networks [27], [38], [54]. For TOKEN, each request can be processed when it has a matching token in the respective bucket (each for the corresponding group). A token generator distributes tokens to different buckets at the rates proportional to the fractions allocated to different groups. Second, a request prioritization method, which gives priority based on a virtual clock (CLOCK) [36], is a memory version of deadline-based arbitration frequently adopted in interconnection networks. CLOCK prioritizes 1) ready commands, 2) column-level commands, and 3) commands with the earliest virtual finish-time. The virtual finish-times of the DRAM commands from each memory request are calculated based on prior work [36]. To prevent priority inversion by bank priority chaining, after a DRAM bank has been restored in the course of row activation (around 32ns in modern DRAM devices), rule 3) is applied first over rule 2) among the requests heading to the same bank. We set both TOKEN and CLOCK to treat all the application groups equally.

Because these schemes are not implemented in existing hardware, we used simulation, whose setup is detailed in Section VI. Two benchmark pairs from SPEC CPU2006 [45] are used (see Figure 6). Both TOKEN and CLOCK perform on par with or better than the baseline memory-access-scheduling scheme of FR-FCFS (BASE in Figure 6), but MiW persists. When two application groups are executed, TOKEN keeps each group from using more than half of the system's peak memory bandwidth. Therefore, TOKEN restricts a group's memory bandwidth only when it requires more than half of the system's peak memory bandwidth, allowing both groups to utilize memory bandwidth more fairly. CLOCK prioritizes a request with the earliest deadline (finish-time) and hence tries to divide the system's memory bandwidth equally for each group. However, because neither TOKEN nor CLOCK eliminates the blocking problem when the main-memory system is bandwidth saturated, MiW does not disappear.

These and other recent memory access scheduling proposals [4], [47], [48] pursue fairness in scheduling by limiting the number of consecutive requests to a specific DRAM bank, by limiting the number of reordering a request can experience to serve other requests with a higher priority, and by rotating the priority among the requests originating from different sources (e.g., cores). However, these proposals prioritize requests within a buffer; if a certain request cannot enter the memory request buffer (as the buffer are full due to blocking, for example), the scheduler cannot address the problem, and the system suffers from MiW.

V. Virtualizing Memory Channels

A. Memory Virtual Channel (mVC)

To prevent/alleviate the blocking in main-memory systems, we propose to virtualize the datapath of memory controllers (MCs) by providing a separate request buffer

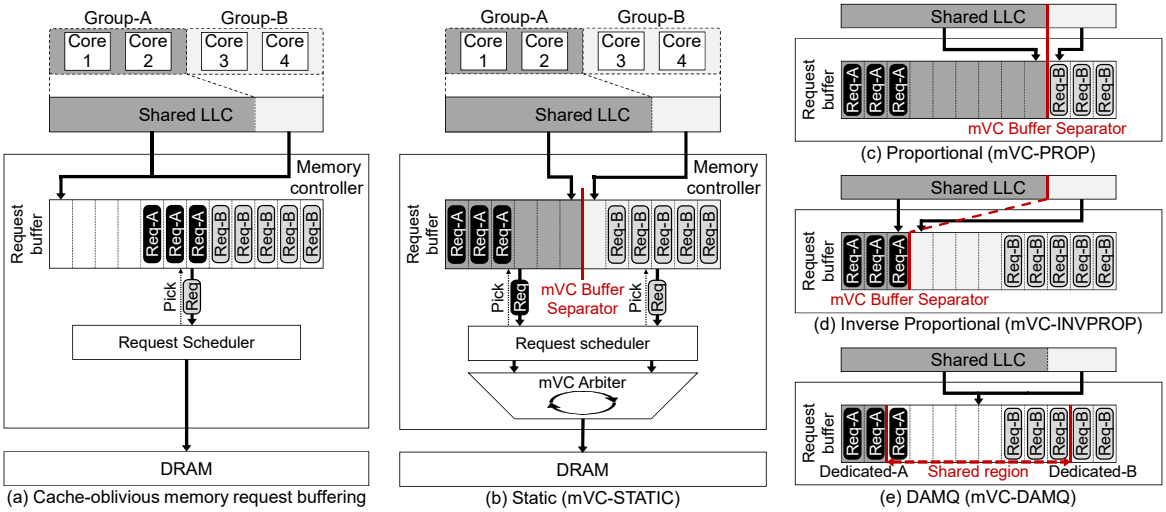


Figure 7: Comparison of the conventional memory request buffering (a) and four buffer allocation strategies for mVC (b)-(e).

per group (class) of LLC. As opposed to the previous works utilizing per-source (CPU vs. GPU) [4] or per-thread [36] request buffers, we use per-group (per-class) separate buffering called *memory virtual channel (mVC)*. Per-source separate buffering is too coarse-grain as it does not separate requests from different cores within CPU or GPU, and per-thread separation is too expensive as the number of hardware threads in modern shared-memory chip multiprocessors can exceed a few hundred. We assume that NoC (network-on-chip) is not a source of blocking[4]; if so, it should be virtualized as well or support other blocking prevention feature (e.g., bufferless flow control [34]).

Similar to Intel MBA, we align the class of MC and that of LLC; therefore, a group (class) of applications have both dedicated LLC capacity and MC's buffer (queue) space. As opposed to the source throttling of Intel MBA, which cannot prevent blocking in MCs because it does not precisely know the amount of traffic filtering by LLC, mVC guarantees blocking prevention. All datapath within a MC must be virtualized. If a MC has multiple stages of buffers (e.g., staged memory scheduling [4]), they all should be virtualized (separated by groups). Otherwise, this un-virtualized portion of datapath becomes the source of blocking.

The control part of a MC (i.e., memory access schedulers) must be augmented to provide fairness among the groups/classes (see Figure 7(b)). For example, FR-FCFS [40] gives a higher priority to a ready request (which can be serviced with a RD or WR DRAM command without any timing constraint) over non-ready requests, on top of the baseline priority rule of first-come-first-serve. With the mVC support, there should be a round-robin arbitration logic between the classes, which should be the highest priority tier compared to both FR and FCFS.

A MC with mVCs requires a round-robin arbitration logic, which we refer to as *mVC arbiter*, that selects a DRAM command at a given cycle among the command candidates from different groups (classes). This round-robin arbiter responds with a single grant. Any buffer without an available DRAM command is simply skipped over and ignored by the mVC arbiter. *However, as opposed to NoC arbiters, an arbiter grant is not released after servicing a single DRAM command but held until a column-level (RD/WR) command is served.* This ensures that if two (or more) request buffers target the same DRAM banks, it avoids DRAM row-buffer conflicts by continuously issuing a sequence of ACT and PRE commands, avoiding deadlocks and providing fairness.

The multiple per-group request buffers do not necessarily increase the cost (in terms of storage) as the total amount of storage for the buffers are held constant; the only difference is the amount of storage *per* request buffer which can be smaller compared to the baseline request buffer. The additional cost for the mVC arbiter is also relatively small because the number of groups is usually much smaller than the aggregate number of entries in the request buffer.

B. mVC Buffer Allocation Strategies

One design question for mVC is how to allocate buffer space in the memory request buffers to the different mVCs. Figure 7 compares the conventional memory request buffering (Figure 7(a)) with the following four different buffer allocation strategies for mVC (Figure 7(b,c,d,e)):

- **Static (mVC-STATIC):** A simple strategy is to partition the request buffer *statically* in the same size among all the mVCs. While preventing starvation of either flow, this

[4]To the best of our knowledge, the NoC prior to Intel Skylake-based Xeon systems implements a ring NoC with prioritized arbitration and thus, blocking does not occur within the NoC itself.

scheme may lead to underutilization of request buffers when the memory request rate from the LLC is highly skewed between the two groups.

- **Proportional (mVC-PROP):** A second strategy is to allocate buffers to each group *in proportion to* its share of LLC ways. For example, if group-A and group-B are allocated 15 and 5 LLC ways, they receive 9 and 3 entries in the request buffer, respectively (see Figure 7(c)). The rationale of this strategy is to partition storage resources along the shared memory access path by the same ratio. It can alleviate MiW by preventing the group with fewer resources (say, group-B) from flooding the entire request buffer and slowing down the other group.
- **Inverse Proportional (mVC-INVPROP):** The next strategy is to allocate buffers to each group *inversely proportionally to* its share of LLC ways. In contrast to mVC-PROP, group-A and group-B receive 3 and 9 entries in the request buffer when 15 and 5 LLC ways are allocated to them, respectively (Figure 7(d)). Because groups that receive fewer LLC ways are likely to incur more LLC misses, this strategy tends to allocate more buffers to groups incurring LLC misses more frequently.
- **Dynamic (mVC-DAMQ):** We also consider a *dynamic* buffer allocation strategy based on DAMQ (dynamically allocated multi-queue) [9]. DAMQ partitions the request buffers dynamically among mVCs based on the request rate of each mVC. By partitioning the request buffer into shared and dedicated regions, a deadlock which would occur when a memory-intensive workload claims all of the buffer entries can be avoided. The shared region is dynamically allocated based on demands; the dedicated region is equally partitioned and dedicated to each mVC. A mVC first uses its dedicated region to store memory requests. Once its dedicated region is full, it claims an entry from the shared region for the next memory request (Figure 7(e)).

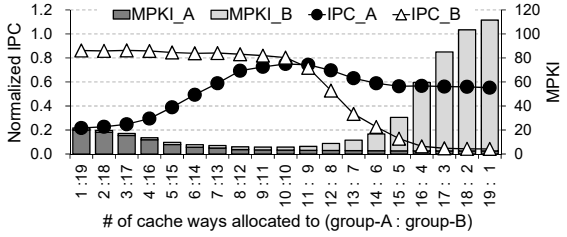
VI. Experimental Setup

We simulated a CMP system to evaluate the effectiveness of mVCs, whose parameters are summarized in Table III. McSimA+ [2] simulator was modified for the simulation. The baseline memory controller has a 20-entry request buffer and adopts FR-FCFS [40] as a memory request scheduling policy and adaptive open policy (which is also adopted at Intel Xeon processors) as a DRAM page management policy. SPEC CPU2006 [45] and SPEC CPU2017 [46] benchmark suites were used for evaluation. Simpoint [43] was used to extract the most representative simulation points of each application, each including 100 million instructions. We sorted and selected cache-sensitive applications in SPEC CPU2006 and SPEC CPU2017 benchmarks, and used them for evaluations.

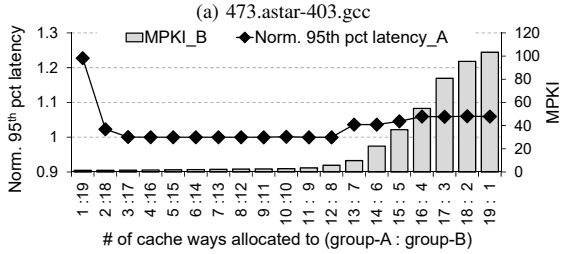
We compare four buffer allocation strategies for mVC: mVC-STATIC, mVC-PROP, mVC-INVPROP, and mVC-DAMQ. For static buffer allocation (mVC-STATIC), 10 entries are allocated to each mVC with two mVCs, which is equal to a total memory request buffer size of 20. For proportional buffer allocation (mVC-PROP), the number of buffer entries allocated to each mVC is based on the number of LLC ways allocated to each mVC. On the contrary, for inverse proportional buffer allocation (mVC-INVPROP), the number of buffer entries allocated to each mVC is (20 - the number of LLC ways allocated to each mVC). We also evaluated mVC with dynamic buffer allocation (mVC-DAMQ) based on 80% shared region size in the request buffer.

Table III: Parameters used in the simulated system.

Resource	Value
Number of cores, MCs	16 cores, 1 MC
Coherence policy	MOESI
Cache line size	64B
Per core:	
Frequency, issue/commit width	3.6GHz, 4/4 slots
Issue policy	Out-of-Order
L1 I/D $ type/size/associativity	Private/32KB/8
L2 $ type/size/associativity	Private/256KB/8
L3 $ type/size/associativity	Shared/40MB/20
Per DDR4-2400 memory controller (MC):	
# of channels, Request Q size	1, 20 entries
# of ranks, bandwidth per channel	2, 19.2GB/s
Scheduling policy	FR-FCFS [40]
DRAM page policy	Adaptive Open [16]



(a) 473.astar-403.gcc

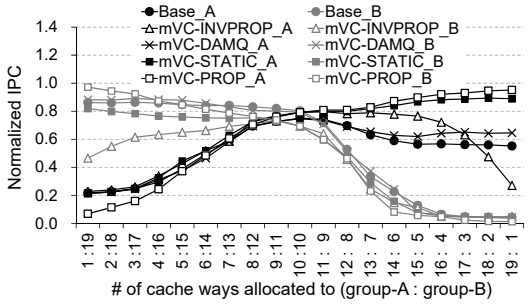


(b) xapian-403.gcc

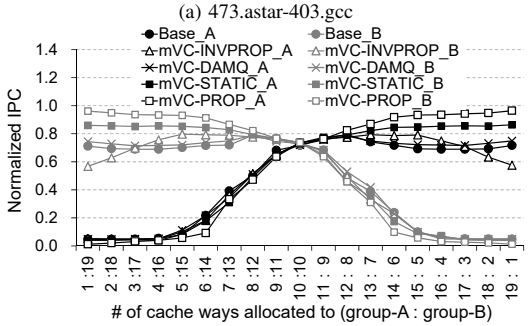
Figure 8: Simulation results on SPEC CPU2006 and TailBench showing trends similar to hardware experiments.

VII. Evaluation

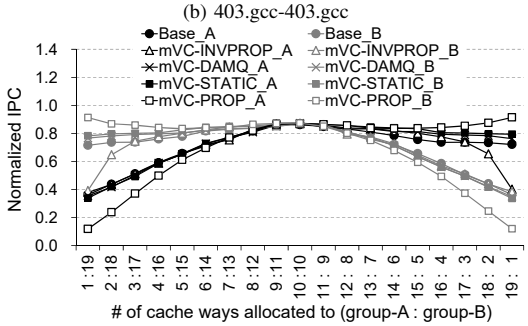
Before evaluating the proposed mVCs, we reproduced the hardware results through simulation (Figure 8). Xapian in Figure 8(b) is an application in TailBench, and the group-A consists of its single-threaded instance. The normalized



(a) 473.astar-403.gcc



(b) 403.gcc-403.gcc



(c) 523.xalancbmk-523.xalancbmk

Figure 9: Simulation results of mVC with different memory request buffer allocation policies: mVC-STATIC, mVC-PROP, mVC-INVPROP, and mVC-DAMQ. The normalized IPC is normalized based on those when each benchmark runs alone with 20 LLC ways allocated.

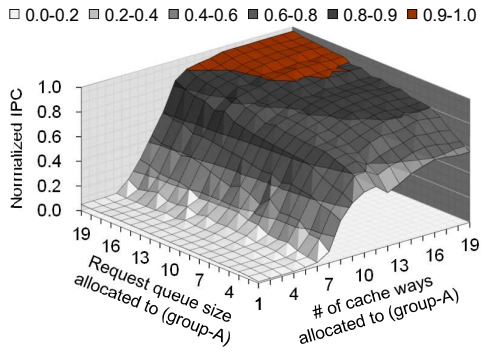
IPC and 95^{th} percentile latency are normalized based on those when each benchmark runs alone with 20 LLC ways allocated. Similar trends as the hardware results are observed (Figure 1(a,b) and Table II(b)) and clearly show MiW. The other case also matches with Table II(a).

Mitigating MiW through mVC. We evaluate the effectiveness of virtualizing the datapath of memory channels by executing multi-programmed workloads on the simulator. Figure 9 shows the IPCs of three workload pairs that demonstrate MiW in Section III-A (Table II), normalized to the IPCs with standalone execution. We compare four buffer allocation strategies for mVC in Section V-B: static (mVC-STATIC), proportional (mVC-PROP), inverse proportional (mVC-INVPROP), and dynamic (mVC-DAMQ). Because there are 16 cores, we executed each benchmark with a rate of 8.

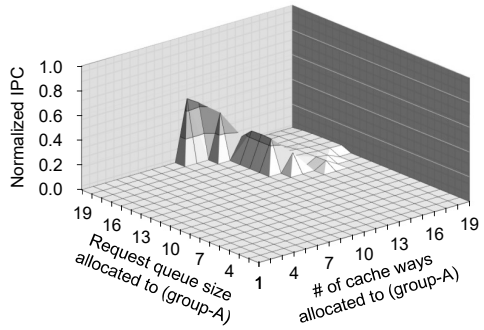
We made the following key observations. First, mVC effectively addresses the blocking problem except for mVC-INVPROP and mVC-DAMQ. As group-A gets allocated with more LLC ways in the baseline without mVC, the requests from group-B flood the request buffer to cause starvation of group-A. With mVCs, however, group-A has a guaranteed share of the request buffer entries and a fair chance for DRAM accesses via round-robin scheduling, alleviating the problem of blocking and eliminating MiW. For example, Figure 9(a) shows the results using a 473.astar-403.gcc pair. With mVC-PROP and mVC-STATIC, 473.astar achieves 95.2% and 86.4% of the IPC of standalone execution, respectively, while the baseline achieves only 75.0% without mVC due to MiW. MiW is also eliminated in Figure 9(b) and (c). By recovering lost IPC from MiW, this opens up additional opportunities for consolidating workloads requiring an IPC service-level objective (SLO) [30] with other best-effort workloads.

Second, mVC-PROP more effectively eliminates MiW than mVC-STATIC at the cost of penalizing the group with fewer resources, while mVC-DAMQ and mVC-INVPROP fail to eliminate MiW. In mVC-PROP, as group-A receives more LLC ways, more request buffer entries are allocated to it, yielding higher memory throughput due to a larger memory scheduling window. mVC-STATIC allocates memory requests fairly, which may increase system-wide throughput in some cases. Assuming an 80:20 division of the shared and dedicated regions, mVC-DAMQ performs slightly better than the baseline, but cannot eliminate MiW because group-B experiences a high LLC MPKI to flood the shared region of the request buffer, leading to starvation of group-A. If the dedicated region is expanded to alleviate this problem, mVC-DAMQ eventually behaves like static buffer allocation (mVC-STATIC) to lose the benefits of dynamic allocation. mVC-INVPROP allocates buffer entries in an opposite way of mVC-PROP. Therefore, in contrast to mVC-PROP, which eliminates MiW, mVC-INVPROP can deteriorate MiW by allocating fewer buffer entries to the group. This trend is clearly observed in our simulated cases.

Potentials for operating cost savings with mVC. mVC provides another knob to control resource allocation between two (or more) groups of applications. Figure 10 shows the results of a two-dimensional parameter sweeping for a 403.gcc-403.gcc pair, which demonstrates the greatest degree of MiW among the three SPEC CPU benchmark pairs we evaluate. X- and Y-axis represent the number of LLC ways allocated to group-A and the number of request buffer entries allocated to group-A, respectively. Figure 10(a) and (b) show the IPC normalized to standalone execution for group-A and group-B, respectively. As we run two copies of the same application, Figure 10(a) and (b) actually have the same shape but are oriented to the opposite direction (i.e., (x,



(a) Simulation result of 403.gcc (group-A)



(b) Simulation result of 403.gcc (group-B)

Figure 10: Simulation result of 403.gcc-403.gcc, showing the design space of LLC ways and memory request buffer size allocated to 403.gcc (group-A). The colored region in the top figure displays the design space where satisfying normalized IPC higher than 0.9, and the bottom figure shows only the remaining region of 403.gcc (group-B).

y)=(1, 1) in (a) has the same IPC with (19, 19) in (b)). Note that the configurations that yield >90% of the standalone IPC for group-A are colored in red in Figure 10(a) and that we only show the IPCs of the *corresponding configurations* for group-B in Figure 10(b).

With a simplifying assumption of (1) cache ways being the only knob we control for resource allocation and (2) a service level objective (SLO) of 90% of the standalone IPC for group-A (group-B has no SLO), we estimate the potential for saving machine count from workload consolidation. We further assume it takes 1,000 dedicated machines for each of the two application groups in standalone mode to satisfy the application throughput requirement. From Figure 10, we can select an appropriate configuration of the number of LLC ways and that of buffer entries to meet the SLO target for group-A and also to maximize the throughput of group-B. For example, if we choose the point of 49.1% IPC for group-B, which is the best IPC achievable while providing a 90% IPC for group-A, we can run group-A and B concurrently on 1,111 machines for group-A and B, and dedicate 454 extra machines to group-B to maintain the same throughput as 2,000 dedicated machines. This consolidation is only possible with mVC because, without it, group-A cannot satisfy the IPC SLO in a consolidated machine due to MiW. Thus, mVC can save 21.8% of machines compared to the baseline with LLC partitioning only, which would still require 2,000 dedicated machines to satisfy the throughput and IPC SLO. Applying the same methodology, we can save the operating cost by 7.9% and 13.3% for the other two pairs of SPEC benchmarks (473.astar-403.gcc and 523.xalancbmk-523.xalancbmk) without violating SLO.

VIII. Related Work

Component-wise QoS/fairness for shared resources. A myriad of techniques has been proposed to support quality-of-service (QoS) and fairness for shared on-chip resources, such as caches [15], [18], [31], [39], [49], [50], [51], on-chip interconnects (NoCs) [14], [25], [37] and DRAM bandwidth [22], [35], [36], [48], [52]. For caches, Suh et al. [49] introduce a dynamic monitoring scheme for the shared cache accessed by multiple concurrent threads and apply it to cache partitioning to minimize the total miss count. Qureshi and Patt [39] improve this by using utility-based cache partitioning (UCP). CQoS [18] identifies the QoS problem in the shared LLC among concurrent threads to propose cache partitioning based on priority classifications.

Locally-fair arbitration in NoC can result in global unfairness, creating *parking lot problem* where remote traffic is penalized by going through more arbitrations. Recent proposals addressing this problem include Globally Synchronized Frames (GSF) [25], Preemptive Virtual Clock (PVC) [14], probabilistic arbitration [26], and LOFT [37], providing fair bandwidth allocation. Song et al. [44] observe an opposite problem in processor-interconnects of NUMA servers, where a remote flow may receive more bandwidth than highly-contended local flows. However, these prior work do not address unfairness from cache partitioning.

Finally, DRAM banks and channels are other major sources of inter-thread interference. Multiple access streams from different threads may be interleaved to reduce the row buffer locality of DRAM accesses, hence degrading QoS and overall throughput. A variety of DRAM access schedulers have been proposed to recover locality and provide QoS [22], [35], [36]. For example, ATLAS [22] prevents memory-intensive processes from monopolizing the memory bandwidth by prioritizing requests from the least-attained memory service thread (the expected shortest job). Though effective for QoS, ATLAS is originally designed to maximize total throughput. MISE [48] estimates the slowdown of an application caused by memory interference through occasionally prioritizing the application over other co-running workloads; it then applies the model to devise scheduling schemes with better QoS.

However, these *component-wise* QoS techniques fail to provide robust performance without considering a complex interplay between different resources (e.g., LLC ways vs.

DRAM bandwidth) as demonstrated in this paper and other literature [12], [30].

Holistic approaches to QoS/fairness. Unlike the component-wise QoS techniques, some QoS frameworks propose to manage multiple shared resources holistically. Fairness via Source Throttling (FST) [12] and GSF memory system (GSFM) [24] aim to achieve better QoS along the shared memory access path by memory injection control at each source. ASM [47] extends MISE [48] by quantifying the effect of interference from co-running applications at a shared cache by using an auxiliary tag store. Then it models application slowdowns due to interference at both the shared cache and main memory and applies the model to improve performance and fairness of the applications. Iyer et al. [19] and Heracles [30] provide performance isolation by jointly partitioning both cache space and memory bandwidth. While providing better end-to-end QoS than component-wise QoS approaches, their solutions are incomplete as they do not prevent blocking caused by shared DRAM request buffers. We show the existence of this problem and propose mVC to resolve it.

IX. Conclusion

In this work, we have demonstrated on real server machines how applications with more allocated LLC capacity can perform worse. Cache partitioning is promising for performance protection of a process by dedicating a portion of LLC, alleviating contention and interference from other processes. Because LLC is a shared resource with limited capacity, when we allocate more LLC capacity to one application, others receive relatively small LLC capacity. This results in a higher LLC MPKI and stresses the congested datapath within memory controllers, which is another shared resource below the shared LLC, causing blocking, slowing down the entire system (a balloon effect). In particular, we identified this MiW phenomenon on latency-critical workloads could deteriorate 95th percentile latency as worse as 547%. To overcome this MiW, we proposed to virtualize the shared datapath of memory controllers by mVCs. mVCs mostly eliminate the MiW phenomenon and improve the performance as the allocated LLC capacity increases, restoring the performance protection intended by cache partitioning. We can reduce the overall system cost using mVCs with a proper memory request queue size and LLC capacity while satisfying the target performance of latency-critical workloads even when executed with multiple workloads together. Results show that on SPEC CPU2006 workloads, up to 21.8% system cost can be saved while obtaining 90% of the performance compared to stand-alone execution on a dedicated machine.

Acknowledgment

This research was supported in part by Samsung Advanced Institute of Technology, and by Nano-Material Technology Development Program (2016M3A7B4909668) and Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2015M3C4A7065647).

References

[1] S. Abousamra, A. El-Mahdy, and S. Selim, "Fair and Adaptive Online Set-based Cache Partitioning," in *Computer Engineering & Systems (ICCES), International Conference on*, 2011.

[2] J. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A Manycore Simulator with Application-level+ Simulation and Detailed Microarchitecture Modeling," in *ISPASS*, 2013.

[3] AMD, "BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors," 2006.

[4] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[5] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, 2nd Edition," 2013.

[6] J. Chang and G. S. Sohi, *Cooperative Caching for Chip Multiprocessors*. ISCA, 2006.

[7] D. Chiou, P. Jain, S. Devadas, and L. Rudolph, "Dynamic Cache Partitioning via Columnization," in *DAC*, 2000.

[8] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, 2010.

[9] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[10] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, 2013.

[11] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake," *IEEE Micro*, vol. 37, no. 2, 2017.

[12] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems," in *ASPLOS*, 2010.

[13] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Bandwidth Bandit: Quantitative Characterization of Memory Contention," in *CGO*, 2013.

[14] B. Grot, S. W. Keckler, and O. Mutlu, "Preemptive Virtual Clock: A flexible, efficient, and cost-effective QOS scheme for networks-on-chip," in *MICRO*, 2009.

[15] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5-2600 v3 Product Family," in *HPCA*, 2016.

[16] Intel, *Intel Xeon Processor 7500 Series Datasheet*, 2010.

[17] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manuals*, 2018.

[18] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," in *ICS*, 2004.

[19] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," in *SIGMETRICS*, 2007.

[20] H. Kasture and D. Sanchez, "TailBench: A Benchmark Suite and Evaluation Methodology for Latency-critical Applications," in *IISWC*, 2016.

[21] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in *ASPLOS*, 2014.

[22] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.

[23] M. K. Kumashikar, S. G. Bendi, S. Nimmagadda, A. J. Deka, and A. Agarwal, "14nm Broadwell XeonÂő processor family: Design methodologies and optimizations," in *IEEE Asian Solid-State Circuits Conference (A-SSCC)*, 2017.

[24] J. W. Lee, "Globally Synchronized Frames for Guaranteed Quality-of-Service in Shared Memory Systems," Ph.D. dissertation, MIT, 2009.

[25] J. W. Lee, M. C. Ng, and K. Asanovic, "Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks," in *ISCA*, 2008.

[26] M. M. Lee, J. Kim, D. Abts, M. Marty, and J. W. Lee, "Probabilistic Distance-Based Arbitration: Providing Equality of Service for Many-Core CMPs," in *MICRO*, 2010.

[27] F. Liu, X. Jiang, and Y. Solihin, "Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance," in *HPCA*, 2010.

[28] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in *PACT*, 2012.

[29] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards Energy Proportionality for Large-Scale Latency-Critical Workloads," in *ISCA*, 2014.

[30] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *ISCA*, 2015.

[31] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic Shared Cache Management (PriSM)," in *ISCA*, 2012.

[32] S. Mittal, "A Survey of Techniques for Cache Partitioning in Multicore Processors," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, 2017.

[33] A. M. Molnos, M. J. Heijligers, S. D. Cotofana, and J. T. van Eijndhoven, "Compositional, Efficient Caches for a Chip Multi-processor," in *DATE*, 2006.

[34] T. Moscibroda and O. Mutlu, "A Case for Bufferless Routing in On-chip Networks," in *ISCA*, 2009.

[35] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[36] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair Queuing Memory Systems," in *MICRO*, 2006.

[37] J. Ouyang and Y. Xie, "LOFT: A High Performance Network-on-Chip Providing Quality-of-Service Support," in *MICRO*, 2010.

[38] M. Priyanka V P and M. K. Pramilarani, "An Analytical Model for Optimum Off-Chip Memory Bandwidth Partitioning in Multi-core Architectures," 2016.

[39] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *MICRO*, 2006.

[40] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.

[41] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *MICRO*, 2010.

[42] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-grain Cache Partitioning," in *ISCA*, 2011.

[43] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *ASPLOS*, 2002.

[44] W. Song, G. Kim, J. Chung, J. Ahn, J. W. Lee, and J. Kim, "History-based Arbitration for Fairness in Processor-Interconnect of NUMA Servers," in *ASPLOS*, 2017.

[45] Standard Performance Evaluation Corporation, "SPEC CPU2006," 2006. [Online]. Available: https://www.spec.org/cpu2006/

[46] Standard Performance Evaluation Corporation, "SPEC CPU2017," 2017. [Online]. Available: https://www.spec.org/cpu2017/

[47] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.

[48] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.

[49] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," *Journal of Supercomputing*, vol. 28, no. 1, 2004.

[50] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "DCAPS: Dynamic Cache Allocation with Partial Sharing," in *EuroSys*, 2018.

[51] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-Service," in *EuroSys*, 2018.

[52] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *RTAS*, 2014.

[53] X. Zhang, S. Dwarkadas, and K. Shen, "Towards Practical Page Coloring-based Multicore Cache Management," in *EuroSys*, 2009.

[54] M. Zhou, Y. Du, B. Childers, D. Mosse, and R. Melhem, "Symmetry-Agnostic Coordinated Management of the Memory Hierarchy in Multicore Systems," *ACM TACO*, 2016.