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, Real-time, Multicore Processor, RISC-V, TileLink**

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 time-predictability in multicore processors is a serious problem especially for safety-critical systems such as avionics that often require evidence of bounded execution time [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 the 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 the 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 runtime. Furthermore, it enables a cycle-granularity fine-grained bandwidth regulation capability compared to the millisecond granularity regulation capabilities in the prior software-based regulation mechanisms. In addition, BRU supports a domain-based regulation scheme where each domain can be composed of one or more cores.

We implement BRU in an open-source out-of-order multicore processor [25] and evaluate its performance with the FireSim simulator [26] running on the Amazon FPGA cloud. We conduct a set of experiments using both synthetic and real-world benchmarks from IsolBench [11] and SD-VBS [27] benchmark suites to evaluate BRU's 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 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 introduces insignificant (up to 2%) timing overhead and negligible (less than 0.3%) chip area overhead.

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[1].
- 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 related work in Section VI and conclude in Section VII.

II. Background

We use the Rocket Chip generator [28] to implement 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 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 design 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].

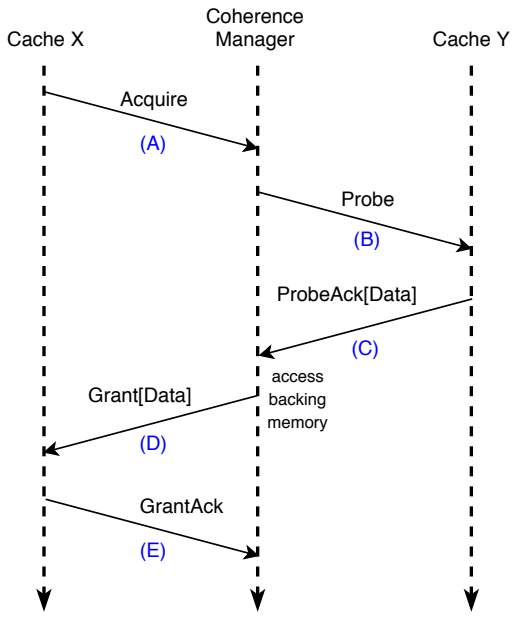


Fig. 1. Cache X sends an Acquire message to the coherence manager then, the manager probes Cache Y. The channel names are shown in parentheses. Adopted from [34].

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

[1]BRU is available as open-source at https://github.com/CSL-KU/bru-firesim

copy of the cached data too, it responds with ProbeAckData message which carries the payload.

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. Note that ReleaseData is transferred over Channel C which is the same channel used for transferring ProbeAck and ProbeAckData. We will see how this affects our design when throttling writebacks in Section IV.

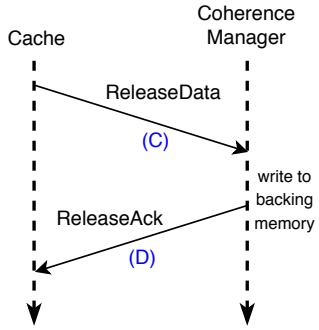


Fig. 2. A cache voluntary releases write permission on a cache line. The channel names are shown in parentheses. Adopted from [34].

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 a read access message which is also used by the instruction cache in the Rocket Chip to read the instructions.

III. BRU Architecture

We start with defining the architecture of our proposed design. BRU is a drop-in hardware module 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 the 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 sent to the shared memory system. BRU is placed where the private caches are connected to the shared memory and regulates memory traffic that goes to it.

Since 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

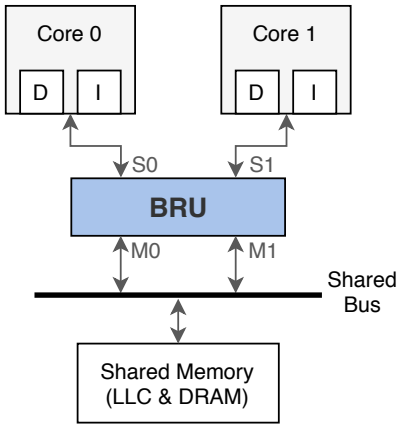


Fig. 3. A simplified view of a multicore processor with shared memory resources. BRU regulates per-core bandwidth at source.

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.

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 at 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. The size of each transaction is equal to the size of a cache line as cache data transfers are typically performed at the granularity of a cache line.

A. Access Bandwidth Regulation Interface

Access bandwidth regulation controls the rate at which private cache misses send a request to the shared memory. Figure 4 shows the registers of a BRU instance for a quad-core processor that supports two regulation domains. At the high level, three groups of registers—Period Registers, Processor Control and Assignment Registers (PCAR), and 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. These are indicated by brackets around their names in Figure 4.

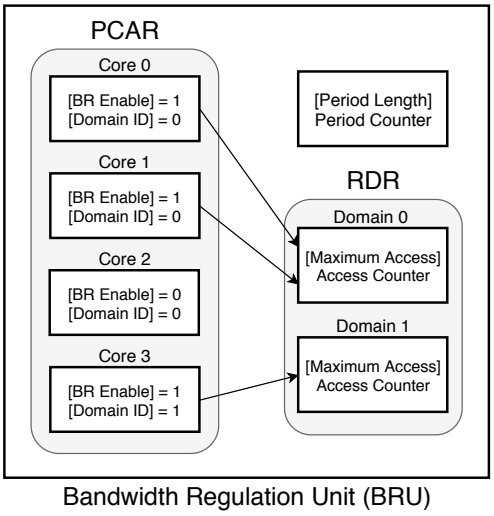


Fig. 4. BRU register interface for access regulation. Cores 0 and 1 are assigned to domain 0, and core 3 is assigned to domain 1. Bandwidth regulation is not enabled for core 2.

Domain control. BRU's bandwidth regulation is performed on a domain. 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 *Bandwidth Regulation Enable Register (BRER)*. DIR determines a core is mapped to which domain, and BRER is used to enable or disable the association. For example, in Figure 4, cores 0, 1, and 2 are assigned 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 taping out the chip as each domain needs some hardware resources.

Budget control. The memory access budget is controlled by 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 memory access budget b. The bandwidth regulation period T is globally applied to all domains and is configured by updating the *Period Length Register (PLR)* in number of clock cycles. When a period begins, *Period Counter (PC)* starts counting from zero and is incremented by one on every clock cycle. Once PC reaches the value programmed in PLR, a period completes, and the domain's access counter (AC) is cleared to replenish the memory bandwidth budget for the next period. Recalling that the size of a memory transaction is equal to the cache line size, the bandwidth budget is calculated using the equation below:

$$B/W\ budget = \frac{MAR}{PLR+1} \cdot LS \cdot f_{clk}, \tag{1}$$

where MAR and PLR are values programmed in their respective registers, LS is the cache line size, and f_{clk} is the system clock frequency. Note that since Period Counter starts counting from zero, the regulation period is equal to $PLR+1$.

B. Writeback Bandwidth Regulation Interface

Our baseline access 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 writeback traffic separately, BRU adds two new registers to each domain: *Writeback Counter (WC)* and *Maximum Writeback Register (MWR)*. Similar to the AC and MAR registers in the access bandwidth regulation, WC is incremented by one on each writeback to the shared memory and MWR determines the writeback budget wb over the regulation period T. Once WC reaches the value programmed in MWR, writebacks are throttled until the beginning of the next period at which writeback budget is replenished. We discuss the implementation of writeback 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 Bandwidth Regulation

Figure 5 shows an example dual-core Rocket Chip SoC with a BRU instance. 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 to. The BRU module sits between the core's private caches and the shared bus so that it can regulate access to the shared memory. Specifically, 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.

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/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[2] 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 that we use to throttle Channel A. In this figure, when $throttle_i$ is set to the logic high, the Channel A corresponding to core i is throttled.

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

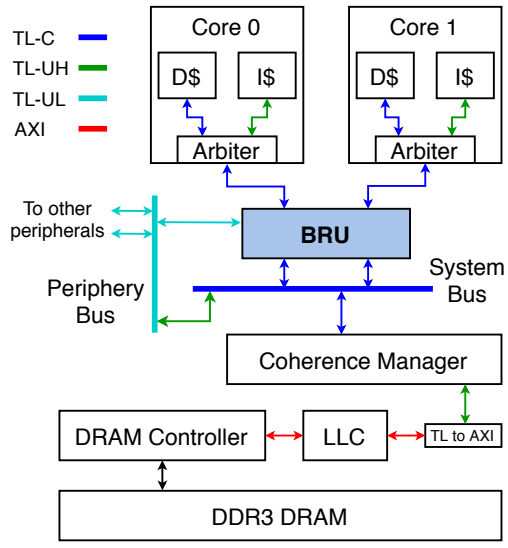


Fig. 5. A dual-core Rocket Chip SoC with BRU.

The other signals of Channel A plus the signals of channels B, C, D, and E pass through BRU without any alternations. We show how the rest of the BRU logic drives $throttle_i$ to implement the desired behavior in the following.

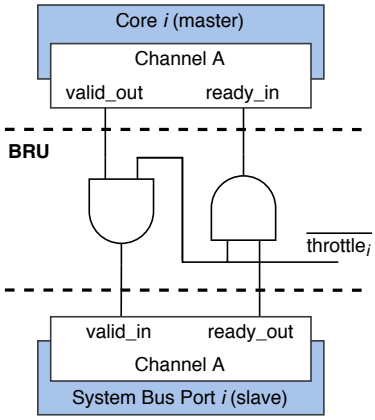


Fig. 6. The logic which controls the flow of messages on Channel A. Boundaries of BRU are denoted with dashed lines.

Algorithm 1 shows the high-level pseudo-code of BRU, which is evaluated at the 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: (1) global period management (line 1-6) and (2) per-domain memory access counters and throttling signal management (line 7-18).

To manage global periodic regulation, the period counter, $PeriodCounter$, is incremented at each clock (line 5) until it reaches to the end the period (line 1), at which point both

Algorithm 1: Access Bandwidth Regulation

```
1  if PeriodCounter ≥ PeriodLength then
2  |   PeriodCounter = 0
3  |   foreach c in AccessCounters do c = 0
4  else
5  |   PeriodCounter++
6  end
7  for i ← 0 to n − 1 do
8  |   throttle(i) = 0
   |   // if enabled for core i
9  |   if BREnables(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
```

the global period and per-domain access counters are reset (line 2-3).

On the other hand, per-domain memory access counters, $AccessCounters$, are incremented whenever data or instruction is requested by the cores which belong to the corresponding domains (line 13-16). To distinguish instruction from memory-mapped I/O accesses, we check the address of the Get messages against the base address of the main memory (line 13). If a domain's access counter reaches to the domain's budget (line 10), then the corresponding core's throttle signal is asserted (line 11).

B. Writeback Bandwidth Regulation

We now describe how BRU regulates the writeback traffic from the core private caches to the shared memory. The writeback regulation, when enabled, allows the user to set a lower budget for writebacks compared to the default access regulation (which only regulates cache misses). When writeback regulation is disabled, writebacks are still limited by the number of cache misses on a writeback cache.

Let us first explain how regulating cache misses (access regulation) limits 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 for both types of writebacks is a cache miss. Therefore, controlling the rate of the cache misses at a certain bandwidth budget, as we do in access regulation, limits the rate of the writeback issuance at the same bandwidth budget.

From the explanation above, we can conclude that to regulate the writebacks at a bandwidth budget lower than the access regulation bandwidth budget, 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 to regulate 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 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 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 $WritebackCounters$ and $MaximumWritebacks$. Moreover, $WritebackCounters$ 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

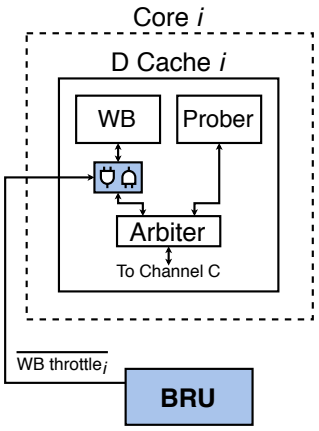


Fig. 7. BRU sends a signal to the data cache to throttle writebacks.

Algorithm 2: Writeback Bandwidth Regulation

```
1  if PeriodCounter ≥ PeriodLength 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 BREnables(i) then
7  |  |  if WritebackCounters(DomainIDs(i)) ≥
   |  |   MaximumWritebackes(DomainIDs(i))
   |  |   then
8  |  |  |  WB_throttle(i) = 1
9  |  |  end
10 |  |  if C(i).isReleaseData ∨ C(i).isProbeAckData
   |  |   then
11 |  |  |  WritebackCounters(DomainIDs(i))++
12 |  |  end
13 |  end
14 end
```

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 actively 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 these cores.

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 as 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 and to run a real-time task for one thousand periods to analyze its response time.

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.

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

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 (Miss Status Holding Registers) 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], [37].

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.[3] 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 from/to the memory. We denote the read and write variants with BwRead and BwWrite, respectively. There is also a periodic variant of Bandwidth benchmark which we denote with BwWrite-RT and BwRead-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.

[3]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

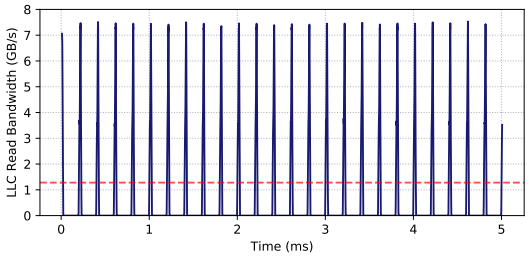
A. Effect of Regulation Period in Regulation Performance

In the first set of experiments, we demonstrate the impact of fine-grained bandwidth regulation over a coarse-grained one. The experiments in this subsection run on one core. Thus, to eliminate the impact of the other cores accessing the memory on our measurements, we run the experiments in this subsection on a single-core processor. The rest of system parameters are as in Table I.

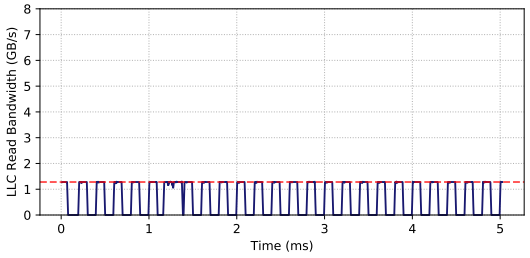
In the first experiment, we use the synthetic BwRead-RT benchmark and configure it to access a 120KB array every 200μs to resemble an application with short burst accesses. The average memory bandwidth of this application is equal to 600MB/s (120KB $\div$ 200μs) and each burst is about 22μs in length. We set the access regulation bandwidth budget at 1280MB/s and run the application once with 1ms and another time with 200ns regulation period. Based on the 2.13GHz clock frequency, these periods are equal to 2.13×10^6 and 426 cycles, respectively. Also, Maximum Access Register (i.e. the access budget) is programmed with $20,000$ and 4, respectively based on the 64-byte cache line size (see Equation 1).

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 periodic bandwidth 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 1280MB/s, it is not throttled for the 1ms regulation period. However the average demand across the length of a burst is much higher (maximum 7.5GB/s). That is why the memory accesses are throttled when the regulation period is set to 200ns.

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.



(a) 1ms regulation period.



(b) 200ns regulation period.

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

Figure 10 and Figure 11 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 200ns 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 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 with 1280MB/s access bandwidth budget. We execute the real-time task for one thousand periods under four different regulation periods and plot the CDF of the task's response times.

Figure 9 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 *100ns*, *1μs*, *1ms*, and *10ms*, 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 observed response time decreases

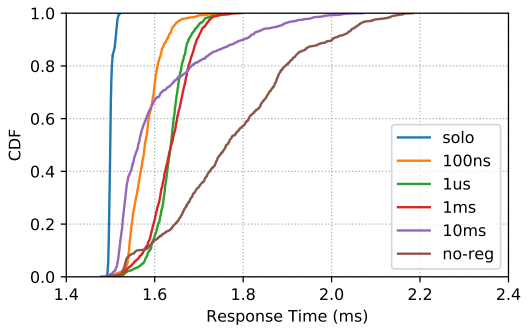


Fig. 9. Response time CDF of BwWrite-RT co-scheduled with disparity best-effort tasks under different regulation period configurations, but with the same access bandwidth budget of 1280MB/s.

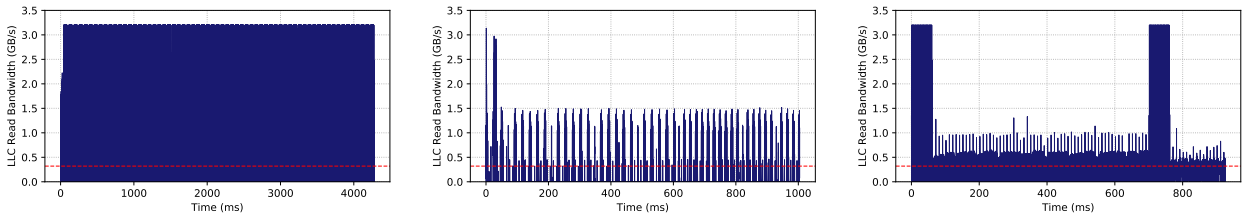
and variations of the real-time task are significantly reduced because BRU's bandwidth regulation limits the best-effort co-runners impact on the real-time task.

Note, however, that at 10ms regulation, which is longer than the real-time task's solo WCET of 1.52ms, we still observe large response time variations despite using BRU. This is because the bandwidth regulation may not always be applied when the real-time task is executed due to the long regulation period. At 1ms, 1μs, and 100ns, the response time variations are significantly reduced because shorter regulation periods make bandwidth regulation to be applied more evenly while executing the real-time task. In general, we find that the smaller the regulation period is, the more effective the BRU is in applying bandwidth regulation.

C. Effect of Group Bandwidth Regulation

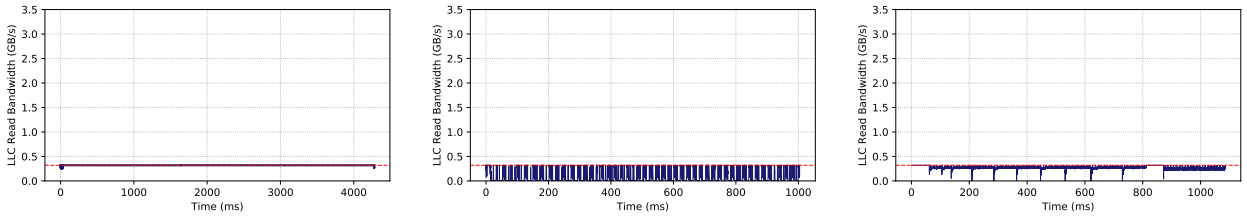
In the following two experiments, we show the effects of using group-based bandwidth regulation to the regulated best-effort tasks and the protected real-time tasks. The basic experiment setup is as follows. In the first experiment we run Latency, which is used as the protected real-time task, on core 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 are used as the best-effort tasks on cores 0~2 and set their WSS to be DRAM-fitting. We regulate the bandwidth of cores 0~2 under two different domain assignment schemes. In the first scheme, *1-domain*, we assign the cores to one domain and regulate their memory accesses collectively. In the second scheme, *3-domain*, we assign each core to a separate domain and split the access bandwidth budget equally among them. For instance, if the total access 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.

Figure 12 shows the execution times of the real-time task (Latency) as a function of the total bandwidth budget for the regulated best-effort tasks (three BwWrite instances). As expected, assigning smaller bandwidth budgets to the best-effort tasks helps protect the real-time task. Furthermore, as



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

Fig. 10. 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. 11. LLC read bandwidth for SD-VBS at 200ns regulation period and 320MB/s budget. Measured at 10μs and applied a 10-point moving average.

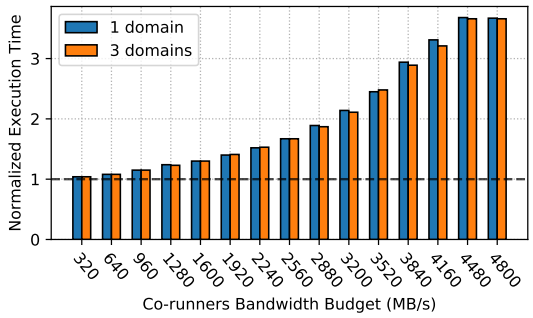


Fig. 12. Normalized execution times of the real-time task (Latency on core 3) as a function of the total access bandwidth budget given to the regulated best-effort co-runners (three BwWrite instances on cores 0~2) under two different domain assignment schemes.

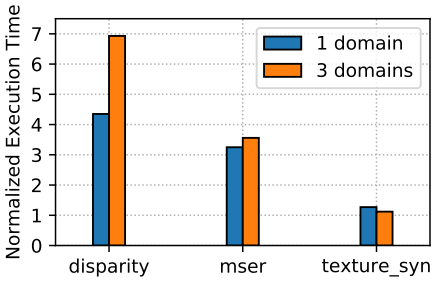


Fig. 13. Normalized execution times of three best-effort SD-VBS benchmarks, regulated under two different domain assignment schemes. The total bandwidth budget is fixed at 1280MB/s in both schemes.

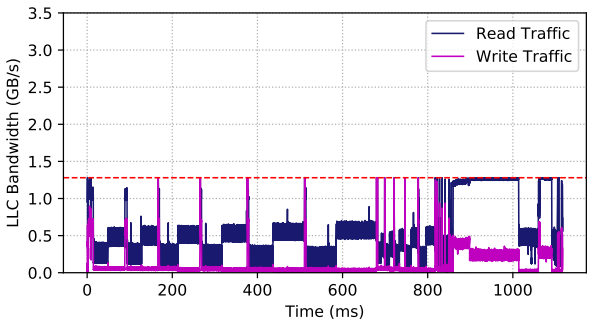
long as the collective bandwidth budget of the best-effort tasks is the same, both 1-domain and 3-domain regulation are similarly effective in protecting the real-time task.

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 synthetic BwWrite benchmark. We choose *disparity*, *mser*, and *texture_synthesis*, which represent high, medium, and low memory intensive workloads, respectively, as per Table II. 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 1280MB/s is assigned to cores 0~2 and these cores compete for the bandwidth with each other. In the 3-domain scheme, on the other hand, each core is assigned with 426.7MB/s (1/3 of 1280MB), which cannot be shared with the other cores.

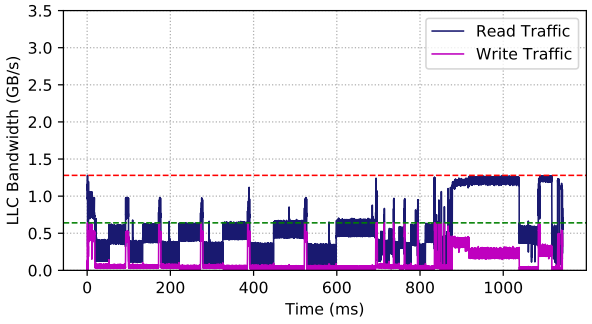
Figure 13 shows the normalized execution times of the three SD-VBS benchmarks. Note first that, under the group regulation scheme (1-domain), the execution of the disparity, which is the most memory intensive benchmark, is markedly smaller, by 37%, than that of the per-core regulation scheme (3-domain). This is because in group regulation, the total bandwidth is more effectively utilized across all cores in the group, while in per-core regulation, any under-utilized bandwidth of an individual core is wasted.

D. Effect of Writeback Regulation

In the experiments above, we only set the budget for access regulation i.e. we only regulated the cache misses. As we described in Section IV-B, this results in regulating the read and write traffic with the same budget. In this subsection, we set up experiments to demonstrate how writeback regulation



(a) Writeback regulation is disabled; access budget: 1280MB/s.



(b) Writeback budget: 640MB/s; access budget: 1280MB/s.

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

can be used to regulate the write traffic with a budget smaller than the access budget.

First, Figure 14(a) shows the result of running *sift* with a 1280MB/s access 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 traffic to the LLC is regulated at the same level as the read traffic.

Next, we run another test in which we set the writeback bandwidth budget to 640MB/s while maintaining the 1280MB/s access budget. Figure 14(b) shows the result. As we can see that using writeback regulation, the write traffic is 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 budget for the read traffic.

E. Hardware Implementation Overhead

To study the implementation cost of our design in hardware, we integrate BRU to a multi-core BOOM processor and run synthesis to estimate the area and timing overhead. We use the Cadence Genus synthesis tool with the Hammer [38] automation scripts targeting the ASAP 7nm technology node [39].

Table III shows the post-synthesis chip area breakdown of dual-, quad-, and octa-core BOOM processors. As we can see, the area overhead of BRU is very low as it is less than 0.3%. Note that this is a conservative number as the area for SRAM needed to implement the caches is not included in the

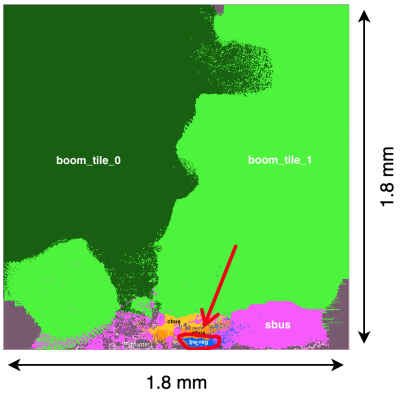


Fig. 15. A dual-core BOOM processor chip layout with BRU circled in red.

measured area. Additionally, we synthesized the processors without BRU to examine the effect of 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 layout of the chip with BRU circled in red.

TABLE III
BOOM PROCESSORS AREA BREAKDOWN (mm^2)

	Dual-core	Quad-core	Octa-core
BRU	0.005 (0.19%)	0.007 (0.17%)	0.023 (0.28%)
BOOM Cores	2.310 (92.41%)	4.072 (95.13%)	8.144 (96.99%)
Others (Buses, Manager, etc.)	0.185 (7.40%)	0.201 (4.70%)	0.230 (2.74%)
Total	2.499	4.280	8.397

VI. RELATED WORK

Deterministic hardware architectures have been extensively studied in the real-time community. PRET [40], T-CREST [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 [41], 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 is a time-consuming and labor-intensive task. Most of these solutions need to redesign the 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 [42], 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 requests 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 [43] is most similar to our work as it is also a hardware-based memory bandwidth regulator. The main difference is that ABU is aiming at regulating AXI [44] bus-based hardware accelerators on FPGAs, whereas our design focuses on regulating cores within a microprocessor design. Note that AXI is not a cache coherent interconnect protocol, thus it is not suitable for on-chip communication between the cores. In contrast, BRU supports coherent on-chip interconnect (TileLink TL-C) and takes into account the complexities that come from communication between the coherence manager and caches (e.g., the coherence traffic).

Understating the problem of the memory contention, major processor designers and chip manufacturers have started adding extensions to their multicore processors to bound the inter-core memory interference. ARM has recently published a specification [45] on the extensions to the architecture of its server processors to partition and regulate the shared memory resources. Similarly, AMD has released a specification [46] on the extension to monitor and control the usage of shared resources. Moreover, Intel incorporated a per-core memory throttling mechanism in their server processors, which they refer to as Memory Bandwidth Allocation (MBA) [47]. However, these are all targeted at enterprise networking and server systems, and we are not able to find any published literature evaluating the performance of these features. To the best of our knowledge, our work is the first hardware bandwidth regulator 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 proposals [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.

For the future work, we plan to provide real-time response analysis for the tasks that are protected using BRU. We also consider extending BRU to regulate the memory traffic of accelerators (e.g., NVDLA [48], [49]) that share the memory system with the processor.

Acknowledgment

This research is supported in part by NSF CNS 1718880, CNS 1815959, and NSA Science of Security initiative contract #H98230-18-D-0009.

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, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *Real-Time and Embedded Technology and Applicat. Symp. (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 *IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2010, pp. 357–366.

[14] J. Yan and Z. Wei, "Time-predictable multicore cache architectures," in *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 *26th Euromicro Conference on Real-Time Systems (ECRTS)*. 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 *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)*. 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, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[24] J. Cardona, C. Hernandez, J. Abella, and F. J. Cazorla, "Maximum-contention control unit (mccu): Resource access count and contention time enforcement," in *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 *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, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Design Automation Conference (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 *International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 330–339.

[37] P. K. Valsan, H. Yun, and F. Farshchi, "Addressing isolation challenges of non-blocking caches for multicore real-time systems," *Real-Time Systems*, vol. 53, no. 5, pp. 673–708, 2017.

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

[39] 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.

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

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

[42] 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.

[43] 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)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

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

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

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

[47] Intel. (2019) Introduction to Memory Bandwidth Allocation. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-allocation

[48] F. Farshchi, Q. Huang, and H. Yun, "Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim," in *2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, 2019.

[49] Nvidia. (2018) NVIDIA Deep Learning Accelerator. [Online]. Available: http://nvdla.org