BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors

Farzad Farshchi
University of Kansas
farshchi@ku.edu

Qijing Huang
University of California, Berkeley
qijing.huang@berkeley.edu

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

***Abstract*—Poor time-predictability of the multicore processors is a well-known issue that hinders their adoption in the real-time systems due to contention in the shared memory resources. In this paper, we present the Bandwidth Regulation Unit (BRU), a drop-in hardware module that enables per-core memory bandwidth regulation at fine-grained time intervals. Additionally, BRU has the ability to regulate the memory access bandwidth of multiple cores collectively to improve bandwidth utilization. Besides eliminating the overhead of software regulation methods, our evaluation results using SD-VBS and synthetic benchmarks show that BRU improves time-predictability of real-time tasks, while it lets the best-effort tasks to better utilize the memory system bandwidth. In addition, we have synthesized our design for a 7nm technology node and show that the chip area overhead of BRU is negligible.**

***Index Terms*—Bandwidth Regulation, Multicore Processors, RISC-V**

I. Introduction

In recent years, high performance multicore processors are increasingly demanded for many safety-critical real-time applications in automotive and the aviation industries. However, execution time variations caused by inter-core interference in multicore processors make their adoption in such applications challenging. The major contributors to inter-core interference are shared hardware resources such as shared caches and DRAM that can be accessed concurrently by multiple cores, which results in unpredictable memory access delays.

The poor timing predictability in multicore is a serious problem especially for safety-critical systems such as avionics that often require evidence of bounded execution timing [1]. A common industry practice is to disable all but one core (known as the "one-out-of-m" problem [2]) as recommended by the Federal Aviation Administration (FAA) for certification of multicore based avionics [1], but it obviously wastes computing capabilities of multicore processors.

There have been many proposals to bound the inter-core interference in multi-core processors, which we categorize as software- and hardware-based solutions. Software-based solutions are typically implemented at the OS or hypervisor level and apply various resource partitioning and access control schemes utilizing hardware features available in COTS processors, such as MMU [3]–[6], hardware performance counters [7], [8] and cache partitioning capabilities [9], [10]. However, due to the black-box nature of COTS hardware, the degree of isolation that can be achieved by these software solutions are fundamentally limited [11], [12]. Furthermore, they often incur considerable performance impact and suffer from high overhead.

On the other hand, hardware based solutions range from proposals to design new memory components such as caches [13]–[15] and DRAM controllers [16]–[19] to a completely new processor and memory system architecture targeted at real-time systems [20]–[23]. When it comes to average performance, however, these architectures that are designed specifically for the real-time applications are difficult to compete with COTS processors. Due to the high development cost of making a new chip, manufacturers tend to target high production volume and it is hard to justify processors that are only suitable for real-time applications. The path to modify the memory components in COTS processors has its own issues. Verification and validation of hardware is a costly and time-consuming task, especially for memory components that deal with complex issues such as cache coherency and memory consistency [24].

In this paper, we propose Bandwidth Regulation Unit (BRU), a hardware unit that enables bounding inter-core interference in the shared memory hierarchy by regulating memory bandwidth at the core-level. At its baseline design, BRU does not modify any memory component and can be dropped in existing multicore processor designs seamlessly. Unlike prior software-based memory bandwidth regulation approaches [7], [8], which often incur high software overhead (e.g., interrupt handling), BRU is a hardware unit and thus incurs no software overhead at run-time. Furthermore, it enables a cycle-granularity fine-grained bandwidth regulation capability compared to the millisecond granularity regulation capabilities in prior software-based regulation mechanisms [7], [8]. In addition, BRU supports a domain based regulation scheme where each domain can be composed of one or more cores.

We implement the BRU in an open-source out-of-order multicore processor [25] using the FireSim simulator [26] on the Amazon FPGA cloud. We conducted a set of experiments using both synthetic benchmarks and SD-VBS [27] benchmarks to evaluate its effectiveness in improving time predictability of real-time tasks and overall bandwidth utilization. We find that BRU offers superior regulation performance over prior software-based coarse-grained bandwidth regulators at a very low added hardware complexity.

Lastly, we synthesize a BRU augmented processor design in a 7nm technology node and analyze the area and timing overhead. Our analysis results show that BRU causes no timing overhead and its space overhead is negligible.

We make the following contributions in this paper:

- We present Bandwidth Regulation Unit (BRU), a cycle-granularity hardware based memory bandwidth regulator for multicore-based real-time systems.
- We implement BRU in an open-source multicore design in an FPGA-accelerated full-system simulator and evaluate its performance using a set of synthetic and real-world benchmarks, showing its feasibility and effectiveness.
- We synthesize the design with a 7nm technology node and present area and timing overhead analysis, showing negligible overhead of using BRU.

The remainder of the paper is organized as follows. Section II describes the necessary background. In Section III, we explain the BRU architecture and its register interface. Section IV describe the implantation details of BRU, including aspects which are specific to the multicore platform our prototype is based on. Section V presents evaluation results. We review the related work in Section VI and conclude in Section VII.

II. Background

We use the Rocket Chip generator [28] to implement the BRU. Although the design of BRU is not fundamentally limited to a specific implementation, we would like to build the necessary background on the platform we use to better describe our design in the following sections.

Rocket Chip is an open-source System-on-Chip (SoC) generator that implements the open RISC-V instruction set architecture (ISA) [29]. It can generate both in-order and out-of-order processors, which are capable of running Linux. The out-of-order processors are supported through the Berkeley Out-of-Order Machine (BOOM) [25] project. The processor designs are written in the Chisel hardware construction language [30] and are taped out multiple times. Rocket Chip is also used as the basis for building several commercial SoCs and IP cores [31].

Rocket Chip uses TileLink protocol for on-chip communication and accessing the shared memory. Since knowing the basics of TileLink is necessary for understanding the details of our current implementation of BRU, we briefly describe the specification in the following.

A. TileLink

TileLink is an interconnect standard for on-chip communication, which enables coherent access to the shared memory and peripheral devices [32]. TileLink standard defines three protocol conformance levels: TileLink Uncached Lightweight (TL-UL), TileLink Uncached Heavyweight (TL-UH), and TileLink Cached (TL-C). TL-C is the most complete protocol that allows managing and transferring cached data. Thus, we focus on describing TL-C for the rest of this section.

TileLink standard is defined by a set of *operations* that are allowed to be performed on a shared address range. A TileLink *operation* is carried out by transferring *messages* across point-to-point *channels*. These *channels* form a *link* between a master *agent* and a slave *agent* [32]. A TL-C link is comprised of five channels: A, B, C, D, and E. The channels are strictly prioritized from A (lowest priority) to E (highest priority). Each channel uses a pair of ready and valid signals for handshaking and flow control.

Transfer messages. TileLink allows the design of the interconnect protocol to be separated from the cache coherence protocol implementation. It defines a set of messages to govern transferring cached data and permission across the chip. These are known as *transfer* messages. A coherence protocol implementation (e.g. MESI [33]) uses these messages to alter a cache line state and transfer permission and data. We describe some of the transfer messages by showing the message flow for two fundamental templates that enable coherent access to the cached memory [34].

First template. Figure 1 shows the message flow in which Cache X attempts to get data and read/write permission on a cache line by sending an Acquire message to a coherence manager agent (or a manager for short) on Channel A. Once the manager receives the Acquire message, it sends a Probe message to Cache Y to query or downgrade the permission that Cache Y owns on the cache line. If needed, Cache Y updates the permission on the cache line and sends a ProbeAck response to the manager on Channel C. If Cache Y owns a dirty copy of the cached data too, it responds with ProbeAckData message which carries the payload.

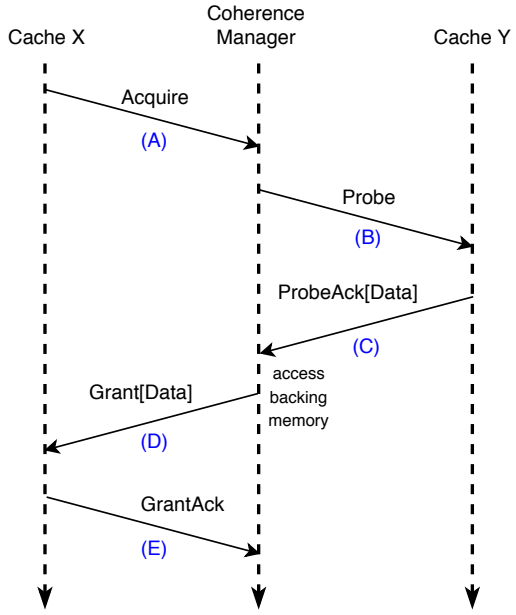


Fig. 1. Cache X sends an Acquire message to the coherence manager then, the manager probes Cache Y. The channel names are shown in parentheses. Adapted from [34].

Upon receiving ProbeAck or ProbeAckData, the manager

accesses the backing memory if required. Next, the manager responds with a Grant or a GrantData to give the required permission and/or data to Cache X. Finally, Cache X sends a GrantAck message to the manager to indicate that the operation is finished.

Second template. Figure 2 shows the message flow in which a cache voluntarily releases permission on a block. This typically happens when a cache performs a dirty eviction and it has to do a writeback. Upon receiving ReleaseData from the cache, a manager writes the dirty data to the backing memory and sends a ReleaseAck response to the cache. As we can see, ReleaseData is transferred over Channel C which is the same channel used for transferring ProbeAck and ProbeAckData.

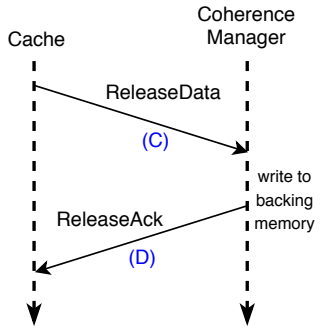


Fig. 2. A cache voluntary releases permission on a cache line. The channel names are shown in parentheses. Adapted from [34].

Access messages. In addition to the messages described above, TileLink defines *access* messages to read/write the uncachable memory addresses. These addresses include the memory-mapped registers of I/O devices. Get is an example of the access messages which is used to read an uncachable memory address. In addition, a Get message is used by the instruction cache in the Rocket Chip to fetch the instructions.

III. BRU Architecture

We start with defining the architecture of our proposed design. BRU is a memory traffic regulator that regulates memory traffic from the cores to the shared memory in a multicore processor. Figure 3 shows a simplified view of a typical multicore processor with a memory system shared between multiple cores. Each core has its own private instruction and data caches. It is also possible for a core to have multiple levels of private caches. On a miss in the outermost private cache, the memory request is directed to the shared memory system. BRU is placed where the private caches are connected to the shared memory and regulate memory traffic that goes to the shared memory.

Because BRU is directly connected to the cores, it is capable of counting the number of memory accesses per core and controlling the flow of the memory traffic for each core independently. This eliminates the need for adding metadata to the bus and the LLC to transfer and store the information about which core has requested the memory access. Note that BRU has equal number of slave and master ports and it does not reroute or arbitrate the traffic. For example, in Figure 3, all the traffic from slave port S0 is routed to master port M0 and similarly the traffic from slave port S1 is routed to master port M1.

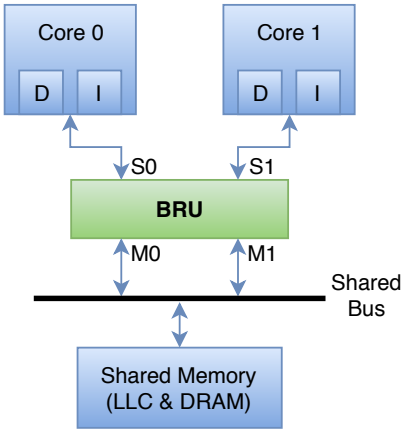


Fig. 3. A simplified view of a multicore processor with shared memory resources. BRU regulates per-core bandwidth at source.

We choose to regulate the maximum bandwidth in our design. This is done by limiting the maximum number of accesses to the shared memory in fixed time intervals. In our design, once the number of memory accesses for a domain—a regulation principal, which can be composed of one or more cores—reaches a programmable maximum, no more accesses are allowed to be issued to the shared memory by the cores assigned to that domain until the current regulation period T is finished. The memory access budget b is then replenished for all domains on the beginning of the next period. The period T is defined in terms of clock cycles, and the budget b is defined in terms of the number of memory access transactions.

A. Access Bandwidth Regulation Interface

Figure 4 shows the registers of an BRU instance for a quad-core processor. In this figure, the BRU is configured to support two regulation domains—one domain for cores 0, 1, and 2, and another domain for core 3. At the high level, four groups of registers—Window Registers, Processor Control and Assignment Registers (PCAR), Regulated Domain Registers (RDR)—are collectively responsible for creating domains and setting their bandwidth regulation parameters. Some of these registers are mapped to the memory address space so that the processor can read or write to them, which are indicated by the brackets around their names in Figure 4.

Domain control. BRU's bandwidth regulation is performed on a *domain* granularity. A domain is composed of one or more cores, and can be created by configuring each core's two domain related registers: `Domain ID Register (DIR)` and `Enable Bandwidth Regulation Register (EBWR)`. The DIR register is used to map a core to a domain ID, and the EBWR is used to enable or disable

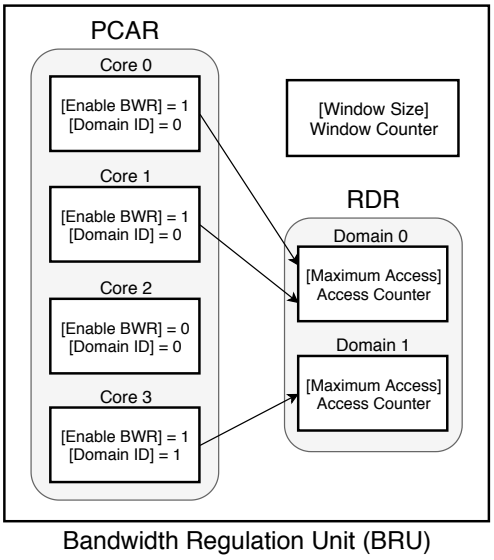


Fig. 4. BRU registers. cores 0 and 1 are assigned to domain 0, and core 3 is assigned to domain 1. Bandwidth regulation is not enabled for core 2.

the association. For example, in Figure 4, cores 0, 1, and 2 are associate to domain 0 but only the cores 0 and 1 are enabled for bandwidth regulation. On the other hand, core 3 is assigned to domain 1 and its bandwidth regulation is enabled. The maximum number of domains is a configurable hardware parameter, which should be decided before making the chip as each domain needs some hardware resources.

Budget control. The memory *bandwidth budget* of a domain can be configured by updating two per-domain registers: `Maximum Access Register (MAR)` and `Access Counter (AC)`. AC is incremented by one on each access to the shared memory by the cores assigned to the domain. MAR is programmed by the software to set the maximum number of access budget to the shared memory allowed in a time window. The bandwidth *regulation period* is globally applied to all domains and is configured by updating the `Window Size Register (WSR)` register at the clock cycle granularity. WSR is compared against a cycle counter at every cycle, and when a period is completed, the domain's access counter is cleared to replenish the memory bandwidth budget for the next period.

B. Write Bandwidth Regulation Interface

Our basic bandwidth regulation mechanism described above equally account both read and write accesses. In other words, write and read accesses are regulated with respect to a single user-defined bandwidth budget. However, prior works have shown that on some COTS multicore processors, the write accesses, particularly cache writeback traffics, can have a more severe effect than read accesses [11], [12]. In order to regulate the write traffic separately, BRU adds two new registers to each domain: `Write Access Counter (WAC)` and `Maximum Write Access Register (MWAR)`. Similar to the AC and WAR registers in the baseline memory bandwidth regulation mechanism, WAC is incremented by one on each write to the shared memory and MWAR determines the maximum number of writes over the regulation period. Once WAC reaches the value programmed in MWAR, writes are throttled until the write bandwidth budget is replenished at the beginning of the next period. We discuss the implementation of write throttling in Section IV.

IV. Implementation

In this section, we describe implementation details that are specific to the TileLink interconnection network and the Rocket Chip SoC, on which our work is based.

A. Access Regulation

Figure 5 shows an example dual-core Rocket Chip SoC with a BRU. In this setup, each core has private instruction and data caches, which are connected to a shared bus, through which the rest of the shared memory hierarchy is connected. The BRU sits between the core's private caches and the shared bus so that it can regulate access to the shared memory hierarchy. Specifically, the BRU is connected to the core private TL-C links (see Section II). In addition, the memory-mapped registers of BRU are accessed through a TL-UL link that is connected to the periphery bus.

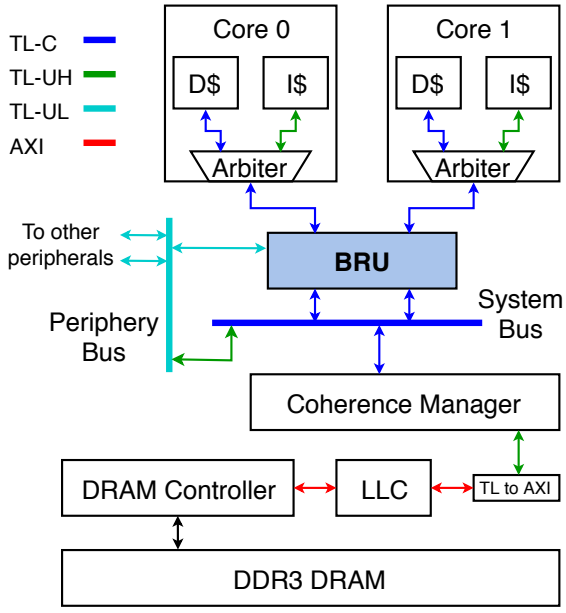


Fig. 5. A dual-core Rocket Chip SoC with BRU.

Let us begin by explaining how the private caches access the shared memory. As we mentioned in Section II, a data cache sends an Acquire message over Channel A, if it does not own the permission or data to preform read or write on a cache line due to a cache miss. On the other hand, in case of an instruction cache miss, a Get message is sent over Channel A. Both Acquire and Get messages are transferred

over Channel A of the TL-C link. Therefore, by throttling this channel, we can control a core's access to the first shared memory component in the system's memory hierarchy, which is the system bus in this example.

TileLink uses a pair of ready and valid signals on each channel for handshaking. A beat[1] flows in the direction of the channel when both ready and valid signals are high on the rising edge of the clock. Figure 6 shows the logic used to throttle Channel A. When *throttle*$_i$ is high, the Channel A corresponding to core *i* is throttled. The other signals of Channel A plus the signals of channels B, C, D, and E pass through the BRU without any alternations. We show how the rest of the BRU logic drives *throttle*$_i$ to implement the desired behavior in the following.

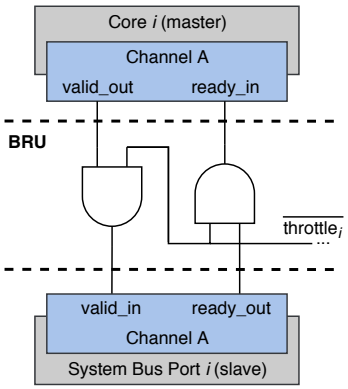


Fig. 6. The logic which controls the flow of messages on Channel A. Boundaries BRU are denoted with dashed lines.

Algorithm 1 shows the high-level pseudo-code of BRU, which is performed at rising edges of the clock. In this algorithm n, $memBase$, and $A(i)$ are the number of cores, the base address of the main memory, and the handle for Channel A corresponding to core *i*, respectively. The other parameters represent the registers defined in Section III.

In essence, the algorithm performs two main tasks. It updates *Window Counter* and *Access Counters* and drives *throttle*$_i$. These are done by iterating on registers and signals belonging to each core in the for loop in line 7. Recalling from Section III, $DomainIDs$ is the array of the *Domain ID Registers (DIR)* which hold the IDs of domains each core is assigned to. The value of this register is used to index the arrays $AccessCounters$ and $MaximumAccesses$.

Let us assume that the value stored in core *i*'s DIR is d. If $AccessCounters(d)$ is greater than or equal to $MaximumAccesses(d)$ (line 10), core *i* is throttled. Additionally, core *i*'s *Enable Bandwidth Regulation Register (EBWR)* must be set to '1' to throttle core *i*. Note that multiple cores can be assigned to the same domain.

Similarly, $DomainIDs$ is used to index $AccessCounters$ array. If an Acquire or a Get message is transferred over

[1]A beat is an individual data transfer in a burst.

Algorithm 1: Bandwidth Regulation for Cache Misses

```
1  if WindowCounter ≥ WindowSize then
2  |  WindowCounter = 0
3  |  foreach c in AccessCounters do c = 0
4  else
5  |  WindowCounter++
6  end
7  for i ← 0 to n − 1 do
8  |  throttle(i) = 0
   |  // if enabled for core i
9  |  if EnableBWRs(i) then
10 |  |  if AccessCounters(DomainIDs(i)) ≥
   |  |   MaximumAccesses(DomainIDs(i)) then
   |  |  |  // throttle Channel A of core i
11 |  |  |  throttle(i) = 1
12 |  |  end
   |  |  // is message instruction fetch?
13 |  |  isInst = A(i).isGet ∧ A(i).addr ≥ memBase
   |  |  // if Acquire or intsruction fetch
14 |  |  if A(i).isAcquire ∨ isInst then
15 |  |  |  AccessCounters(DomainIDs(i))++
16 |  |  end
17 |  end
18 end
```

Channel $A(i)$, the *Access Counter* which core *i* is assigned to is incremented by one (line 15). When the *Window Counter* reaches the *Window Size*, all the *Access Counters* are set to zero and the bandwidth budgets of all domains are replenished (line 3).

As discussed in Section II, the Get message is used, in addition to fetching instructions, to access the memory-mapped I/O registers. However, the I/O registers are mapped to the addresses below $memBase$. To make sure that we only count the instruction fetch accesses, the address of the Get message is checked against the base address of the main memory. If a memory location accessed by a Get message is stored in the main memory, it is an instruction.

B. Writeback Regulation

We now describe how BRU regulates write traffic—more specifically write-back traffic from the core private caches—to the shared memory (i.e., LLC).

Let us explain how regulating cache misses results in regulating writebacks. There are two types of writebacks to the lower level of the memory from the L1 data caches. The first type is a dirty eviction. A dirty eviction may happen when the cache performs a refill and there is a cache conflict. In such a scenario, a cache line must be evicted to free up space for the refill. If the cache line selected for eviction is dirty, a writeback is carried out to update the backing memory. A refill, in turn, is the result of a cache miss. When the data cache needs to perform a dirty eviction, it sends a ReleaseData message over Channel C (second template in Section II).

The second type of writeback happens when data is shared between two caches. Suppose that Cache X wants to get permission to read/write a cache line (first template in Section II). Then, Cache Y must be probed and if it has a dirty copy of the same cache line, it responds with a ProbeAckData message over Channel C. Upon receiving ProbeAckData, the coherence manager writes the dirty cache line to the backing memory. As we observe, the event that triggers a writeback in both cases is a cache miss. Therefore, controlling the rate of the cache misses, as we do in our baseline configuration, results in regulating the rate of the writeback issuance.

From the explanation above, we can conclude that to regulate the writebacks at a threshold lower than the cache miss regulation threshold, we need to count and throttle ReleaseData and ProbeAckData messages sent to the shared memory system. Counting these messages is not complicated. However, unlike what we did for regulating the cache misses by throttling Channel A, we cannot simply throttle Channel C to regulate the writebacks. As we explained for the first template in Section II, Cache Y must send a ProbeAck message before the coherence manager can respond to Cache X with the permission/data. If we delay the ProbeAck message by throttling Channel C of Cache Y, the response to the request of Cache X is delayed too. This essentially causes undesired interference between cores which are assigned to different domains. We refer to this as inter-domain interference. Note that Cache Y must respond with a ProbeAck even if it does not own a permission on the cache line.

The key takeaway from the above is that to regulate writebacks, we need to find a way to throttle ReleaseData and ProbeAckData messages without inhibiting ProbeAck messages. Figure 7 shows how this can be done by slightly modifying the data cache and sending a signal from BRU to throttle the desired messages. In the data cache, WB unit is responsible for sending ReleaseData and ProbeAckData messages and Prober issues the ProbeAck messages. We have inserted a logic—similar to the one in Figure 6—at the output of the WB unit that throttles writeback messages when *WB throttle*$_i$ is high. Note that there is a *WB throttle* signal for each core. That means BRU can throttle writebacks for each core independently. We kept the modifications to the data cache as minimum as possible and we have only modified 5 lines of code in the data cache module.

Algorithm 2 shows the pseudo-code which extends Algorithm 1 to support writeback regulation. These algorithms are very similar, except that in Algorithm 2, the decision to drive *WB throttle*$_i$ is made by comparing *Write Access Counter (WAC)* and *Maximum Write Access Register (MWAR)*. Moreover, WAC is incremented whenever a ReleaseData or a ProbeAckData message is transferred over Channel C.

Sharing the dirty cache lines. Although we avoid throttling ProbeAck, it is still possible to incur inter-domain interference by throttling ProbeAckData. The mechanism that results in inter-domain interference is similar for both ProbeAckData and ProbeAck, however, ProbeAckData is only issued when a dirty cache line is accessed by a remote cache. Often times, dirty cache lines are shared when two or more cores are working on the same data set. An example of such scenario is when a producer and a consumer are working on the same job but are running on two different cores. In such a case, these collaborating cores should be assigned to the same domain so that the bandwidth is regulated collectively for those cores.

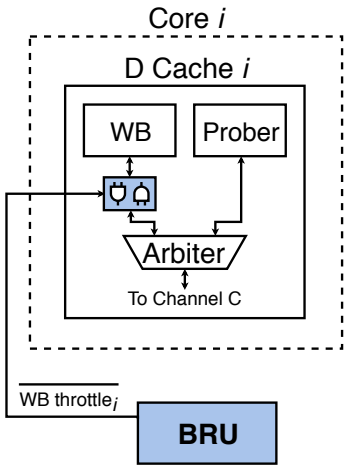


Fig. 7. BRU sends a signal to the data cache to throttle writebacks.

Algorithm 2: Writeback Regulation

```
1  if WindowCounter ≥ WindowSize then
2  |  foreach c in WriteAccessCounters do c = 0
3  end
4  for i ← 0 to n − 1 do
5  |  WB_throttle(i) = 0
6  |  if EnableBWRs(i) then
7  |  |  if WriteAccessCounters(DomainIDs(i)) ≥
   |  |   MaximumWriteAccesses(DomainIDs(i))
   |  |   then
8  |  |  |  WB_throttle(i) = 1
9  |  |  end
10 |  |  if C(i).isReleaseData ∨ C(i).isProbeAckData
   |  |   then
11 |  |  |  WriteAccessCounters(DomainIDs(i))++
12 |  |  end
13 |  end
14 end
```

V. Evaluation

To evaluate the performance of BRU, we utilize FireSim [26]—an FPGA-accelerated full-system simulator. We use FireSim mainly for better accuracy and simulation speed that it offers over the other options such as software simulators. In FireSim, the simulated design is directly derived from the RTL and is implemented on the FPGA. Thus, we can get highly accurate performance results as if the design is fabricated on a chip. Additionally, since FireSim is running on FPGA, it is orders of magnitude faster than the architectural

software simulators such as gem5 [35]. In our experiments, FireSim runs at about 60MHz. As we will see in the rest of this section, this enables us to run real world benchmarks for their entire execution time duration and to run a real-time task for one thousand periods to analyze its WCET behavior.

Note that the approach that FireSim takes to simulate the design is different from FPGA prototyping, which is a common industry practice for early software development before having the chip delivered. The problem with FPGA prototyping is that the processor is clocked at a lower frequency comparing to an ASIC implementation but the DRAM is still fast. This makes FPGA prototyping unsuitable for performance analysis. FireSim uses a special technique to decouple the timing of the simulated design from the host FPGA DRAM to simulate the DRAM access time accurately [36]. As a result, we believe our performance evaluation results in the remainder of this section are realistic.

System Setup. Table I shows the system configuration. The architecture of the SoC is similar to Figure 5 except that it is configured as a quad-core processor. We choose the number of BRU domains to be equal to the number of cores. The L1 data caches are non-blocking with 4 MSHRs each. We configure LLC to have enough MSHRs to handle all the parallel requests issued by the L1 caches ((4 data + 1 instruction) $\times$ 4 = 20). This eliminates the MSHR contention in the LLC [11].

TABLE I
SYSTEM CONFIGURATION

Processor	Quad-core BOOM (RISC-V ISA), 2.13 GHz out-of-order, 1-wide, 3-issue, ROB: 16, LSQ: 8/8
Caches	L1-I/D: 16/16KiB, 4-way, MSHRs: 4 (D), 1 (I) LLC: 2MiB, 8-way, 20 MSHRs, 64-byte lines
System Bus	TileLink, out-of-order completion, round-robin
DRAM Controller	FR-FCFS, open-page policy, scheduler window: 8
DRAM	DDR3-2133, 1 rank, 8 banks, 32KiB row-buffers

For the OS, we use the RISC-V port of the Linux kernel 4.15. We evaluate our design using the San Diego Vision Benchmark Suite (SD-VBS) [27] with CIF input size. Table II shows the average bandwidth utilization of SD-VBS running on our system.[2] Additionally, we use Bandwidth and Latency benchmarks from the IsolBench benchmark suite [11]. *Bandwidth* is a synthetic benchmark that accesses the memory at the cache line strides to generate the maximum memory traffic. It is a memory-intensive program, which we use to create the worst-case memory interference. Bandwidth can be configured to either read or write to the memory. We denote the read and write variants with BwRead and BwWrite, respectively. There is also a periodic variant of BwWrite which we denote with BwWrite-RT. *Latency* is another synthetic benchmark that traverses the nodes of a linked-list with each node located on a separate cache line. This benchmark it designed to be sensitive to the memory access latency.

[2]We omitted the *multi ncut* benchmark due to its long simulation times.

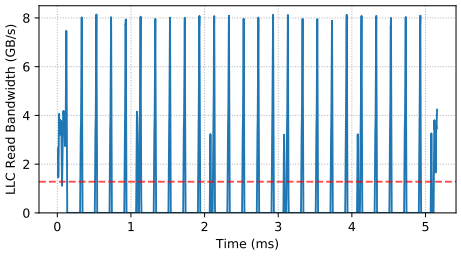
TABLE II
SD-VBS BENCHMARK CHARACTERISTICS (MB/S)

Benchmark	Ave. LLC Read B/W	Ave. LLC Write B/W	Ave. DRAM Read B/W	Ave. DRAM Write B/W
disparity	2806	1165	276	155
localization	142	57	0.32	0.18
mser	1513	420	247	122
sift	602	124	128	66
svm	444	107	0.68	0.56
texture_syn	148	50	20	15
tracking	479	199	61	45

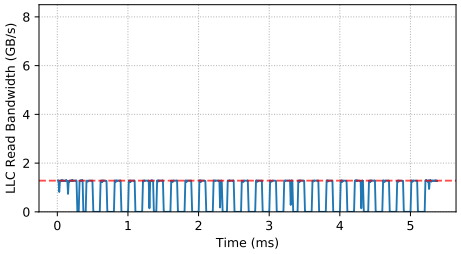
A. *Effect of Regulation Period in Regulation Performance*

In the first set of experiments, we demonstrate the impact of fine-grained bandwidth regulation over a coarse-grained one.

In the first experiment, we use a synthetic BwRead benchmark and configure it to access a 120KiB array every 200μs to resemble an application with short burst accesses. The average bandwidth of this application is equal to 600MB/s (120KiB $\div$ 200μs) and each burst is about 22μs in length. We set the cache miss bandwidth budget at 1280MB/s and run the application once with 1ms and another time with 100ns regulation period (window size). Based on the 2.13GHz clock frequency, these periods are equal to 2.13×10^6 and 213 cycles, respectively.



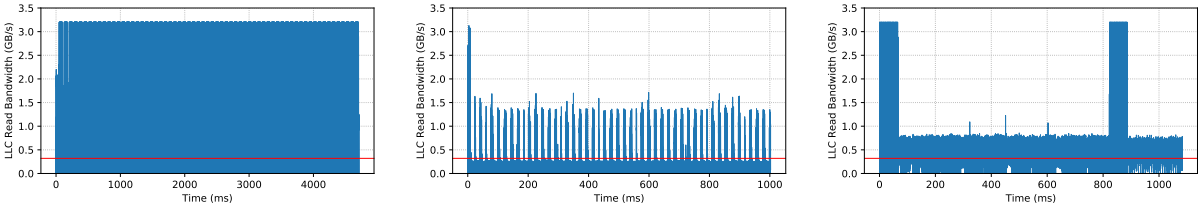
(a) 1ms regulation period.



(b) 100ns regulation period.

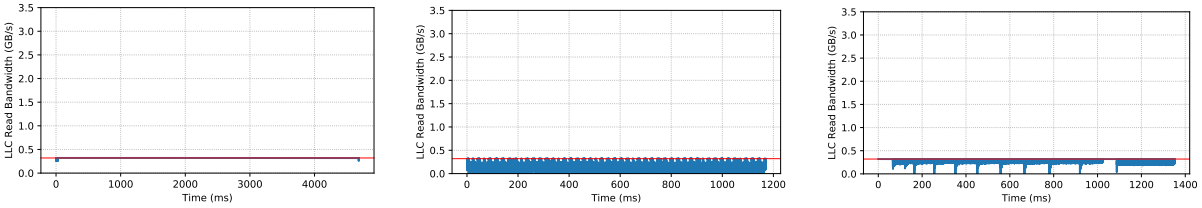
Fig. 8. Synthetic benchmark with burst memory accesses regulated at 1280 MB/s. Measured at 1μs and applied a 10-point (over 10 μs) moving average.

Figure 8 shows the LLC read bandwidth of these two tests. We observe that the memory accesses are not throttled with the 1ms period, however, we see that with the 200ns period, the bursts are capped at 1280MB/s across the 1μs measurement intervals. This experiment can help us better understand how the window-based regulation works. This regulation method guarantees that the average bandwidth *across the regulation period* does not exceed the budget. Since the average memory bandwidth of the application across 1ms is less than



(a) disparity (b) localization (c) svm

Fig. 9. LLC read bandwidth for SD-VBS at 1ms regulation period and 320MB/s budget. Measured at 10μs and applied a 10-point moving average.



(a) disparity (b) localization (c) svm

Fig. 10. LLC read bandwidth for SD-VBS at 200ns regulation period and 320MB/s budget. Measured at 10μs and applied a 10-point moving average.

1280MB/s, it is not throttled in the case of the 1ms regulation period. However the average demand across the length of a burst is much higher (maximum 8GB/s). That is why the memory accesses are throttled when the regulation period is set to 100ns.

In the second experiment, we instead use real-world benchmarks from the SD-VBS suite to further demonstrate the effect of fine-grained regulation. For this experiment, we set the bandwidth budget at 320MB/s, which is less than the unregulated average LLC read bandwidth of most of the SD-VBS benchmarks as can be seen in Table II. We repeat the experiments using two regulation periods: 1ms and 200ns.

Figure 9 and Figure 10 show the results for 1ms and 200ns periods, respectively. We include only three benchmarks due to space limitation. In case of the 1ms, we can see that although the average bandwidth is below 320MB/s the peaks can be as high as 3GB/s. However, for the 100ns period, the bandwidth is capped at 320MB/s. Again, this is because fine-grained regulation can handle bursty memory accesses more evenly distributed over time.

B. Effect of Regulation Period in Protecting Real-time Tasks

In this experiment, we demonstrate the effect of regulation period in protecting the real-time tasks. The basic experiment setup is that we run a real-time task on Core 3, while three best-effort tasks are co-scheduled on cores 0~2. For the real-time task, we use BwWrite-RT, which is configured to access a 4MiB array periodically at every at 4.1ms. The WCET of the real-time task, measured in isolation, is 1.52ms. For the best-effort tasks, we use three instances of *disparity* benchmark from SD-VBS. The three cores for the best-effort tasks are assigned to one regulation domain, which is regulated at 1280MB/s. We ran the real-time task for one thousand periods under four different scenarios and plotted the CDF of the task's observed response times.

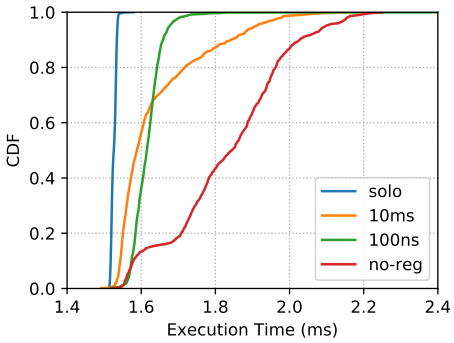


Fig. 11. The performance of BwWrite-RT co-scheduled with disparity co-runners.

Figure 11 shows the results. In *Solo*, the real-time task is running in isolation without the best-effort tasks. In *No-reg*, the real-time task is co-scheduled with the best-effort co-runners but regulation is disabled (i.e., BRU is not used). In both *10ms* and *100ns*, regulation is enabled but at different regulation periods. Note first that without regulation the real-time task's observed response times vary considerably. When BRU is used, the response time variance reduces. It is noteworthy that using a shorter regulation period (100ns) tends to reduce variance better than using a longer one (10ms). This is because memory regulation is more evenly applied by using a shorter regulation period. In general, to effectively protect real-time tasks with regulation, the regulation period should be configured to be significantly shorter than the WCETs of the protected real-time tasks.

C. Effect of Group Bandwidth Regulation

In the following two experiments, we show the effects of using group-based bandwidth regulation to the regulated best-effort tasks and the protected real-time tasks. The basic experiment setup is as follows. In the first experiment we run Latency, which is used as protected real-time task, on Core 3 and configure its working set size (WSS) to be larger than the size of the LLC (i.e DRAM-fitting). We then co-schedule three instances of BwWrite, which is used as best-effort tasks, on cores 0 through 2 and set their WSS to be DRAM-fitting. We regulate the bandwidth of cores 0~2 under two different schemes. In the first scheme we assign the cores to one domain and regulate their memory accesses collectively. In the second scheme, we assign each core to a domain and split the bandwidth budget equally among them. For instance, if the total bandwidth budget is 320MB/s, each core is assigned with 106.6MB/s. We run the experiment under different budget assignments and measure the execution time of Latency.

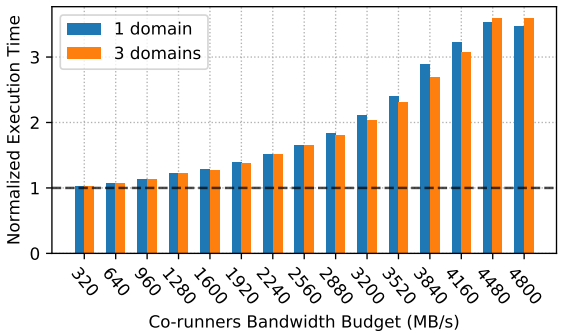


Fig. 12. The performance of Latency versus the total bandwidth budget of BwWrite co-runners.

Figure 12 shows the normalized execution time of Latency, which is normalized to its solo execution time. First, note that the results suggest that there is a trade-off between the deterministic execution time guarantee of the real-time task and the bandwidth budgets given to the best-effort tasks. For tighter guarantee, the smaller bandwidth budget given to the best-effort tasks may be better. Second, we see that for most of the bandwidth budgets, the execution time of Latency in 1-domain regulation is equal to the 3-domain regulation or just slightly above it. This means that group regulation, may come at a small cost for the execution time of the real-time tasks (maximum 7% for 3840MB/s) compared to 3-domain regulation.

In the second experiments, the basic setup is the same but we use three benchmarks from the SD-VBS suite as best-effort tasks instead of the BwWrite benchmark. We choose *disparity*, *mser*, and *texture_synthesis* to be representative of workloads with different levels of memory intensiveness. Having the execution time of the real-time task from Figure 12, we target at maximum slowdown of 1.5x. This gives us the bandwidth budget of 2240MB/s for the best-effort tasks. Similar to the first experiment, Latency is running on Core 3 and the SD-VBS benchmarks are running on cores 0~2. We run the experiment under 1-domain and 3-domain regulation schemes. In the 1-domain scheme the total budget of 2240MB/s is assigned to cores 0~2 and these cores compete for the bandwidth with each other. Conversely, in the 3-domain scheme, each core is assigned with 746.6MB/s and they cannot share the unused bandwidth. Figure 13 shows the execution time of the SD-VBS benchmarks normalized to their solo execution times. As we can see, under the group regulation scheme (i.e. 1-domain) the bandwidth that is not used by texture_synthesis is utilized by *disparity* to improve its performance by 35% compared to 3-domain regulation. Under 3-domain scheme *disparity* does not get benefit from the *texture_synthesis* leftover bandwidth.

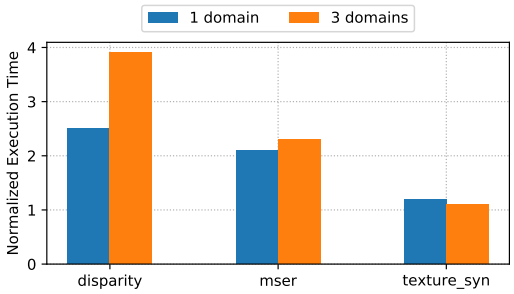


Fig. 13. The performance of three SD-VBS benchmarks under different domain assignment schemes. Normalized to the solo execution time.

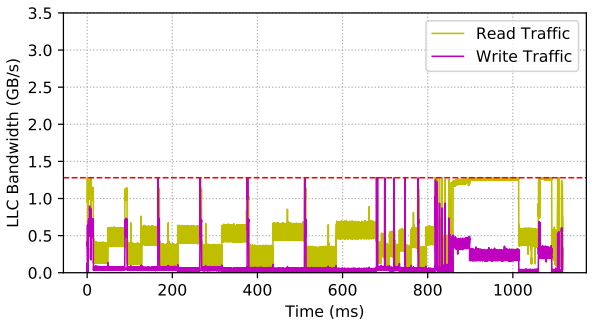
D. Effect of Writeback Regulation

In the experiments above, we have only regulated the cache misses. As we described in Section IV-B, this results in regulating the read and write accesses at the same threshold. In this section we set up experiments to demonstrate how writeback regulation can be used to regulate write accesses at a lower threshold.

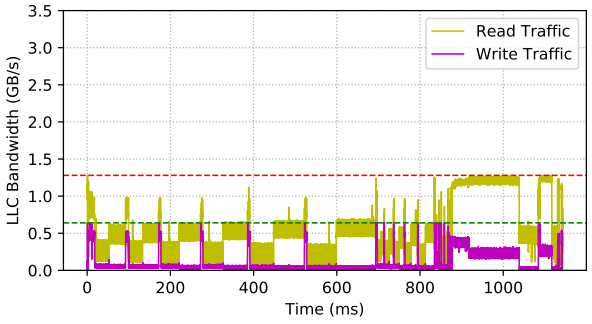
Figure 14(a) shows the result of running *sift* with 1280MB/s cache miss bandwidth budget and no writeback regulation. The regulation period is set to 100ns in this experiment. We see that although we have not set the writeback budget, the write accesses to the LLC are regulated at the same level as the read accesses. We run another test in which we set the writeback bandwidth budget at 640MB/s while maintaining the 1280MB/s budget for the cache misses. The result is show in Figure14(b). We see that using writeback regulation, the write accesses are regulated at 640MB/s while the reads are still regulated at 1280MB/s. This helps to reduce the peak rate at which writes are issued to the shared memory while maintaining the same threshold for the read accesses.

E. Hardware Implementation Overhead

To study the cost of our design in the hardware, we integrate BRU to a multi-core BOOM system, and run synthesis to estimate its area overhead and its effect on the maximum clock frequency. To save on the synthesis run time, we implement BRU on a dual-core processor, yet the trend in the area overhead of BRU remains the same for higher number of



(a) Writeback: no regulation; cache miss: 1280 MB/s.



(b) Writeback: 640 MB/s; cache miss: 1280 MB/s.

Fig. 14. LLC Bandwidth of *sift* under different cache miss and writeback budgets. Measured at 10μs and applied a 10-point moving average.

cores. We use the Cadence Genus synthesis tool with the Hammer [37] automation scripts targeting the ASAP 7nm technology node [38].

Table III shows the post-synthesis chip area breakdown of the dual-core BOOM system. As it can be seen, the area overhead of BRU is pretty small and it only takes less than 0.2% of the total area. Note that the area for SRAM needed for the caches is not included in the total area. Additionally, we synthesized the same design without BRU to examine the effect of the integrating BRU on timing. The results show that BRU has less than 2% impact on the maximum clock frequency. Consequently, both timing and area results show that adding BRU leads to negligible overhead in hardware. We also performed place and route on the dual-core BOOM with integrated BRU. Figure 15 shows the chip layout with BRU circled in red.

TABLE III
DUAL-CORE BOOM CHIP AREA BREAKDOWN

Modules	Area (μm^2)	Ratio
BRU	4,669	0.19%
Boom Core × 2	2,309,681	92.41%
Others (System Bus, Manager, etc.)	184,950	7.40%
Total	2,499,300	100%

VI. RELATED WORKS

Deterministic hardware architectures have been extensively studied in the real-time community. PRET [39], T-CREST

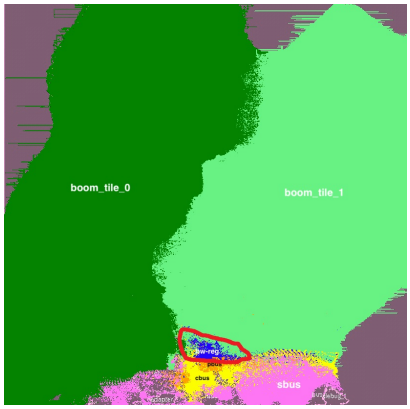


Fig. 15. Dual-core BOOM chip layout with BRU circled in red.

[23], MERASA [22], and CoMPSoC [20] projects have proposed processor architectures and complete systems which are specifically targeted at real-time applications. Additionally, in works such as LEOPARD [21] and Deterministic Memory [40], extensions are added to the bus, the L2 cache, and the DRAM controller to facilitate timing analysis. The architectures that are specifically targeted at real-time applications, however, generally do not perform well when it comes to the average performance and because of the relatively small market size [21], it is difficult to justify the cost of building the such architectures.

There are also challenges involved with adding extensions to the existing hardware. Firstly, validation and verification of new hardware design is a time-consuming and labor-intensive task. Most of these solutions need to redesign existing hardware components. The problem is exacerbated when adding complexity to the already complex and hard to verify algorithms that deal with maintaining memory consistency and coherency. Even with modifying the memory system components, the challenge of coordinating these components at multiple levels of the memory hierarchy still exists. In [41], it is shown that independently enforcing the priority of the requests at each memory resource may not be effective because of the interactions between these resources and the effect of prioritizing in one resource on the others.

In this work, we have chosen a less intrusive approach which, at its baseline design, does not modify any existing hardware components in the processor. There are two recently proposed closely related works. MCCU [24] proposes to extend the capabilities of hardware performance counters to enable tracking and regulation of memory related interference. One important difference of this work compared with our proposal is that, MCCU interrupts the processor when the budget is exhausted, similar to prior software based memory bandwidth regulation solutions [7]. Therefore, it does not eliminate the interrupt handler overhead and cannot regulate the memory accesses at fine-grained time intervals as we do in our proposal. ABU [42] is most similar to us as it is also a hardware based memory bandwidth regulator. The main difference is that ABU is aiming at regulating AXI [43] bus

based hardware accelerators on FPGAs, whereas our design focuses on regulating cores within a microprocessor design.

Understating the problem of the memory contention, major processor designers and chip manufacturers have started adding extensions to their multicore processors to bound the memory interference. ARM recently published a specification [44] on the extensions to the architecture of its server processors to partition and regulate the shared memory resources. Similarly, AMD has published a specification [45] on the extension to monitor and control the usage of shared resources. However, both specifications are targeted at enterprise networking and server systems, and we are not able to find any published literature evaluating performance of these features. To the best of our knowledge, our work is the first hardware bandwidth regulator and implementation to bound the inter-core interference in the context of safety-critical real-time embedded systems.

In the real-time systems community, many OS-level solutions have been proposed to manage the shared resources in COTS multicore processors to improve temporal isolation on such systems. For instance, page-coloring [3]–[5] is used to partition the cache and the DRAM banks. There are also proposal [7], [8] which use hardware performance counters to improve the isolation in the multicore processors. However, because the implementation details of the COTS platforms are not typically disclosed by the manufacturers, the degree of isolation that can be achieved by these solutions is limited. Moreover, many of these software methods incur runtime overhead. In particular, there is non-negligible interrupt handling overhead in hardware performance counter based OS-level memory bandwidth regulation approaches [7], [8]. As the result, it is not possible to regulate the bandwidth at fine time intervals using these solutions.

VII. Conclusion and Future Work

We have presented BRU, a hardware unit that regulates per-core accesses to the shared memory resources. Since BRU is implemented in hardware, it eliminates the runtime overhead associated with prior software based regulation solutions. Moreover, BRU is able to regulate at a much finer time intervals. This enables it to more effectively protect real-time tasks, especially those with short execution times. In addition, BRU improves bandwidth utilization for the best-effort tasks using group bandwidth regulation that enables efficient bandwidth sharing. Compared to most other hardware solutions, BRU is less intrusive as it eliminates the need for redesigning and verifying the existing hardware components. We have synthesized BRU in a 7nm technology node and showed that the overhead of integrating it on chip is limited.

In this work, we have used an open-source hardware architecture to design and evaluate BRU. One important benefit of using open-source architectures is that, we can investigate the details of their implementations to better understand the timing behavior. We plan to further utilize open-source architectures in our future work.

References

[1] Certification Authorities Software Team, "CAST-32: Multi-core processors," Federal Aviation Administration (FAA), Tech. Rep., May 2014.

[2] N. Kim, B. C. Ward, M. Chisholm, J. H. Anderson, and F. D. Smith, "Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning," *Real-Time Systems*, vol. 53, no. 5, pp. 709–759, 2017.

[3] J. Liedtke, H. Hartig, and M. Hohmuth, "Os-controlled cache predictability for real-time systems," in *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. IEEE, 1997, pp. 213–224.

[4] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, 2014.

[5] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *Computational Sci. and Eng. (CSE)*. IEEE, 2013, pp. 685–692.

[6] 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 *Parallel Architecture and Compilation Techniques (PACT)*. ACM, 2012, pp. 367–376.

[7] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "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, 2013, pp. 55–64.

[8] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 109–118.

[9] Intel, *Improving Real-Time Performance by Utilizing Cache Allocation Technology*, April 2015.

[10] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vcat: Dynamic cache management using cat virtualization," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017, pp. 211–222.

[11] P. K. Valsan *et al.*, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *RTAS*. IEEE, 2016, pp. 1–12.

[12] M. Bechtel and H. Yun, "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, 2019, pp. 357–367.

[13] J. Yan and W. Zhang, "Time-predictable l2 cache design for high-performance real-time systems," in *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2010, pp. 357–366.

[14] J. Yan and Z. Wei, "Time-predictable multicore cache architectures," in *2011 3rd International Conference on Computer Research and Development*, vol. 3. IEEE, 2011, pp. 1–5.

[15] B. Lesage, I. Puaut, and A. Seznec, "Preti: Partitioned real-time shared cache for mixed-criticality real-time systems," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*. ACM, 2012, pp. 171–180.

[16] S. Goossens, B. Akesson, and K. Goossens, "Conservative open-page policy for mixed time-criticality memory controllers," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 525–530.

[17] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, "A rank-switching, open-row dram controller for time-predictable systems," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 27–38.

[18] L. Ecco and R. Ernst, "Improved dram timing bounds for real-time dram controllers with read/write bundling," in *2015 IEEE Real-Time Systems Symposium*. IEEE, 2015, pp. 53–64.

[19] P. K. Valsan and H. Yun, "MEDUSA: A Predictable and High-Performance DRAM Controller for Multic ore based Embedded Systems," in *Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE, 2015, pp. 86–93.

[20] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "Compsoc: A template for composable and predictable multi-processor system on chips," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 1, p. 2, 2009.

[21] C. Hernández, J. Abella, F. J. Cazorla, A. Bardizbanyan, J. Andersson, F. Cros, and F. Wartel, "Design and implementation of a time predictable processor: Evaluation with a space case study," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[22] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf *et al.*, "Merasa: Multicore execution of hard real-time applications supporting analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, 2010.

[23] M. Schoeberl, C. Silva, and A. Rocha, "T-crest: A time-predictable multi-core platform for aerospace applications," in *Data systems in aerospace, DASIA 2014*. European Space Agency, 2014.

[24] J. Cardona, C. Hernandez, J. Abella, and F. J. Cazorla, "Maximum-contention control unit (mccu): Resource access count and contention time enforcement," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 710–715.

[25] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep., Jun 2015.

[26] S. Karandikar *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *ISCA*, 2018, pp. 29–42.

[27] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 55–64.

[28] K. Asanović *et al.*, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep., Apr 2016.

[29] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified," 2019.

[30] J. Bachrach *et al.*, "Chisel: Constructing hardware in a Scala embedded language," in *DAC*. IEEE, 2012, pp. 1212–1221.

[31] SiFive. SiFive's Freedom Platform. [Online]. Available: https://github.com/sifive/freedom

[32] SiFive, "SiFive TileLink Specification," 2017.

[33] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *ACM SIGARCH Computer Architecture News*, vol. 12, no. 3. ACM, 1984, pp. 348–354.

[34] H. C. Cook, "Productive design of extensible on-chip memory hierarchies," Ph.D. dissertation, UC Berkeley, 2016.

[35] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[36] D. Biancolin, S. Karandikar, D. Kim, J. Koenig, A. Waterman, J. Bachrach, and K. Asanovic, "Fased: Fpga-accelerated simulation and evaluation of dram," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 330–339.

[37] "Highly agile masks made effortlessly from RTL," https://github.com/ucb-bar/hammer.

[38] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finFET predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[39] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A pret microarchitecture implementation with repeatable timing and competitive performance," in *2012 IEEE 30th international conference on computer design (ICCD)*. IEEE, 2012, pp. 87–93.

[40] F. Farshchi *et al.*, "Deterministic memory abstraction and supporting multicore system architecture," in *ECRTS*, vol. 106, 2018.

[41] 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 *ACM Sigplan Notices*, vol. 45, no. 3. ACM, 2010, pp. 335–346.

[42] M. Pagani, E. Rossi, A. Biondi, M. Marinoni, G. Lipari, and G. Buttazzo, "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, 2019.

[43] Arm, "AMBA AXI and ACE Protocol Specification," 2013.

[44] Arm, "Arm Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A," July 2019.

[45] AMD, "AMD64 Technology Platform Quality of Service Extensions," August 2018.