A Closer Look at Intel Resource Director Technology (RDT)

Parul Sohal
Boston University
psohal@bu.edu

Michael Bechtel
University of Kansas
mbechtel@ku.edu

Renato Mancuso
Boston University
rmancuso@bu.edu

Heechul Yun
University of Kansas
heechul.yun@ku.edu

Orran Krieger
Boston University
okrieg@bu.edu

ABSTRACT

Unarbitrated contention over shared resources at different levels of the memory hierarchy represents a major source of temporal interference. Hardware manufacturers are increasingly more receptive to issues with temporal interference and are starting to propose concrete solutions to mitigate the problem. Intel Resource Director Technology (RDT) represents one such attempt. Given the wide adoption of Intel platforms, RDT features can be an invaluable asset for the consolidation of real-time systems on complex multi- and many-core machines.

Unfortunately, to date, a systematic analysis of the capabilities introduced by the RDT framework has not yet been conducted. Moreover, no clear understanding has been matured about the implementation-specific behavior of RDT primitives across processor generations. And ultimately, the ability of RDT to provide real-time guarantees is yet to be established.

In our work, we conduct a systematic investigation of the RDT mechanisms from a real-time perspective. We experimentally evaluate the functionality and interpretability of RDT-aided allocation and monitoring controls across the two most recent processor generations. Our evaluations show that while some features like Cache Allocation Technology (CAT) yield promising results, the implementation of other primitives such as Memory Bandwidth Allocation (MBA) has much room for improvement. Moreover, in some cases, the presented interfaces range from blurry to incomplete, as is the case for MBA and Memory Bandwidth Monitoring (MBM).

KEYWORDS

memory management, bandwidth control, shared cache partitioning, performance guarantees, application profiling, Resource Director Technology, CAT, MBA, MBM, CMT, RDT

ACM Reference Format:
Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. 2022. A Closer Look at Intel Resource Director Technology (RDT). In *Proceedings of the 30th International Conference on Real-Time Networks and Systems (RTNS '22), June 7–8, 2022, Paris, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3534879.3534882

RTNS '22, June 7–8, 2022, Paris, France

ACM ISBN 978-1-4503-9650-9/22/06…$15.00
https://doi.org/10.1145/3534879.3534882

1 INTRODUCTION

As multi-core systems have become more prevalent, providing accurate performance guarantees has become increasingly challenging. With the advent of additional computing units, such as accelerators, the pressure on the limited shared cache and memory controllers has increased [3, 51, 55, 56]. Embedded research areas, such as real-time edge computing have become increasingly more data-intensive. The real-time community has developed multiple hardware and software techniques to mitigate both spatial and temporal interference at different levels of the memory hierarchy [9, 12, 13, 19, 49, 63, 67]. As already recognized in some of these works, high-performance platforms are an attractive alternative for complicated, highly interconnected real-time systems in need of soft real-time guarantees [60, 61, 63].

Shared caches lead to inter-core interference as different tasks can evict each other's data from the cache and, as a result, make calculating the worst-case execution time (WCET) convoluted and pessimistic [32, 41, 64, 66] or overly pessimistic [20, 59]. Similarly, main memory bandwidth is a significant bottleneck when various applications are concurrently access DRAM [2, 30, 49, 66].

Hence, hardware techniques for cache partitioning and main memory bandwidth regulation have found widespread adoption in the real-time community [6, 67]. For example, ARM introduced a specification for Memory Partitioning and Monitoring (MPAM) in 2017, which enables monitoring and control over the main memory usage [5]; ARM QoS extensions are already available in commercial systems [49, 50]. In parallel, Intel has implemented new hardware support for cache partitioning and memory bandwidth regulation under their Resource Director Technology (RDT) umbrella. For cache partitioning, Intel has introduced Cache Allocation Technology (CAT), which has already been used in a number of works [61, 63]. For bandwidth regulation, Intel has introduced Memory Bandwidth Allocation (MBA) [46, 60]. The management tools are accompanied by their corresponding monitoring support, i.e., Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM).

In this paper, we provide a comprehensive analysis of the RDT mechanisms as their use become more common in the real-time community. Using synthetic benchmarks, we study their effectiveness in providing temporal isolation for co-running applications. Furthermore, we also investigate RDT monitoring features. We substantiate our conclusion on the maturity of these features with experiments conducted on two Intel platforms of different generations, namely Cascade Lake and Ice Lake.

2 RELATED WORK

Shared resource contention is a well-known issue in the real-time community, and many research papers have explored performance degradation in multi-core systems [38]. For example, Sha et al. [48] demonstrated how contention at both the shared last-level cache (LLC) and main memory impacts real-time performance. Knowing this, research works have proposed and used several resource allocation techniques to limit and bound contention over shared resources at different levels of the memory hierarchy [19, 39, 39, 44, 52, 57, 62, 65, 67, 68]. Some works mandate strict resource partitioning between individual cores [31, 40, 48], or propose to decompose real-time tasks into a sequence of memory and execution phases that can be explicitly scheduled. The Predictable Execution Model (PREM) [47] and Acquisition Execution Restitution (AER) model [37] are such examples. Additionally, specific works have proposed custom hardware resource management primitives to resolve contention over shared resources at different levels of the memory hierarchy [15, 35, 36, 38]. For instance, the work in [36] proposes a methodology to prioritize memory requests from high criticality tasks by tagging them. Tag-based prioritization is enforced via customizable hardware modules in the memory sub-system. In this work, we focus on a commercially available resource management primitives.

In particilar, we focus on the analysis of Intel RDT. Intel introduced RDT features in the latest-generation Xeon processors to manage contention over key memory resources. Several research works have already employed Intel RDT technologies. Most notably, many works employed CAT to manage cache allocation for critical applications [19, 61, 63]. Farshin et al. implemented a slice-aware cache management framework and compared it to CAT [16]. In particular, they find that enabling CAT does not provide the desired isolation and that limiting the available cache ways can instead add more pressure on main memory bandwidth. However, the critical assumption in these works is that CAT partitioning provides deterministic results and can be trusted in providing way-based cache partitioning.

Similarly, many techniques to regulate DRAM bandwidth have been proposed. In software, a popular approach has been to monitor memory bandwidth at the OS-level with LLC miss performance counters and throttle cores that exceed a set bandwidth threshold [67]. More recently, hardware-level designs have proposed arbiters capable of enforcing bandwidth partitioning among co-running applications [14, 44, 49, 57, 68]. Intel MBA—which is part of RDT—is one such hardware-based bandwidth regulation mechanism. Even though MBA is a recent addition to RDT, some works have already attempted to apply MBA to decrease the performance degradation caused by unregulated applications sharing the limited main memory bandwidth [46, 60]. However, these works focus on the earlier version of MBA which is known to have major limitations and bugs [27].

While much prior research fundamentally assumed that CAT and MBA have been correctly implemented, our goal is to specifically put this hypothesis to the test. The description of the errors in the document presented by Intel is vague and not comprehensive [27]. For example, specific workloads, especially memory intensive, can use more of the shared cache than what allocated to them. Also, MBA might throttle the cores at a different setting than the one applied if the hardware register controlling the settings is read directly after changing the MBA level. Differently from the provided errata documents, our approach is experiment-driven and our conclusions are tailored to the applicability of RDT from a real-time perspective.

As RDT mechanisms become more common and more research works attempt to build atop RDT to restore predictability, it is crucial to analyze their functionality and maturity in depth. We experimentally investigate the performance of both allocation and monitoring primitives across two generations of processors. Furthermore, we study whether the RDT framework is reliable enough to be used by the real-time community by documenting the level of isolation guarantees the resource allocations offer and the accuracy of the monitoring counters. Therefore, this paper sets itself apart from previous works that have used Intel RDT mechanisms as black-boxes.

3 RESOURCE DIRECTOR TECHNOLOGY

In this section, we provide an overview of the different components of Intel RDT. The objective of these techniques, as explained by Intel, is to reduce performance interference at the shared cache and the main memory subsystem while enabling "key" applications to maintain desired progress when the system has multiple applications running [10, 23]. These design goals are in line with the effort placed by the real-time community in ensuring that high-criticality tasks receive temporal isolation. Hence, the RDT framework is (in principle) a viable alternative to existing techniques and tools.

RDT is made up of five mechanisms that can be subdivided into resource allocation and resource monitoring capabilities - 1) Cache Allocation Technology (CAT), 2) Code and Data Prioritization (CDP), 3) Cache Monitoring Technology (CMT), 4) Memory Bandwidth Allocation (MBA), and 5) Memory Bandwidth Monitoring (MBM). CAT and MBA help manage the shared cache and main memory bandwidth, respectively. CDP is an extension of CAT which enables the user to select the placement of code and data in the shared cache. CMT and MBM monitor the shared cache and main memory bandwidth. RDT was first introduced in Intel Xeon E5 v3 family of processors with limited functionality [25]. Our paper uses two different generations of Intel platforms: 1) 2nd Generation Xeon Scalable Processors (Cascade Lake), and 2) 3rd Generation Xeon Scalable Processors (Ice Lake). Table 1 lists the platforms that support RDT features.

3.1 Resource Allocation

It has been extensively shown that sharing of unregulated hardware resources leads to performance degradation due to a lack of temporal isolation [7, 8, 43, 58, 65, 67]. Partitioning resources ensures high-criticality applications can maintain their quality of service (QoS) by mitigating interference. The resource allocation mechanisms provided by RDT work on the same principle. They allow system designers to enforce specific limits on using performance-critical shared hardware resources by applications scheduled on individual cores.

Table 1: RDT with Availability Details

	RDT Components	Variations	Generations
ALLOCATE	Cache Allocation Technology	L2	Atom Server C3000 11th Gen i3,i5,i7 Atom X Series Xeon W
		L3	Xeon E3 v4 Xeon E5 v3,v4 Xeon D Xeon Scalable **Xeon Scalable Gen2** **Xeon Scalable Gen3** 11th Gen i3,i5,i7 Atom X Series Xeon W
	Code and Data Prioritization	L2	11th Gen i3,i5,i7 Atom X Series Xeon W
		L3	Xeon E5 v4 Xeon Scalable Xeon Scalable Gen2 Xeon Scalable Gen3
	Memory Bandwidth Allocation	MBA 1.0	Xeon Scalable **Xeon Scalable Gen2**
		MBA 2.0	**Xeon Scalable Gen3** Snow Ridge
		MBA 3.0	Xeon Scalable Gen4
MONITOR	Cache Monitoring Technology	N/A	Xeon E3 v4 Xeon E5 v3,v4 Xeon D Xeon Scalable **Xeon Scalable Gen2** **Xeon Scalable Gen3**
	Memory Bandwidth Monitoring	N/A	Xeon E5 v4 Xeon D Xeon Scalable **Xeon Scalable Gen2** **Xeon Scalable Gen3**

Currently, RDT specifications support management of L2 cache, L3 cache, and main memory bandwidth. The number of implemented management controls differs between processor generations. For cache management, the partitioning is way-based, whereas the main memory bandwidth controls limit the amount of bandwidth extracted on a per-core basis. In this paper, we focus on LLC partitioning and bandwidth throttling. In Section 3.1.1 we discuss CAT, followed by two versions of MBA in Section 3.1.2.

RDT uses a notion of "criticality" to manage applications. Intel calls this Class of Service (CLOS/COS). The number of CLOS available on a machine varies. Multiple cores can be mapped to one CLOS. All cores with the same CLOS abide to the resource allocation policy associated with that CLOS.

The `CPUID` assembly instruction can be used to see the resource allocation features that exist for a particular platform [24][1]. If allocation for a resource exists, additional information about various knobs for controlling the resource partition is also provided. Specifically, for each resource type, `CPUID` instruction tabulates the number of CLOS and the available allocation settings. By using per-CLOS model-specific registers (MSRs), one can define the quantity of a particular resource available to a given CLOS.

Once the values are assigned to each CLOS using the various MSRs, a one-to-many mapping can be created to associate each processor to its respective CLOS's available resources. This mapping is established by setting the value for an MSR that exists on each logical core to one of the CLOS identifiers. This control register, namely `IA32_PQR_ASSOC`, can also be read to retrieve the current CLOS at each context switch.

To summarize, to correctly allocate resources to applications, the following steps need to be performed: 1) for each resource that can be managed (e.g. shared cache, main memory bandwidth) use the MSRs to define a partitioning scheme for shared resources for each of the CLOS, and 2) create a mapping of cores to CLOS.

The remainder of this section describes the interfaces for managing shared cache and memory bandwidth resource allocations.

3.1.1 Cache Allocation Technology. CAT has been extensively used to provide temporal isolation at the shared cache level [19, 34, 61, 63]. CAT itself is a hardware way-based partitioning mechanism. On the two micro-architectures that we considered, i.e., Cascade Lake and Ice Lake, there are 16 CLOS available.

To do CLOS-based cache way partitioning, CAT provides a set of per-CLOS MSRs where a bitmask forces the assignment of cache ways. In each of those registers, the least W significant bits encode whether each of the W ways can be used for allocation by any of the CPUs in the considered CLOS. These registers are called, `IA32_L3_MASK_n` MSRs where $n \in \{0, ..., 15\}$ is the number of CLOS available.

For example, setting `IA32_L3_MASK_5=0b00011100000` expresses that CLOS5 allows a core to allocate cache lines in only three cache ways. This can be used to portion out a fixed number of ways that are not allocated to any other CLOS. However, nothing in the current definition prevents CLOS from sharing cache ways. For example, `IA32_L3_MASK_4=0b0011_0000000` could be assigned this value, which means that CLOS4 and CLOS5 share only one cache way. By default, all CLOS have access to all the cache ways as all of the bits in their MSR bitmasks are set to one after reset.

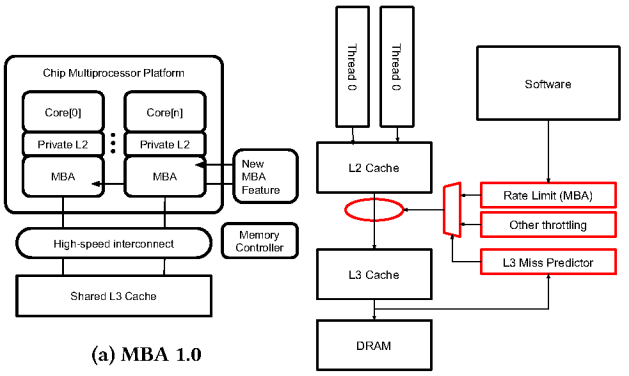
Until now the literature has employed CAT for isolation, but no systematic effort has been placed in investigating the strength of the isolation that can be achieved with CAT. Over the years, many researchers have also reported bugs in different generations of CAT, including Intel themselves [27]. In this paper, we provide one systematic approach to assess CAT's validity along with methods to improve temporal isolation when using current implementations of CAT.

3.1.2 Memory Bandwidth Allocation. Currently, there are three different implementations of MBA: MBA1.0, MBA2.0, and MBA3.0. The available CLOS are different in the three versions of MBA, and the granularity of MBA controls is linear, starting at 10% up to 100% in increments of 10%. Percentage to represent regulation settings is a piece of notation used by Intel [26]. Because it is unclear what these percentages refer to, we call these settings "throttling levels "(TLs).

[1]When RDT exists, use `CPUID` with EAX=0X10 and ECX=0x0 to obtain the list of RDT features available on the current platform. Register EDX reveals which resources for either monitoring or allocating exist. Please refer to the Intel Software Manual for more details [26].

Like CAT, MBA uses CLOS-based bandwidth allocation and provides a set of MSR registers, `IA32_L2_QoS_Ext_BW_Thrtl_n` MSRs, where n represents the CLOS. Only values corresponding to available throttling levels are acceptable, such as 10%, 20%.

MBA1.0: The initial implementation of MBA enacted indirect throttling over the bandwidth. This implementation of MBA uses a Programmable Rate Controller (Figure 1a) between the per-core L2 cache and the L3 interconnect. It would appear that the controller introduces a constant delay on L2 cache misses based on the throttling level. This hypothesis on the behavior of MBA1.0 is consistent with the use of the term "delay" interchangeably with threshold in the documentation, as well as with the statement that "MBA throttles accesses to the last-level cache, and care should be taken to not throttle applications which are LLC-intensive [26]."



(a) MBA 1.0

(b) MBA 2.0

Figure 1: Implementation differences between MBA1.0 [22] and MBA2.0 [28]

MBA2.0: In MBA2.0, the changes can be separated into two components. First, the amount of throttling enacted by each MBA TL is not fixed but is subject to change with BIOS calibration. Using different calibration settings, the control over the inter-spacing between requests for different MBA settings can be modified[2]. Second, the addition of a new hardware controller to measure the precise bandwidth delivered to a CPU allows the controller to limit the request rate to meet the bandwidth setpoint specified for the associated CLOS [28]. This is similar to traditional feedback-based bandwidth control [1].

By having a hardware controller external to the core parts of the processor, the newer implementation of MBA can, in theory, provide more refined control over the bandwidth allocated to different CLOS. Furthermore, as the controller tracks the requests made to the DRAM and not to the last-level cache. This prevents throttling requests that could be satisfied by the L3 cache.

MBA3.0: The only difference between MBA2.0 and MBA3.0 is that the assignment of throttling levels is no longer restricted to a per-core basis. As the CLOS association registers are defined on a per-logical-core basis, MBA settings can be set to different levels for each hyper-thread. This is different from MBA1.0 and MBA2.0 where the same throttling level was applied to all the hyper-threads on the same physical core.

[2]Albeit the feature is mentioned in the manuals, both the machines we used provided no options for the calibration of MBA in the BIOS. This might be available in the future.

3.2 Resource Monitoring

Monitoring the resource utilization of an application at different levels of the memory hierarchy can help ensure adequate amount of resource allocation and prevent over-provisioning [19, 61]. Furthermore, monitoring can also help contextualize the observed performance degradation when contention over shared resources occurs [18, 49].

Monitoring is performed by tagging each core via a Resource Monitoring ID (`RMID`) [26]. This infrastructure in RDT is common across both of the monitoring features (CMT and MBM). Multiple cores can be set to the same `RMID`; this can be useful to monitor an application running on multiple cores simultaneously. The set of supported event types (resources that can be monitored) varies by generation.

An event code represents the shared resource that needs to be monitored. Currently, RDT only supports three event codes: 1) L3 cache occupancy, 2) total external bandwidth, and 3) local bandwidth. Before we can retrieve the data, the MSR register, `IA32_QM_EVTSEL` needs to be set with an event code and `RMID` for which the resource monitoring is presented. Multiple event codes can be stored in this register to monitor multiple resources simultaneously. Same with `RMID`s. Once this is done, the data can be read from the counter register `IA32_QM_CTR`.

3.2.1 Cache Monitoring Technology. The `RMID` along with CMT event code provides the last-level cache occupancy for the application/core linked to it. CMT and other L3 performance counters can help configure the appropriate cache partitioning for sensitive cache tasks subject to temporal constraints.

3.2.2 Memory Bandwidth Monitoring. In complex systems, applications can perform memory requests to memory subsystems outside the scope of the local processor. Multi-socket systems with non-uniform memory access (`NUMA`) can differentiate the type of memory requests into "local" and "remote." Local requests represent L3 misses, including prefetches, that a memory controller completes on the same socket [26]. Remote requests are L3 misses fulfilled by a memory controller attached to a different socket. RDT has two MBM event code options: 1) local external bandwidth and, 2) total external bandwidth. The former only includes the bandwidth extracted by requests completed by memory controller(s) on the local NUMA node. Conversely, the latter also includes the bandwidth extracted from other NUMA nodes. The bandwidth extracted by remote requests alone can be calculated by subtracting local from total bandwidth. Lastly, even though the manuals refer to what is measured as "bandwidth," in practice, the raw counter tracks the number of transferred bytes.

Each of the RDT features can be manipulated via the corresponding MSR register. The summary of the registers we considered is provided in Table 2. All these registers can be modified at run-time.

3.3 Scope of RDT Analysis

Section 5 and Section 6 discuss the results of CAT, CMT, MBA, and MBM on two different processor generations. The paper aims to

Register	Use
IA32_PQR_ASSOC	Set the desired CLOS and RMID Defined for each logical processor
IA32_QM_EVTSEL	Contains event codes and the RMID to be monitored Need to set before retrieving the data
IA32_QM_CTR	Reports the monitored data Contains bits for checking errors and validation
IA32_L3_MASK_n	Bitmask to assign cache ways to each CLOS n registers, one per CLOS
IA32_L2_QoS_Ext_BW_Thrtl_n	Set to one of the available throttling levels n registers, one per CLOS

Table 2: Summary of the MSR registers used in RDT.

justify whether the current implementation of RDT mechanisms is useful for the real-time community. We structured the two sections by asking a set of questions that we must evaluate before we can respond with a conclusive answer.

In particular, we focus on answering the following questions.

- Is CAT helpful to enforce LLC cache partitioning?
- Can we strengthen the degree of isolation provided by CAT?
- Are the results provided by CMT interpretable and in line with the theoretical value of the synthetic benchmark?
- Is MBA effective in throttling interfering cores to protect a target application?
- Are the limitations of MBA1.0 addressed in MBA2.0?
- How accurately is MBM able to track memory transactions of known applications?

By answering these questions, we aim to help future works make more informed design decisions when using RDT mechanisms.

4 TARGET PLATFORM AND SYNTHETIC BENCHMARKS

To better understand the experimental studies conducted in the following sections, we hereby introduce the target platforms. In this section, we also describe the synthetic benchmarks used to stress-test the systems and characterize the RDT tools' workings.

4.1 Target Platforms

We provide an analysis of CAT, CMT, MBA, and MBM on two different dual-socket Intel micro-architectures, released two years apart, Cascade Lake and Ice Lake.We assume a single victim task and the remaining physical cores on the same socket are used by co-running tasks. A summary of the main system characteristics can be found in Table 3 [11]. The platforms used in this paper were released in April 2019 and 2021, respectively.

This paper uses more complex hardware than the traditional embedded platform as our goal is to evaluate the most complete RDT implementation available to date. Notwithstanding, a subset of RDT mechanisms is available on Intel's embedded platforms. For instance, Tiger Lake, released in late 2020, introduces support for CAT. We expect full-fledged RDT implementations to be more available in embedded platforms as the remaining features (CMT, MBA, MBM) become more stable.

The architecture of the Cascade Lake processor is shown in Figure 2. The CPU cores are connected in a mesh, where mesh components are the individual cores (with a slice of LLC), the memory controllers, PCI lanes, and Ultra Path Interconnect (UPI) endpoints. Cascade Lake systems have two memory controllers with three

Feature	Details
CPU	Intel Xeon Gold 6248 40c/80t, dual socket 2.5 GHz base, 3.9 GHz boost
Cache	32 KB(I)/32KB(D) L1 cache 1 MB L2 cache 27.5 MB L3 cache (LLC)
Memory	~ 376 GB Node 0 ~ 378 GB Node 1 6 DDR4-2933 MHz (per node) 2 Memory Controllers (per node)
RDT	L2 CAT (not tested) L3 CAT L3 CMT MBA1.0

(a) Cascade Lake Machine (Released 2019)

Feature	Details
CPU	Intel Xeon Gold 6338 64c/128t, dual socket 1.8 GHz base, 2.4 GHz boost
Cache	32 KB(I)/48 KB(D) L1 cache 1.25 MB L2 cache 48 MB L3 cache (LLC)
Memory	~ 31GB Node 0 ~ 31.5GB Node 1 8 DDR4-3200 MHz (per node) 4 Memory Controllers (per node)
RDT	L3 CAT L3 CMT MBA2.0 MBM

(b) Ice Lake Machine (Released 2021)

Table 3: Notable System Characteristics

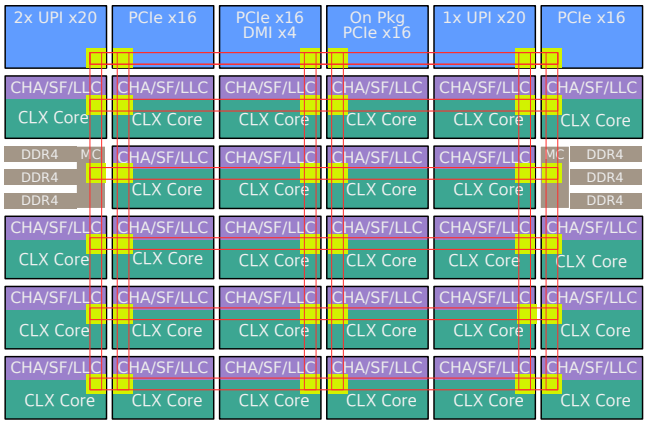


Figure 2: Cascade Lake Processor Architecture [17]

channels each. The newer Ice Lake systems have a similar design but contain four memory controllers with two channels per controller.

4.2 Noise Control

In order to limit performance fluctuations, we configured the target systems as follows. First, each of the experiments in this paper is run 30 times to provide statistically significant results. Next, we disable simultaneous multi-threading (SMT) and only consider a single socket. We also disable the dynamic frequency governor

such that the processor operates at the fixed base frequencies on each micro-architecture mentioned in Table 3. We also disable Intel Turbo Boost in our experiments.

Additional features that introduce non-determinism in the performance measurements are disabled. These include hardware prefetchers, OS-level load balancing and power-saving features. Specifically, load balancing was restricted via the `isolcpus` Linux kernel boot parameter [21] for all the cores in the socket under analysis. Furthermore, the kernel is compiled with the `NOHZ_FULL` [4] configuration option to disable the scheduling ticks when the CPU is idle or has only one application scheduled on the core. This is true in our experimental setup as each application is explicitly pinned to a core and not allowed to migrate. Power-saving features were disabled by restricting the C-states [53] of the processors.

Each socket in the target platform has multiple main memory (DRAM) controllers, attached to a set of DRAM modules (DIMM) local to the socket. We restrict physical memory allocation for application workload to the local socket to limit inter-socket data exchange. We do so with a combination of two methods. First, we disable inter-socket memory interleaving. Second, we use the `numactl` [45] utility to force physical memory allocation from the local node/socket. We only consider one socket under analysis in the remainder of this paper. The other socket is left unloaded and is used for handling interrupts and other OS-level management tasks.

Platform	Tot. Ways	Vic. Ways	Co-Runner Ways	Vic. Part. MB
Cascade Lake	11	6	5	15
Ice Lake	12	6	6	24

Table 4: Static cache partitioning on two micro-architectures.

An essential factor that impacts performance isolation is contention over shared LLC cache space. We use strict cache partitioning for the workload under analysis via CAT [26]. We have fixed the number of ways allotted to the core under analysis unless mentioned otherwise. The rest of the LLC is collectively assigned to the remaining cores on the socket. As shown in Table 4, the same number of ways was assigned to the core under analysis for both the Cascade Lake and Ice Lake machines. The number of ways allocated to the victim core is 6 in both platforms, but the partition size is different as the two systems drastically differ in terms of total LLC size. In our Cascade Lake machine, six ways correspond to 15 MB of partitioned LLC, whereas in our Ice Lake machine, it adds up to 24 MB. Lastly, any change to the RDT registers is verified by reading back the registers value.

4.3 Synthetic Benchmarks

The synthetic workload we use in our experiments is designed to be memory intensive. We use the same "bandwidth" benchmark as in [67]. It iterates multiple times — until terminated — over a buffer with a given size. Each iteration performs a load or store every 64 bytes of data, which corresponds to the cache line size. Since there are no dependencies between consecutive requests, they can be carried out in parallel which maximizes the load on the DRAM. The benchmark estimates the bandwidth received by measuring its runtime and the number of completed memory operations. Depending on the size of the data buffer, this benchmark can be made LLC sensitive or DRAM sensitive.

DRAM-Bomb: In cases where we are interested in studying the performance impact of contention over main memory resources, we set up our synthetic benchmark to be DRAM sensitive. This is done by using a buffer of 3X the size of the shared cache (much bigger than the LLC cache partitioning in both platforms). When the synthetic benchmark is configured with these parameters, we refer to it as a "DRAM-Bomb."

LLC-Bomb: Contention over LLC bandwidth is another important aspect of our study. The synthetic benchmark described above is configured to maximize LLC interference. For our Cascade Lake machine experiments, we use a buffer size in the range [2.5 MB, 15 MB] since that is bigger than the L2 but still fits within the 15 MB cache partition.

On Ice Lake, the buffer size is in the range [4 MB, 24 MB]. It is three times the size of the L2 cache. When the synthetic benchmark is configured in this way, we refer to it as a "LLC-Bomb." Recall that apart from the core under analysis, all the other cores share an LLC partition of about 24 MB in our Ice Lake machine and 12.5 MB in our Cascade Lake system.

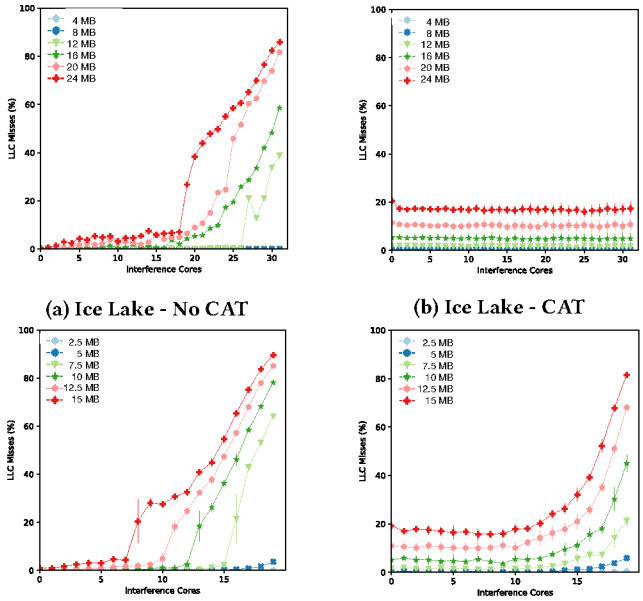
5 ANALYSIS OF CAT AND CMT

Cache partitioning is a widely used mechanism for providing temporal isolation at the shared last level cache between applications running on different cores simultaneously. In past research, both software and hardware-based mechanisms have been used [39, 52, 63]. In general, hardware techniques have lower overheads and do not need assistance from the OS or the compiler to create cache partitions [61, 63]. An alternative hardware approach is implemented in [16] where a slice-aware cache management methodology was proposed. The paper shows that memory access latency can be reduced by allocating memory in LLC slices that are closer to the core on the mesh architecture. The benefits of slice-aware allocation are beyond the scope of this paper. Specifically, this work focuses on Intel's hardware-based cache partitioning mechanism, i.e., CAT, which has previously been used in the real-time community. In this section, we take a closer look at this mechanism and the related cache monitoring primitive (CMT).

5.1 Is CAT Helpful to Enforce Partitioning?

Our first experiment is designed to understand the benefits of using CAT for LLC sensitive benchmarks. This is done by running the same application with and without a private cache partition allocated via CAT. The victim core executes an LLC-Bomb with varying buffer size performing *read* operations. The results are presented in Figure 3. The x-axis tracks the number of other active cores running DRAM-Bombs performing *write* operations. As the number of co-runners increases, the pressure exerted on the limited shared cache grows. The y-axis captures the percentage of LLC misses triggered by the victim core on two considered micro-architectures. The four sub-plots in Figure 3 present the results of four different configurations: 1) Ice Lake machine without CAT (Figure 3a), 2) Ice Lake machine with CAT (Figure 3b), 3) Cascade Lake machine without CAT (Figure 3c), and 4) Cascade Lake machine with CAT (Figure 3d). The cache misses are recorded via `perf`; a userspace utility for performance monitoring [33]. Also, as mentioned in Section 4.2, when CAT is used, 6 ways in each platform are provided to

the core under analysis. Each line on the four sub-plots represents a different working set size (WSS). Hence, on our Cascade Lake processor, the WSS of the application goes from 2.5 MB to 15 MB and on our Ice Lake system from 4 MB to 24 MB. The increment in the WSS is equivalent to the size of one cache way on the respective machines. Additionally, the largest buffer size in our experiments is when the WSS is equal to the size of the L3 cache partition provided to the victim core via CAT.



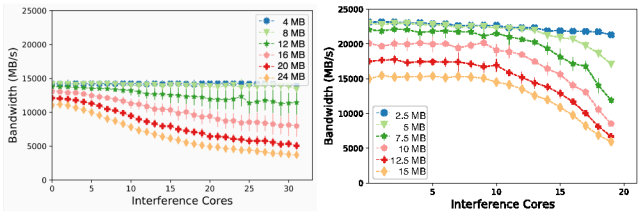
(a) Ice Lake - No CAT (b) Ice Lake - CAT

(c) Cascade Lake - No CAT (d) Cascade Lake - CAT

Figure 3: The difference between shared cache misses in percentage as the working set size increases with DRAM interference with and without CAT partitioning.

The answer to whether CAT provides isolation in the shared cache is multi-fold. When no hardware partitioning exists in both platforms, as the number of contenders increases, there is a drastic increase in the percentage of cache misses, as measured by hardware performance counters. In the worst-case scenario, the main memory serves 80% of the data accesses. Even though the application is cache sensitive, its progress is bounded by the rate at which main memory can satisfy read/write requests. The inflection point in the application's performance is dependent on its WSS. The larger the WSS, the earlier degradation in performance is observed.

Unfortunately, CAT behaves very differently in the two generations of processors. For fewer interfering cores (around less than 18 cores), the Ice Lake machine (newer generation) has more or equal cache misses with CAT than without. Without reserved shared cache, we see initial near 0% cache misses that increase slowly up to ~5%, regardless of the WSS. CAT changes this behavior as even with a few contenders, the percentage of LLC misses remains constant as shown in Figure 3b. For example, when the buffer size is 20 MB in the Ice Lake experiments, with no CAT, and when there are no co-runners, we see 0% cache misses. But with CAT under the same conditions, we observe ~13% cache misses. Essentially, CAT limits the LLC misses to a relatively constant value (flat line observed in Figure 3b) which is dependent on the WSS of the task compared to the allocated shared cache; the values start around 0% and go up to ~20% when the WSS is equal to the cache partition size. In short, for our Ice Lake platform, the observed LLC misses do not depend on the number of contenders performing DRAM-Bombs. This deterministic behavior is better than having a sharp increase in misses depending on the co-runners, as with no partitioning. Even though the results for our Cascade Lake system (Figure 3c and Figure 3d) initially perform similar to the newer generation, after a certain number of contenders, a dramatic increase in cache misses is observed. The initial flat lines for each WSS value ultimately exhibit exponential growth, similar in magnitude to the case with no hardware-based cache isolation. This increase is consistent with different WSSs and occurs roughly at 11 contenders (Figure 3d).



(a) Ice Lake (b) Cascade Lake

Figure 4: Impact on applications bandwidth while using CAT.

Because the number of cache misses is not null, even when CAT partitioning provides some determinism in the cache miss rate, we expect additional side effects as the number of contending cores increases. Specifically, the victim core suffers in terms of extracted main memory bandwidth. In our Cascade Lake platform, this decrease in bandwidth, as shown in Figure 4b, can be attributed to the increase in LLC misses. But for the Ice Lake machine (Figure 4a), the percentage of LLC misses stays constant even as the co-runners increase. Hence, it is necessary to note that even if the LLC misses reported by the application do not increase; this does not correlate to the application performing at peak performance. For the core under analysis, the main memory serves the LLC misses. As other cores run instances of DRAM-Bomb, the victim core competes for main memory bandwidth. Hence, even when LLC misses are constant, the victim core's performance will degrade due to contention over main memory.

No matter the generation, even when the WSS is half of the cache partition size, we observe a non-negative cache miss rate. As seen on these two platforms, even allocating a partition size three times larger than the WSS, our Ice Lake machine at 8 MB incurs ~2% of cache misses and for Cascade Lake system for the 5 MB WSS suffers ~8% cache misses. Furthermore, there is a lack of consistency between the results obtained on the two micro-architectures. While over-provisioning the cache partition is beneficial, cache space is an expensive resource, and over-provisioning negatively impacts the rest of the system.

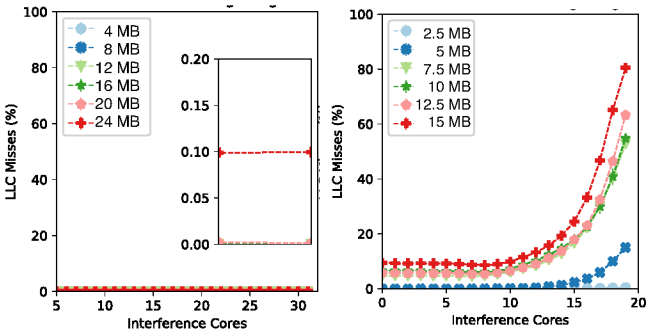
A plausible hypothesis for unexpectedly high LLC misses when the partition size is larger than the WSS is set conflicts. Set conflicts

are inevitable in CAT as CAT fundamentally reduces the *associativity* of the cache by partitioning cache ways. Reduced associativity causes more set conflicts for the same cache space. In contrast, set-based partitioning, i.e., page coloring, does not have this problem of reduced associativity and thus can potentially better utilize the given cache space [65].

Also, Intel platforms map addresses to cache sets by computing a hash function over multiple bits of the physical address [16, 29, 42]. It is reasonable to assume that the hash function is balanced over large enough continuous physical address spaces. In a typical OS, physical memory is allocated at the granularity of 4 KB pages. Demand paging causes the allocated pages to be spread randomly across the physical address space, creating unevenness in the cache sets allocated to the user-space applications. We experiment with allocating larger continuous sequences of physical addresses space by changing the default page size to be greater than 4 KB, as discussed in Section 5.2.

5.2 Can Cache Partitioning be Strengthened?

We increased the page size from the default 4 KB to 1 GB (huge pages). The findings from this experiment are depicted in Figure 5, with results on the Ice Lake machine on the left and the Cascade Lake system on the right. Increasing the page size helps distribute the physical addresses across the shared cache via the unknown hash function more uniformly. Reverse engineering the hash function is beyond the scope of this work, even though successful attempts have been made in the past [16, 42]. In both these platforms, there is abundant main memory. Hence, we study whether using huge pages effectively limits self-eviction due to set conflicts.



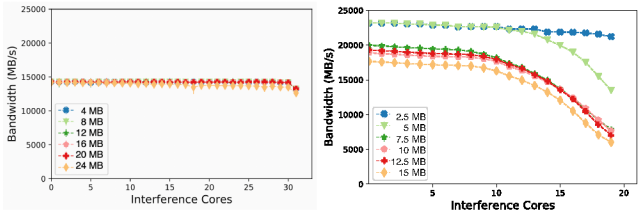
(a) Ice Lake - Huge Pages (b) Cascade Lake - Huge Pages

Figure 5: Percentage of LLC misses as the working set size increases with DRAM interference and with CAT partitioning and huge pages.

Again, the results in the two micro-architectures vary. The newer class of Intel platforms exhibit visible improvements from the use huge pages. This can be observed in the zoomed inset of Figure 5a. In most of the considered buffer sizes, the percentage of LLC misses drops to almost zero. The only exception is when the WSS of the synthetic LLC-Bomb is equal to the cache partition. Even in this situation, the cache misses are minimal at 0.10% compared to the 20% observed when the page size was 4 KB. Unfortunately, our Cascade Lake machine with huge pages does not exhibit the same drastic performance boost. When the WSS of the synthetic benchmark is equal to the allocated cache, the percentage of LLC misses drops from ~20% to ~10%. This is only true up to a certain number of contenders. When 11-15 co-runners are active, our victim core suffers an exponential performance degradation even with huge pages (Figure 5b).

Also, in our Ice Lake system (Figure 6a) we do not observe a decrease in bandwidth extracted by the victim core. The results for our Cascade Lake machine in Figure 6b corroborate the previous results where we observed an increase in LLC misses (Figure 5b).

In conclusion, we can have better results when the page size increases, but it is not guaranteed. It is crucial to analyze the performance of a system with CAT as optimal temporal isolation using CAT is strictly dependent on the specific RDT implementation. Indeed, the results vary depending on the number of co-runners and the considered micro-architectures.



(a) Ice Lake (b) Cascade Lake

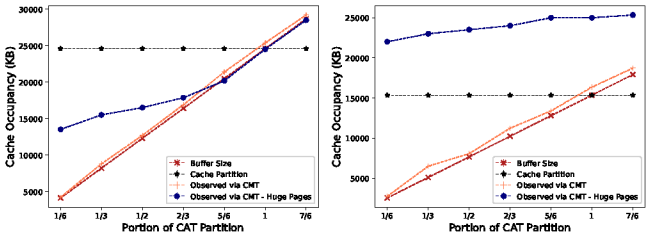
Figure 6: Impact on applications bandwidth while using CAT with huge pages.

5.3 Are CMT Results Interpretable?

Modern computing systems have substantially increased in complexity. Numerous hardware performance features have been introduced to shed light on the interplay between applications and hardware resources [5, 25, 33, 54]. Intel CMT, which tracks the cache occupancy for a given RMID, is another available performance monitoring capability.

We deploy the LLC-Bomb benchmark with a known WSS. The WSS ranges from size of one way to seven ways on respective machines. The size of the cache partition is six ways, as in all other experiments. For each of these scenarios, we monitor the cache occupancy reported by CMT. Figure 7 shows the findings of CMT on the considered two platforms. The cache occupancy in both micro-architectures without huge pages is slightly higher than the buffer size. Overall, it would appear that CMT can track the WSS of the target application. With huge pages enabled, the results on the two platforms differ drastically. The cache occupancy reported by CMT on the Cascade Lake platform is almost equal to the total cache size, whereas on the Ice Lake platform the cache occupancy is greater than the buffer size until the 5/6 portion case. In either case, it is hard to explain the values reported by CMT, as it is not clear how the cache occupancy of an application can be larger than the used buffer size. The discrepancy between the cache occupancy value reported by CMT for the same buffer size with and without huge pages is unexpected and undermines the reliability of CMT. Hence, the results from CMT are inconclusive. In theory, this counter might have remarkable practical value for conducting

a live analysis of cache occupancy in deployed applications and adjusting CAT-enforced partitioning accordingly. However, the current implementation appears to be too imprecise (at least when huge pages are used).



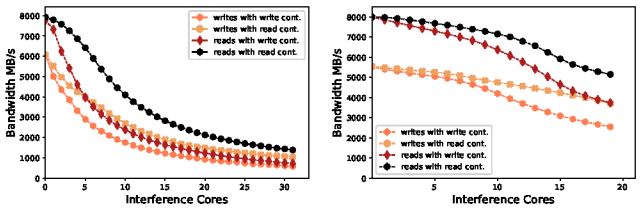
(a) Ice Lake (b) Cascade Lake

Figure 7: CMT results for Ice Lake and Cascade Lake with and without huge pages.

6 ANALYSIS OF MBA AND MBM

Memory bandwidth regulators, both hardware and software-based, are used in real-time systems to control the memory traffic to the DRAM subsystem. The goal is to limit the temporal interference between applications that share resources [3, 19]. A few research works have looked at the earlier version of MBA [46, 60] in the context of both real-time and general-purpose systems. This section shows the differences in behavior between the two versions experimentally and if they are viable for the real-time community.

MBA1.0, available on Cascade Lake machines, does not distinguish between memory requests fulfilled by the shared cache versus the main memory. This is due to the lack of additional hardware present on this micro-architecture to track the memory transactions leaving the core. Each MBA setting adds a fixed delay value to the requests sent to LLC. On the other hand, MBA2.0, on Ice Lake platforms, track the DRAM bandwidth through a hardware controller and can change the inter-arrival between requests in response.

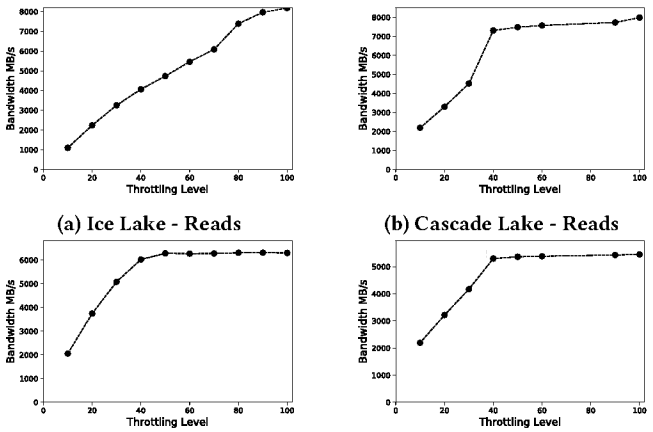


(a) Ice Lake (b) Cascade Lake

Figure 8: Degradation in performance due to interfering cores on the same socket.

6.1 Performance Degradation due to Limited Bandwidth

In Figure 8, our victim application experiences a drastic decline in the bandwidth received when multiple memory intensive applications are contending over the same shared resource – main memory. The x-axis shows the number of contending cores, and the y-axis reports the extracted main memory bandwidth. As more applications compete for the same limited main memory bandwidth, both the Cascade Lake system and the Ice Lake machine exhibit decline in performance. On our Ice Lake platform, when the victim core is the sole runner performing reads, the reported bandwidth is ~8000 MB/s. However, when 31 contending cores do the same, it drops to below 1500 MB/s, showing an 80% drop in main memory bandwidth performance. Our Cascade Lake system also demonstrates a similar trend, even with fewer cores contending on the same socket.



(a) Ice Lake - Reads (b) Cascade Lake - Reads

(c) Ice Lake - Writes (d) Cascade Lake - Writes

Figure 9: Default relation of throttling settings and bandwidth when performing reads vs writes.

6.2 How to Interpret MBA Settings?

For MBA to be useful, it is important to interpret the 10%, 20%,..., throttling levels. To translate the throttling levels to a bandwidth value, we run DRAM-Bomb pinned to a core and track the reported bandwidth over several executions. The results on both micro-architectures are shown in Figure 9. The x-axis denotes the throttling levels, and the y-axis captures the main memory bandwidth reported by the synthetic benchmark. The reported bandwidth over multiple runs is consistent.

Figure 9b and Figure 9d depict the results for MBA1.0. The results for bandwidth reported for both reads versus writes are comparable. 40% and above settings report the same bandwidth value. The main memory bandwidth for each core for lower levels is not restrictive enough. Furthermore, after the first few settings, the remainder of the options quickly reach the peak bandwidth that can be extracted via a core (~5200 MB/s for writes and ~8000 MB/s for reads). Lastly, as mentioned before, the throttling controls are implemented between L2 and L3. So, memory requests that might be served in LLC might also suffer delays.

On our Ice Lake platform (Figure 9a), when the victim core is only performing loads, we observe a linear trend in the bandwidth. This complies with the type of bandwidth control available on our current platform reported via `CPUID`. Each throttling level offers

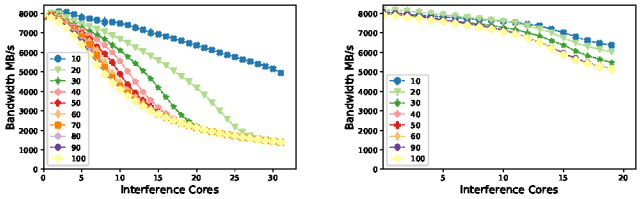
a distinct main memory bandwidth for the restricted core. However, when the core performs write operations, the trend (Figure 9c) becomes very similar to that of the Cascade Lake machine. The initial three settings report different main memory bandwidth, but beyond 30%, all throttling levels report the same extracted bandwidth. Manipulating the inter-spacing of memory requests with BIOS calibration might be possible. However, our current system does not have support for this. Hence, we can not verify it in this paper.

MBA's throttling is designed to throttle reads (cache-line refills) more harshly than writes (write-backs) in both implementations. Furthermore, another limitation of MBA that is evident from the current programming interface is that reads and writes cannot be independently regulated, even though they have a different impact on the application's performance and main memory controller saturation.

Knowing the limitations of MBA, it is still essential to investigate if MBA-enforced regulation is enough to shield an application with strict temporal constraints from performance interference.

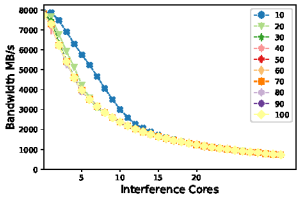
6.3 Is MBA Enough to Provide Protection?

Realizing the need for restricting certain non-critical applications while ensuring bandwidth guarantees for critical applications, Intel increased the scope of RDT to include MBA. In the next set of experiments, we evaluate if one can protect applications with strict temporal constraints by restricting the bandwidth available to interfering cores via MBA controls. The results from this study can be found in Figure 10. Each line on the plot represents an MBA setting for the interfering cores. On our Cascade Lake processor, only 8 options are available – 10%, 20%, 30%, 40%, 50%, 60%, 90% and 100%. On the Ice Lake machine, we have the full range from 10% to 100% in 10% increments.

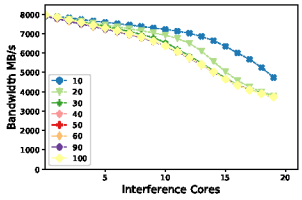


(a) Ice Lake - Read Cont.

(b) Cascade Lake - Read Cont.



(c) Ice Lake - Write Cont.



(d) Cascade Lake - Write Cont.

Figure 10: Isolation guarantees to victim synthetic benchmark from interfering cores running memory intensive synthetic benchmarks.

Figure 10a showcases the result on our Ice Lake platform when the victim core and the co-runners are running DRAM-Bombs performing reads. As mentioned earlier, reads are throttled more than writes in the current implementations on MBA. Hence, the maximum degree of protection to the victim core is observed in Figure 10a. With all contenders throttled at maximum level, the performance decline of 80% without any control has dropped to only 36%. We see a similar result on our Cascade Lake machine. When the 19 co-runners are set to 10%, the victim cores bandwidth drops by 20% compared to 40% earlier without bandwidth restrictions.

However, in most cases, limiting bandwidth even with few contending cores is not restrictive enough to protect the critical task. To provide better protection, we need to decrease further the amount of bandwidth extracted by each core at 10% and other settings. The manual mentions the possibility of calibration settings in the BIOS. But we ascertained that the current version of the BIOS we are using does not support it, despite it being the latest firmware available to us at the time of the writing.

In summary, a notable shortcoming is not being able to throttle reads versus writing separately. Figure 8 depicts how the application under analysis suffers different performance degradation when contending cores perform loads versus stores. Also, the number of available throttling levels (8 on Cascade Lake and 10 on Ice Lake) is limited in number. This lack of fine granularity between levels prevents users from providing precise bandwidth controls to their applications. Even if calibration through the BIOS restricts the bandwidth for each throttling level, there is a chance at runtime this might be a significant limitation, especially when the set of applications running has drastically different load characteristics.

With the current results, it is not viable to use MBA by itself to restrict the non-critical cores to provide temporal isolation to high criticality tasks, and software-based approaches might still be more suitable for real-time workloads.

	Write - Max TL	Write - No TL	Read - Max TL	Read - No TL
Ice Lake	96%	96%	108%	93%
Cascade Lake	80%	81%	92%	75%

Table 5: Percentage of memory transcation reported by MBM compared to the theoretical value.

6.4 How Accurate are MBM Counters?

A plethora of research works have tracked the number of LLC misses of an application in isolation to compute the rate of memory transactions [30, 67]. To avoid doing so, certain new platforms include counters to track the requests sent to the DRAM subsystem [49]. MBM monitors per-core memory transactions to the DRAM sub-systems. It can sepearte traffic between local and remote sockets. In this paper, our focus is on local DRAM sub-syetem as we use one socket and force all main memory transactions to be completed by DRAM on the same node.

The findings of MBM are summarized in Figure 11. The x-axis tracks the progress of the application in seconds. The target application is run for 5 seconds and monitored every 1μs on our Ice Lake machine and every 1s on the Cascade Lake platform. The massive difference in the monitoring interval is due to an improvement in the granularity of measurements between the Cascade Lake platform and the Ice Lake machine. All eight graphs report the cumulative number of memory requests to the DRAM sub-system

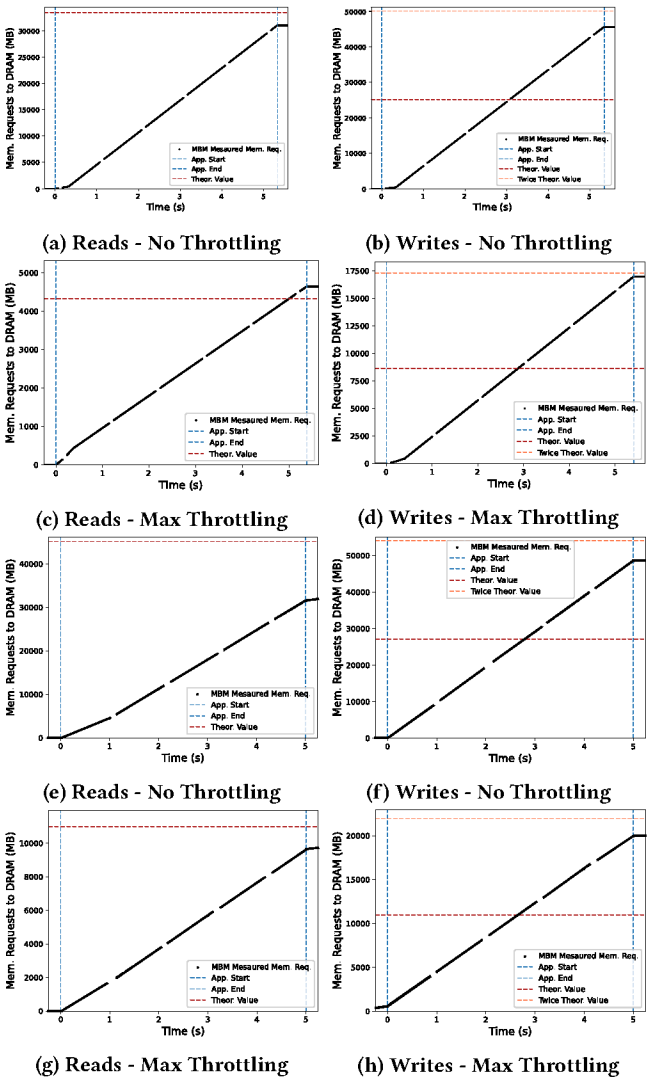


Figure 11: Comparing MBM results for Ice Lake (a-d) and Cascade Lake (e-h) under different throttling conditions.

on the y-axis. The goal is to verify the results reported by MBM compared to the synthetic benchmark on both micro-architectures.

We plot the theoretical data reported by the synthetic benchmark (horizontal flat line). The vertical lines represent the application's beginning and ending. The initial slope of the curve is slower than the trend in the remainder of the plot. Also, even though the loop in the synthetic benchmark executes for five seconds, all of these eight graphs show longer runtimes. These observations can be attributed to the additional time and bandwidth needed to set up the application. Table 5 summarizes the results for the percentage of data transferred from the DRAM sub-system as reported by MBM compared to the theoretical value. The figures also show that when the benchmarks are throttled more, the data requests completed decrease in number because the termination of the application is time-triggered.

There appear to be non-negligible discrepancies between the theoretical values expected in our synthetic benchmark and the value reported by MBM. There is no immediate justification for the under-accounting observed in all figures except Figure 11c. Profiling the application for the precise memory bandwidth has advantages as shown in previous works [49]. Hence, these counters need to be accurate. MBM uses a single counter to accumulate main memory read and write transactions. This makes characterizing the impact of an unknown application even harder as read-intensive applications have a rather different impact on the available system bandwidth compared to write-intensive applications.

In short, even though the monitoring interval granularity has changed significantly over one generation, further refinement is needed for MBM to become an asset that can be confidently used by the real-time community.

7 CONCLUSION

We analyzed the applicability of the Intel RDT framework for real-time applications. The implementation of these features is changing rapidly over processor generations. Our evaluation indicates that RDT management and monitoring do not always behave as expected. Therefore, we currently caution against the indiscriminate reliance on these features for real-time system consolidation. Instead, we encourage the reuse of our methodology to verify their correctness on the considered platform experimentally. Even with CAT, the most mature feature, there is a drastic difference in isolation guarantees achieved by static partitioning between the two generations of processors we considered. The monitoring tools also need to be further refined to become interpretable enough to be employed by the real-time community. Both CMT and MBM readings cannot be wholly trusted, as neither accurately measured the resource utilization of the synthetic benchmarks under observation. We further invite the community to verify or contrast our findings as newer generations of RDT-enabled platforms are released.

ACKNOWLEDGMENTS

The National Science Foundation (NSF) under the grant number CCF-2008799 provided support for the work presented in this paper. We also thank the Red Hat Collaboratory and the partners in the Massachusetts Open Cloud Alliance. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. 2002. Analysis of a Reservation-Based Feedback Scheduler. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.* IEEE, 71–80.

[2] Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. 2020. Dynamic Memory Bandwidth Allocation for Real-Time GPU-based SoC Platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3348–3360.

[3] Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I Davis. 2014. Outstanding Paper: Evaluation of Cache Partitioning for Hard Real-Time Systems. In *2014 26th Euromicro Conference on Real-Time Systems.* IEEE, 15–26.

[4] The Linux Kernel Archives. 2001. NO_HZ: Reducing Scheduling-Clock Ticks. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt. Accessed on 04.12.2022.

[5] Arm. 2018-2020. Arm Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring(MPAM), for Armv8-A. Accessed

on 10.16.2020.
[6] Arm. 2020. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. Accessed on 10.16.2021.
[7] Michael Bechtel and Heechul Yun. 2019. Denial-of-service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 357–367.
[8] Michael Bechtel and Heechul Yun. 2021. Memory-Aware Denial-of-Service Attacks on Shared Cache in Multicore Real-Time Systems. *IEEE Trans. Comput.* (2021).
[9] Dai Bui, Edward A. Lee, Isaac Liu, Hiren Patel, and Jan Reineke. 2011. Temporal Isolation on Multiprocessing Architectures. In *Design Automation Conference (DAC)*. 274 – 279. http://chess.eecs.berkeley.edu/pubs/839.html
[10] Intel Corporation. 2015. Intel® Resource Director Technology (Intel® RDT) Framework. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html. Accessed on 03.09.2019.
[11] Intel Corporation. 2019. Welcome to the intel-cmt-cat Wiki, https://github.com/intel/intel-cmt-cat/wiki. Accessed on 01.23.2022.
[12] Cédric Courtaud, Julien Sopena, Gilles Muller, and Daniel Gracia Pérez. 2019. Improving Prediction Accuracy of Memory Interferences for Multicore Platforms. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 246–259.
[13] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2020. BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 364–375. https://doi.org/10.1109/RTAS48715.2020.00011
[14] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2020. Bru: Bandwidth regulation unit for real-time multicore processors. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 364–375.
[15] Farzad Farshchi, Prathap Kumar Valsan, Renato Mancuso, and Heechul Yun. 2018. Deterministic Memory Abstraction and Supporting Multicore System Architecture. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)* (Dagstuhl, Germany) *(Leibniz International Proceedings in Informatics (LIPIcs))*, Sebastian Altmeyer (Ed.), Vol. 106. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Barcelona, Spain, 1:1–1:25. https://doi.org/10.4230/LIPIcs.ECRTS.2018.1
[16] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.
[17] Andrei Frumusanu. 2021. Intel 3rd Gen Xeon Scalable (Ice Lake Sp) review: Generationally Big, competitively small. https://www.anandtech.com/show/16594/intel-3rd-gen-xeon-scalable-review/4
[18] Golsana Ghaemi, Dharmesh Tarapore, and Renato Mancuso. 2021. Governing with Insights: Towards Profile-Driven Cache Management of Black-Box Applications. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[19] Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. 2021. DNA: Dynamic Resource Allocation for Soft Real-Time Multicore Systems. In *2021 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 196–209.
[20] Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. 2014. WCET Analysis with MRU Cache: Challenging LRU for Predictability. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 1–26.
[21] Red Hat. 2011. Isolating CPUs Using Tuned-Profiles-Realtime. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/isolating_cpus_using_tuned-profiles-realtime. Accessed on 01.23.2019.
[22] Herdrich, Andrew J. and Cornu, Marcel and Abbasi, Khawar Munir. 2019. Introduction to Memory Bandwidth Allocation. *Data Center Documentation* (March 2019). https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-allocation Accessed on 01.23.2021.
[23] Intel Cloud Technology. 2017. *Are Noisy Neighbors in Your Data Center Keeping You Up at Night?* Technical Report. Accessed on 08.11.2019.
[24] Author Andi Kleen Intel Corporation. 2009. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M, 3-180 CPUID reference. Accessed on 01.23.2022.
[25] IntelCorporation. 2016. *Increasing Platform Determinism with Platform Quality of Service for the Data Plane Development Kit.* 8–9 pages.
[26] IntelCorporation. 2019. *Intel 64 and IA-32 Architectures Software Developer's Manual* (volume 3 ed.). 17–64–17–68 pages.
[27] IntelCorporation. 2019. *Intel® Resource Director Technology (Intel® RDT) on 2nd Generation Intel® Xeon® Scalable Processors Reference Manual.* 4–24 pages.
[28] IntelCorporation. 2021. *Intel® Architecture Instruction Set Extensions and Future Features.* 10–2–10–4 pages.
[29] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *2015 Euromicro Conference on Digital System Design*. IEEE, 629–636.
[30] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014. Bounding Memory Interference Delay in COTS-based Multi-Core Systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 145–154.
[31] Namhoon Kim, Jeremy P Erickson, and James H Anderson. 2014. Mixed-criticality on Multicore (MC2): A Status Report. *OSPERT 2014* (2014), 45.
[32] NG Chetan Kumar, Sudhanshu Vyas, Ron K Cytron, Christopher D Gill, Joseph Zambreno, and Phillip H Jones. 2014. Cache Design for Mixed Criticality Real-Time Systems. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE, 513–516.
[33] Linux. 2014. Performance Analysis Tools for Linux. Accessed on 01.23.2022.
[34] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. Catalyst: Defeating Last-Level Cache Side channel Attacks in Cloud Computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 406–418.
[35] Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. 2022. A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms. *IEEE Access* 10 (2022), 21853–21882. https://doi.org/10.1109/ACCESS.2022.3151891
[36] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, et al. 2015. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 131–143.
[37] Cláudio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. 2016. A Closer Look into the AER Model. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 1–8.
[38] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. 2018. *A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems.* Technical Report. Verimag Research Report TR-2018-9 (Technical Report).
[39] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. 2013. Real-Time Cache Management Framework for Multi-Core Architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 45–54. https://doi.org/10.1109/RTAS.2013.6531078
[40] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. 2015. WCET (m) Estimation in Multi-Core Systems using Single Core Equivalence. In *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 174–183.
[41] Renato Mancuso, Heechul Yun, and Isabelle Puaut. 2019. Impact of DM-LRU on WCET: A Static Analysis Approach. *Leibniz international proceedings in informatics* 133 (2019).
[42] Clémentine Maurice, Nicolas le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse engineering Intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*. Springer, 48–65.
[43] Thomas Moscibroda and Onur Mutlu. 2007. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security Symposium*. USENIX.
[44] Marco Pagani, Enrico Rossi, Alessandro Biondi, Mauro Marinoni, Giuseppe Lipari, and Giorgio Buttazzo. 2019. A Bandwidth Reservation Mechanism for AXI-based Hardware Accelerators on FPGAs. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[45] Linux Man Pages. 2004. numactl - Control NUMA policy for processes or shared memory. https://linux.die.net/man/8/numactl. Accessed on 04.19.2019.
[46] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. 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*. 1–16.
[47] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A predictable execution model for COTS-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 269–279.
[48] Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russell B Kegley, Dennis R Perlman, Greg Arundale, et al. 2016. Real-time Computing on Multicore Processors. *Computer* 49, 9 (2016), 69–77.
[49] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. 2020. E-WarP: A System-Wide Framework for Memory Bandwidth Profiling and Management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 345–357.
[50] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. 2022. Profile-driven memory bandwidth management for accelerators and CPUs in QoS-enabled platforms. *Real-Time Systems* (Dec. 2022). https://doi.org/10.1007/s11241-022-09382-x
[51] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. 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*. ACM, 62–75.
[52] Noriaki Suzuki, Hyoseung Kim, Dionisio De Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. 2013. Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses. In *2013 IEEE 16th International*

Conference on Computational Science and Engineering. IEEE, 685–692.

[53] Taylor IoT Kidd. 2014. Power Management States: P-States, C-States, and Package C-States. *Intel® Xeon Phi™ Processor Documentation* (April 2014). https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states

[54] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.

[55] Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, et al. 2010. Merasa: Multicore Execution of Hard Real-Time applications Supporting Analyzability. *IEEE Micro* 30, 5 (2010), 66–75.

[56] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. 2016. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2016), 1–28.

[57] Prathap Kumar Valsan and Heechul Yun. 2015. MEDUSA: A Pedictable and High-Performance DRAM Controller for Multicore based Embedded Systems. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*. IEEE, 86–93.

[58] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. 2016. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–12.

[59] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. 2013. Outstanding Paper Award: Making Shared Caches more Predictable on Multicore Platforms. In *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 157–167.

[60] 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 *Proceedings of the 48th International Conference on Parallel Processing*. 1–12.

[61] Meng Xu, Robert Gifford, and Linh Thi Xuan Phan. 2019. Holistic Multi-Resource Allocation for Multicore Real-Time Virtualization. In *Proceedings of the 56th Annual Design Automation Conference (DAC)*. IEEE, 1–6.

[62] Meng Xu, Linh Thi Xuan Phan, Hyon-Young Choi, and Insup Lee. 2016. Analysis and Implementation of Global Preemptive Fixed-Priority Scheduling with Dynamic Cache Allocation. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–12.

[63] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. 2017. vCAT: Dynamic Cache Management using CAT Virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 211–222.

[64] Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. 2016. BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms. *IEEE Trans. Comput.* 66, 7 (2016), 1247–1252.

[65] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 155–166.

[66] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. 2015. Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems. In *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 184–195.

[67] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. 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)*. IEEE, 55–64.

[68] Yanqi Zhou and David Wentzlaff. 2016. MITTS: Memory Inter-Arrival Time Traffic Shaping. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 532–544.