Dynamic Memory Bandwidth Allocation for Real-Time GPU-Based SoC Platforms

Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni

***Abstract*—Heterogeneous SoC platforms, comprising both general purpose CPUs and accelerators such as a GPU, are becoming increasingly attractive for real-time and mixed-criticality systems to cope with the computational demand of data parallel applications. However, contention for access to shared main memory can lead to significant performance degradation on both CPU and GPU. Existing work has shown that memory bandwidth throttling is effective in protecting real-time applications from memory-intensive, best-effort ones; however, due to the inherent pessimism involved in worst-case execution time estimation, such approaches can unduly restrict the bandwidth available to best-effort applications. In this paper, we propose a novel memory bandwidth allocation scheme where we dynamically monitor the progress of a real-time application and increase the bandwidth share of best-effort ones whenever it is safe to do so. Specifically, we demonstrate our approach by protecting a real-time GPU kernel from best-effort CPU tasks. Based on profiling information, we first build a worst case execution time estimation model for the GPU kernel. Using such model, we then show how to dynamically recompute on-line the maximum memory budget that can be allocated to best-effort tasks without exceeding the kernel's assigned execution budget. We implement our proposed technique on NVIDIA embedded SoC and demonstrate its effectiveness on a variety of GPU and CPU benchmarks.**

I. Introduction

Heterogeneous SoC platforms, comprising both general purpose CPUs and accelerators such as a GPU, are becoming increasingly attractive for real-time and mixed-criticality systems as they can offer the computing performance and efficiency needed to cope with demanding data parallel applications such as those in vision, artificial intelligence, and robotics. However, a major challenge is that on a heterogeneous SoC, CPU, and GPU share the same main memory subsystem, which has limited bandwidth. Therefore, contention for access to the shared main memory can lead to significant performance degradation for applications on both CPU and GPU [1].

Memory regulation, where each processing element is assigned a maximum allowable memory bandwidth, has been proven to be effective in mitigating access interference [2]–[4]. Specifically, in this work, we follow the approach taken by [4], where a real-time GPU program (a *kernel*) is protected from interference generated by Best-Effort (BE) tasks executing on CPU cores. In details, while the kernel is running, the memory bandwidth consumed by each best-effort core is regulated so that it does not exceed a predefined threshold; in turn, this allows the designer to determine a Worst Case Execution Time (WCET) for the kernel with limited inflation compared to the run-alone case. Such worst case execution time can then be used to guarantee schedulability for real-time applications that use the GPU. We point out that static analysis of GPU code is currently infeasible [5], [6], at least for commercial-off-the-shelf architectures, as manufacturers do not release sufficient details to build an accurate hardware model. For this reason, we instead rely on measurement-based WCET estimation, and we argue that real-time GPU computing is necessarily restricted to soft, firm or probabilistic systems.

While memory regulation can significantly reduce the maximum contention, and thus WCET inflation, suffered by a real-time kernel, its impact on the performance of BE applications can be significant. For example, our evaluation in Section VII shows that to limit WCET inflation to 10% for the memory-intensive kernel *histo*, we have to reduce the bandwidth of BE cores to 13% of their maximum throughput. While this severe constraint is needed to provide a WCET bound for the real-time kernel, it is important to notice that the contention caused by BE applications at run-time can sometimes be significantly less than the worst-case measurable one. In particular, computationally intensive BE applications might require low memory throughput and thus cause limited interference; furthermore, the pattern of memory accesses might be different from the worst-case one. Hence, by forcing a constant regulation threshold based on the worst-case interference pattern, we might unnecessarily reduce the performance of BE tasks.

To address this limitation, in this work, we propose to adopt a dynamic approach to memory regulation. Specifically, when a kernel first starts executing, our system enforces the statically-computed bandwidth threshold limit. We then monitor the progress of the kernel at run-time: if we determine that its execution is ahead compared to the worst-case behavior, we can increase the threshold limit by dynamically recomputing the maximum BE bandwidth that allows the kernel to still complete within its original WCET; hence, increasing the performance of BE applications at no cost to real-time guarantees. In details, the key contributions of this paper are as follows:

- We propose a methodology to estimate the progress of a GPU kernel at run-time. Our methodology is based on the observation that a kernel executes a large number of threads with the same code; while such code can

Homa Aghilinasab and Rodolfo Pellizzoni are with the Department of Electrical and Computer Engineering, University of Waterloo, e-mail: (haghilin@uwaterloo.ca, rpellizz@uwaterloo.ca).

W. Ali and H. Yun are with the Department of Electrical Engineering and Computer Science, University of Kansas, e-mail: (wali@ku.edu, heechul.yun@ku.edu).

Manuscript received April 17, 2020; revised June 17, 2020; accepted July 6, 2020. This article was presented in the International Conference on Embedded Software 2020 and appears as part of the ESWEEK-TCAD special issue.

include control instructions, the number of different program paths is usually limited. We thus classify groups of threads into clusters, each with different execution time profiles; then, at run-time, we count the number of completed groups for each cluster as a measure of progress.
- Following the proposed progress mechanism, we introduce a measurement-based WCET approach to estimate the execution time of a kernel based on its remaining number of thread groups per cluster, and the bandwidth threshold for BE cores.
- Using the discussed WCET estimation approach, we then show how to re-compute the BE bandwidth online while ensuring that the kernel completes within its original WCET.
- Finally, we implement and validate our approach on an NVIDIA Jetson TX-2 board, and test it with benchmarks from multiple suites [7]–[9].

II. Background

In this section, we first review basics on GPU architecture, and then we introduce the BWLOCK++ memory regulation scheme [4], which we employ in our work.

A. GPU Execution Model

A GPU is a highly parallel co-processor that performs operations requested by CPU code. A CPU application makes use of the GPU through a parallel-programming framework such as NVIDIA's CUDA, which offers standard APIs. A request to GPU typically comprises the following steps: 1) allocate memory in GPU's memory region and copy data from CPU memory to GPU memory; 2) launch the GPU function—called *kernel*—to process data in GPU memory; 3) wait for kernel completion; 4) copy the processed data from GPU memory region to CPU memory and 5) free the allocated GPU memory.

A GPU kernel consists of a combination of instruction code and a group of threads which execute those instructions. In CUDA terminology, this group of threads is denoted as a *thread block*; the number of thread blocks comprising the kernel and the dimensions of each thread block are specified by the programmer as part of the kernel's launch parameters. At the hardware level, each thread block is processed by a number of hardware threads which form a *warp*. In NVIDIA GPUs, a warp comprises 32 hardware threads—executing in lock-step—and a number of warps can execute simultaneously on a streaming-multiprocessor (SM). A GPU consists of one or more SMs. For example, the integrated GPU in NVIDIA's Jetson TX-2 contains two SMs, each of which comprises 128 GPU cores and can thus execute up to 4 warps simultaneously. Overall, the GPU contains 256 cores and can execute instructions of up to 8 warps at any given time.

Internally, the GPU contains a hardware scheduler which decides which warps to execute out of a pool of active warps. The behavior of the warp scheduler is proprietary and undisclosed; hence, we do not make any specific assumption on how warps are selected for execution. The pool of active warps is formed by selecting threads within a set of active thread blocks; within each thread block, threads are selected in increasing ID order. The number of active blocks depends on the GPU architecture and the resources consumed by each specific kernel; we will use M to denote the number of active blocks for a given kernel in the whole GPU (hence, for a GPU with two SMs, each SM is allocated $M/2$ blocks). When a kernel starts, blocks with IDs 0 to $M-1$ first become active; once all warps within a thread block complete execution, the block finishes and the not-yet started block with lowest ID becomes active. Hence, from the point of view of block scheduling, the GPU can be abstracted as a multiprocessor with M processors using a non-preemptive, global FIFO scheduling policy.

B. BWLOCK++

In an integrated CPU-GPU platform, CPU and GPU share the same memory subsystem, which makes bandwidth sensitive GPU kernels susceptible to interference from CPU applications [3]. BWLOCK++ [4] is a software framework to protect GPU kernels from CPU-side interference on integrated CPU-GPU platforms. In BWLOCK++, one CPU core is dedicated to execute GPU using real-time tasks while the rest of the CPU cores are dedicated to execute best-effort CPU tasks. A GPU-using real-time task declares an acceptable interference budget from co-executing best-effort CPU tasks in the form of total number K of Last-Level Cache (LLC) miss events (and hence, fetches from main memory) from co-executing tasks that the subject task can tolerate in an interval of time T, called a *regulation period*. The budget K is split equally among the regulated CPU cores. A kernel level memory throttling framework [2] then limits the interfering memory traffic from co-executing CPU applications to the specified threshold value through periodic regulation using hardware performance monitoring counters. The implementation in [2], [4] uses a value of T equal to 1-msec. In addition, BWLOCK++ implements a throttling-aware best-effort CPU scheduling algorithm, called Throttle Fair Scheduler (TFS), which favors CPU intensive tasks over memory intensive ones to minimize throttling while real-time GPU tasks are being executed.

III. System Model and Assumptions

We consider an integrated CPU-GPU platform, comprising a GPU and multiple CPU cores, all sharing the same main memory. One core is used to execute real-time tasks, while the remaining cores execute best-effort applications with no real-time constraints. Only real-time tasks can use the GPU, by invoking the execution of a GPU kernel and suspending on the real-time core until the kernel completes. We do not make any assumption on the execution model or scheduling policy for best-effort applications; i.e., a regulation-aware scheduler such as TFS can still be used to improve throughput of best-effort tasks under throttling.

Real-Time Task Model: we assume that the real-time core executes a set of periodic or sporadic real-time tasks. Each task τ_i comprises an alternating sequence of one or more CPU segments and zero or more GPU segments. Each

GPU segment comprises the execution of a kernel $\kappa_{i,j}$, as well as the required memory copy operations. We assume that GPU operations are executed non-preemptively; while kernel preemption can improve the responsiveness of GPU operations [10], [11], it can also incur overhead in terms of additional memory operations. We further assume that only one GPU kernel is executed at a time. While recent work has shown that co-scheduling multiple kernels can improve GPU resource utilization [12], [13], it also complicates the issue of timing analysis. For this reason, we reserve such an extension to future work. The methodology presented in this work does not require any further assumption on how real-time tasks are scheduled; the work in [4] presents a schedulability analysis for the same task model described above, assuming that tasks are scheduled according to fixed-priority preemptive policy on the CPU, and that bounds on the length of each memory copy operations, each kernel execution and the total amount of CPU execution are known.

Regulation Model: each kernel $\kappa_{i,j}$ is protected by enforcing a maximum budget ratio Q for best-effort cores through BWLOCK++. Specifically, let $BW^{\max}$ to denote the maximum cumulative memory throughput that can be generated by the best-effort cores in number of LLC misses per second; we will obtain the value of $BW^{\max}$ experimentally in Section VII. Then, a budget ratio of Q corresponds to a regulation budget of $K = BW^{\max} \cdot T \cdot Q$ LLC misses per period T. Note that $Q = 0$ corresponds to the case where the kernel runs in isolation (without interference from the CPU), while $Q = 1$ corresponds to the case where no regulation is applied (maximum CPU-caused interference). Our WCET estimation method in Section IV can compute a bound on the WCET $G^e_{i,j}(Q)$ of $\kappa_{i,j}$ for any given value of Q. For each kernel $\kappa_{i,j}$, we define a *nominal* budget ratio $\overline{Q}_{i,j}$, such that BWLOCK++ uses $Q = \overline{Q}_{i,j}$ at the start of the kernel. We expect that the system designer selects the nominal budget so that the resulting WCET $G^e_{i,j}(\overline{Q}_{i,j})$ satisfies any required constraints (e.g., the slowdown of the GPU kernel is within an acceptable margin and the system is schedulable). The overarching goal of our approach is to *increase the actual budget ratio Q used at run-time as much as possible, while guaranteeing that the execution time of the kernel does not exceed its nominal WCET $G^e_{i,j}(\overline{Q}_{i,j})$*. This guarantees that schedulability analysis can be based on the nominal WCET computed off-line.

Platform Assumptions: as we introduced in Section II-A and further elaborate in Section IV, our approach relies on analyzing the execution patterns of thread blocks. To extract detailed timing information, we thus assume that the platform provides the following three functionalities: 1) a cycle accurate timer that can be used to count the elapsed time on the GPU since the beginning of a kernel; 2) a mechanism to determine the IDs of all co-running thread blocks on the GPU; 3) a way to synchronize the timers of the GPU and the real-time core.

IV. WCET Estimation

Based on the discussion in Section III, our goal is to estimate an upper bound on the completion time of a kernel based on the budget ratio Q assigned to BE cores. For simplicity of notation, in the rest of this section, we shall drop subscripts and use κ to refer to the kernel under analysis. As mentioned in the introduction, WCET estimation for GPU kernels is especially difficult because key architectural details, such as the way warps are scheduled, GPU caches are managed, etc., are both undisclosed and difficult to reverse-engineer. Inspired by the approach taken in [5], we thus propose to employ a hybrid approach to WCET analysis: specifically, we assume that the WCET of each thread block can be estimated through measurement-based techniques. We then analytically compose the per-block information to obtain a WCET bound for the whole kernel.

Hence, let N_κ denote the number of thread blocks for kernel κ, and $\forall i, 1 \leq i \leq N_\kappa$, let e_i denote the execution time of the i-th block. Without loss of generality, assume that the kernel starts at time 0, let j be the index of the block that finishes last in the kernel, and t_j be its starting time. Then by definition, the execution time of κ is equal to $t_j + e_j$. We next note that since the j-th block is not started until time t_j, it follows that there must always be M other active thread blocks in the interval $[0, t_j)$. Therefore, it must hold: [1]

$$t_j \leq \frac{\left(\sum_{i=1\ldots N_\kappa} e_i\right) - e_j}{M}; \tag{1}$$

and since $t_j + e_j$ is increasing in e_j, we obtain the following bound on the WCET G^e of the kernel:

$$G^e = \frac{\left(\sum_{i=1\ldots N_\kappa} e_i\right) - e_{\max}}{M} + e_{\max}, \tag{2}$$

where $e_{\max} = \max_{i=1}^{N_\kappa} e_i$.

It remains to determine how to compute an upper bound to the execution time of each thread block e_i. Given that a kernel can comprise thousands of blocks, we find that maintaining per-block WCET information is too cumbersome, especially for on-line estimation. Instead, we propose to first classify the thread blocks in each kernel into clusters, where all blocks in the same cluster have similar execution profiles.

A. Block Clustering

All threads within the same kernel execute the same code; but due to the presence of control instructions such as branches and loops, different threads can execute along different code paths. We find that the following two observations typically hold for well-coded kernels: 1) the number of paths is small, as GPU code tends to be more predictable than CPU code. 2) It is highly desirable for threads within the same thread block to follow the same execution path: when threads in the same warp execute along different paths, the resulting thread divergence forces the GPU to execute the warp along all such paths, incurring a significant performance penalty. For these reasons, thread blocks can typically be classified into a small set of clusters.

Our proposed clustering approach works in two steps: 1) first, we measure the execution time of each thread block by

[1] Note that our logic is equivalent to the computation of the interference rectangle in global scheduling analysis, see [14] for example.

instrumenting the code of the kernel (see Section VI-A for details) and executing it many times in isolation. This allows us to construct an Empirical Distribution Function (EDF) of the execution time of each block. While this process takes time and requires storing a large amount of data, we stress that the clustering step is performed off-line. 2) We then cluster thread blocks together based on their EDFs by repeatedly applying the two-sample Kolmogorov-Smirnov (K-S) test [15] at a level $\alpha = 0.05$.

Let $\mathcal{C}$ be the resulting number of clusters, where the i-th cluster comprises N_i thread blocks. We can then modify Equation 2 to obtain an analytical WCET bound based on per-cluster, rather than per-block, execution time values. Specifically, let e_i^0 be the WCET for blocks of cluster i when executed in isolation (that is, with BE budget $Q = 0$), and let e_i^1 be the WCET when executed under maximum interference ($Q = 1$). We then obtain:

$$G^e(0, \{N_i\}) = \frac{\left(\sum_{i=1\ldots\mathcal{C}} N_i \cdot e_i^0\right) - e_{\max}^0}{M} + e_{\max}^0, \quad (3)$$

for the WCET in isolation, and

$$G^e(1, \{N_i\}) = \frac{\left(\sum_{i=1\ldots\mathcal{C}} N_i \cdot e_i^1\right) - e_{\max}^1}{M} + e_{\max}^1 \quad (4)$$

for the case $Q = 1$, where $e_{\max}^0, e_{\max}^1$ have the obvious meaning: $e_{\max}^0 = \max_{i=1}^{\mathcal{C}} e_i^0, e_{\max}^1 = \max_{i=1}^{\mathcal{C}} e_i^1$.

Finally, we point out that the problem of extracting the values of e_i^0 and e_i^1 from the EDF of each cluster is fundamentally orthogonal to our approach. In our evaluation, we simply set them to the maximum observed execution time in the cluster. In Table I, we report the corresponding values for the kernel of the *histo* benchmark [7], for which $\mathcal{C} = 3$; the kernel has been executed one million times with $Q = 0$ and one million times with $Q = 1$ and memory-intensive BE tasks.

An alternative approach, following probabilistic timing analysis [6] practices, would be to fit the EDF to a test distribution, and then obtain e_i^0 (respectively, e_i^1) as a given percentile of the distribution, depending on the desired confidence level. To this end, we decided to again employ the K-S test with level $\alpha = 0.05$ for the fit of the execution time of each cluster to a normal distribution. Results for *histo* are also reported in Table I, where we show the obtained mean and standard deviation of the normal distribution for each cluster and the goodness of fit, expressed as the ratio of the K-S statistic and the critical value of the K-S distribution [2]. We show the goodness of fit for other benchmarks in Section VII-B. Since in our evaluation we picked e_i^0 and e_i^1 based on the maximum measured times, in the table we also report their corresponding percentile levels based on the fitted normal distribution. We acknowledge that higher percentiles would be needed to satisfy strict certification requirements; however, note that this would result in higher WCET estimates, which would *improve* the performance gain of our dynamic allocation.

[2]The K-S statistic is the maximum difference between the EDF and the cumulative distribution function of the fit distribution; note that ratios below 1 indicate that the null hypothesis is not rejected, and hence the distributions are considered to be equal at the specified level.

Cluster #	1	2	3
worst-case measured e_i^0	1.70	2.50	3.30
worst-case measured e_i^1	3.69	6.52	8.83
$Q = 0$, goodness of fit	0.76	0.61	0.70
$Q = 0$, mean	1.61	2.3	3.21
$Q = 0$, std	0.035	0.067	0.039
e_i^0 percentile	99.5%	99.9%	99.0%
$Q = 1$, goodness of fit	0.46	0.51	0.24
$Q = 1$, mean	3.44	6.27	8.53
$Q = 1$, std	0.09	0.099	0.11
e_i^1 percentile	99.7%	99.4%	99.7%

TABLE I: Clustering: hist benchmark. Time values are in us.

B. Memory Interference Estimation

Equations 3 and 4 provide a way to compute the WCET of the kernel under either no interference ($Q = 0$) or full interference ($Q = 1$). It remains to determine how to bound the WCET for Q values between 0 and 1. This is significantly more difficult due to the way regulation works in our system: namely, BWLOCK++ does not mandate *when* BE cores can perform memory accesses during a regulation period, but only *how many* they can perform. Hence, without further assumptions on the interference model, we cannot determine the worst case memory request pattern. For this reason, we will provide a WCET estimation under the following *interference hypothesis*:

Hypothesis 1: The interference suffered by a kernel for any value of Q is maximized when the BE cores issue consecutive requests at the same time and as fast as possible.

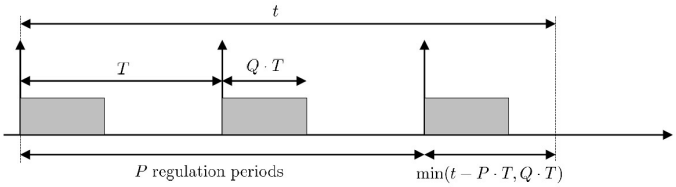
We do not claim that Hypothesis 1 holds generally for all architectures and number of cores. Rather, in Section VII-A we show through extensive testing that the hypothesis holds for our hardware platform; and we remark that systematic testing is accepted as proof of validation for even critical systems by certification authorities. In general, we believe it is likely that the hypothesis holds on other platforms as well, since several previous studies have highlighted that worst-case delays are generated when hardware request queues saturate [8], [16], and maximizing concurrent activity of all cores increases the probability of such occurrence. Finally, in case the hypothesis does not hold, but a precise model of main memory is available, we argue that a more complex analysis, along the lines of [17], [18], could be used to bound the maximum delay suffered by the kernel.

Under Hypothesis 1, in the worst case interference pattern the BE cores perform memory accesses at the maximum rate for $Q \cdot T$ time during each regulation period, and no memory access for the remaining $(1 - Q) \cdot T$ time. We call *memory time* and denote with t^{mem} the total amount of time, over the entire execution of the kernel, when BE cores perform memory accesses. We can then bound the WCET of κ as following: we compute the number of thread blocks for each cluster that execute during the memory time, and assume that they take e_i^1 time each to complete; while blocks which execute outside the memory time take e_i^0 each. Since the number of such blocks can be fractional, we also assume that the blocks that can be partially covered by the end of the memory time in each regulation period (at most M per period) execute for e_i^1. We

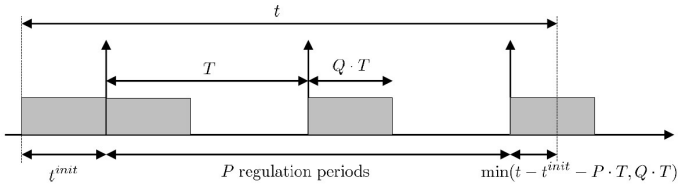
Algorithm 1: WCET Estimation

Input: $Q, \{N_i\}, \{e_i^0\}, \{e_i^1\}, sync$
Output: Kernel WCET

```
t = G^e(0, {N_i})
while 1 do
    Compute t^mem(t, Q, sync) based on Eq. 7
    Compute G^e(Q, {N_i}, sync) by solving Eq. 8-12
      as a Linear Programming (LP) Problem
    if t == G^e(Q, {N_i}, sync) then
        return t
    t = G^e(Q, {N_i}, sync)
end
```



(a) $sync = 1$



(b) $sync = 0$

Fig. 1: t^{mem} derivation

use x_i to denote the (integer) number of fully-covered blocks, and y_i for the partially covered ones.

Algorithm 1 formalizes the corresponding WCET analysis. Note that the memory time depends on the number of regulation periods that the kernel's execution spans; while in turn, the WCET of the kernel depends on the memory time. To solve such circular dependency, Algorithm 1 iterates over the WCET t of the kernel, starting from the WCET in isolation $t = G^e(0, \{N_i\})$ (Equation 3). At each step, the algorithm first uses the value of t to determine the memory time, and then computes a new estimate for the WCET based on t^{mem}. The algorithm then set t to be equal to the new WCET bound and iterates until convergence [3].

We first show how to compute t^{mem} for a time window of length t [4]. The worst case scenario changes whether the kernel starts at the same time as a regulation period ($sync = 1$), or no such assumption can be made ($sync = 0$). In the former case, t^{mem} is maximized when the BE cores access memory as early as possible in all regulation periods, as this maximizes memory accesses within the time window; this case is depicted in Figure 1a. In the latter case, t^{mem} is still maximized when the beginning of the kernel is aligned with the start of BE accesses; however, BE accesses in the first regulation period should happen as late as possible to maximize memory accesses within the time window. Figure 1b shows the described case of $sync = 0$. If we use t^{init} to denote the time between the start of the kernel and the beginning of the first regulation period, and P for the number of regulation periods that are fully contained in the time window, we have:

$$t^{init} = (1 - sync) \cdot Q \cdot T, \tag{5}$$

$$P = \left\lfloor \frac{t - t^{init}}{T} \right\rfloor. \tag{6}$$

We thus obtain the following expression for t^{mem}:

$$t^{mem}(t, Q, sync) = \begin{cases} t & \text{if } t \leq t^{init}; \\ (1 - sync + P) \cdot Q \cdot T + & \\ \min(t - t^{init} - P \cdot T, Q \cdot T) & \text{otherwise,} \end{cases} \tag{7}$$

where note that $t - t^{init} - P \cdot T$ denotes the time within the window in the last regulation period.

Finally, we consider $G^e(Q, \{N_i\}, sync)$. Based on our definition of variables x_i and y_i, we can bound the WCET of the kernel by solving the following linear problem:

$$\max G^e(Q, \{N_i\}, sync) = e^1_{\max} - \frac{e^1_{\max}}{M} + \frac{\left(\sum_{i=1...\mathcal{C}} (x_i + y_i) \cdot e_i^1 + (N_i - x_i - y_i) \cdot e_i^0\right)}{M} \tag{8}$$

$$\frac{\left(\sum_{i=1...\mathcal{C}} x_i \cdot e_i^1\right)}{M} \leq t^{mem}(t, Q, sync) \tag{9}$$

$$\sum_{i=1...\mathcal{C}} y_i \leq M \cdot (\lceil t/P \rceil + 1) \tag{10}$$

$$\forall i = 1...\mathcal{C} : x_i + y_i \leq N_i \tag{11}$$

$$\forall i = 1...\mathcal{C} : x_i \geq 0, y_i \geq 0 \tag{12}$$

Note that Equation 9 constrains the variables x_i based on the length of the memory time, while Equation 8 bounds the WCET in the same way as Equation 2. Variables y_i are constrained to be at most M per regulation period, where the number of regulation periods that are (partially) overlapping with a time window of length t is at most $\lceil t/P \rceil + 1$. To solve the problem in polynomial time, we overapproximate the WCET bound by relaxing the variables to be real rather than integer; the added pessimism is very small as the numbers of thread blocks N_i is generally quite large.

V. Dynamic Budget Allocation

We can now introduce our run-time algorithm to dynamically allocate the memory budget ratio Q to BE cores. As discussed in Section III, let $\overline{Q}$ be the nominal budget for kernel under analysis κ. Since the kernel can be invoked at any point in time, we cannot guarantee that its start time $\bar{t}$ coincides with the beginning of a regulation period. Hence, following the analysis in Section IV-B, we have $sync = 0$ and we let $G^e(\overline{Q}, \{N_i\}, 0)$ be the computed WCET bound for κ. Then, the real-time requirement that our algorithm satisfies is to ensure that κ finishes no later than $\bar{t} + G^e(\overline{Q}, \{N_i\}, 0)$.

[3] A similar iterative strategy is employed by several other works incorporating the effects of resource interference in the execution time of tasks, see [19] for example.

[4] Note that the derivation is equivalent to computing the workload of a sporadic task in a problem window under global scheduling [14].

Specifically, we employ the following approach. The first regulation period, which starts before or at the beginning of the kernel, is assigned the nominal budget $\overline{Q}$. For each successive time t^{reg}, corresponding to the beginning of a regulation period, we perform two steps: 1) first, we determine the number of remaining thread blocks $\{R_i\}$ for each cluster. We instrument the code similarly to Section IV-A to determine which blocks have finished, and use this information to determine R_i; details are provided in Section VI-B. 2) Based on the remaining thread blocks, we determine the maximum budget Q that can be assigned to the next regulation period while ensuring that κ completes by $\bar{t} + G^e(\overline{Q}, \{N_i\}, 0)$.

We next discuss the second step. A trivial solution would be to employ Algorithm 1; since we know that t^{reg} corresponds to the beginning of a regulation period, and the number of not-yet-completed blocks in each cluster is R_i, it follows that for a budget Q, the kernel must complete by $t^{reg} + G^e(Q, \{R_i\}, 1)$. We could thus use binary search to find the maximum Q such that $t^{reg} + G^e(Q, \{R_i\}, 1) \leq \bar{t} + G^e(\overline{Q}, \{N_i\}, 0)$, or equivalently:

$$\max_{Q} : G^e(Q, \{R_i\}, 1) \leq G^e(\overline{Q}, \{N_i\}, 0) - (t^{reg} - \bar{t}), \quad (13)$$

where $t = G^e(\overline{Q}, \{N_i\}, 0) - (t^{reg} - \bar{t})$ is the remaining time to complete execution, and note that $G^e(\overline{Q}, \{N_i\}, 0)$ is a constant computed off-line.

However, we next show in Algorithm 2 that it is possible to directly compute a valid value of Q. At Line 1, the algorithm first checks whether the kernel can complete within the remaining time t even under maximum interference for all blocks, in which case we return $Q = 1$. Otherwise, the algorithm solves the following LP problem to determine the maximum feasible memory time $t^{mem}_{\max}$:

$$\min t^{mem}_{\max} \quad (14)$$

$$t = e^1_{\max} - \frac{e^1_{\max}}{M} + \frac{\left(\sum_{i=1...\mathcal{C}} (x_i + y_i) \cdot e^1_i + (R_i - x_i - y_i) \cdot e^0_i\right)}{M} \quad (15)$$

$$\frac{\left(\sum_{i=1...\mathcal{C}} x_i \cdot e^1_i\right)}{M} \leq t^{mem}_{\max} \quad (16)$$

$$\sum_{i=1...\mathcal{C}} y_i \leq M \cdot (\lceil t/P \rceil + 1) \quad (17)$$

$$\forall i = 1...\mathcal{C} : x_i + y_i \leq R_i \quad (18)$$

$$\forall i = 1...\mathcal{C} : x_i \geq 0, y_i \geq 0 \quad (19)$$

and then derives the corresponding value of Q. Note that the problem at Equations 14-19 is the same as Equations 8-12 with $N_i = R_i$ for each cluster, except that rather than maximizing the WCET for a given memory time, it minimizes the memory time for a given WCET. Intuitively, the problem needs to compute the minimum of $t^{mem}_{\max}$ as this ensures that for any memory time greater than such value, we can actually derive an execution time (right hand side of Equation 15) that is larger than t, and thus unfeasible (note that a higher value of $t^{mem}_{\max}$ would result in higher feasible values of x_i according to Equation 16, and thus of the execution time).

Algorithm 2: On-line Budget Computation

Input: $t, \{R_i\}, \{e^0_i\}, \{e^1_i\}$, with clusters ordered by increasing e^0_i/e^1_i

Output: Budget ratio Q for next regulation period

1 **if** $t \geq e^1_{\max} - e^1_{\max}/M + \left(\sum_{i=1...\mathcal{C}} R_i \cdot e^1_i\right)/M$ **then**
2 **return** 1
3 **else**
4 Compute $t^{mem}_{\max}$ by solving Eq. 14-19 as a Linear Programming (LP) Problem
5 **return** Q s.t. $t^{mem}(t, Q, 1) = t^{mem}_{\max}$ by inverting Equation 7
6 **end**

Lemma 1: The remaining execution time of kernel κ with budget ratio Q, where Q is the result of Algorithm 2, cannot exceed t.

Proof: Since $e^1_{\max} - e^1_{\max}/M + \left(\sum_{i=1...\mathcal{C}} R_i \cdot e^1_i\right)/M$ is an upper bound to the remaining execution time of κ under full interference, if the equation at Line 1 of the algorithm holds, then κ completes within t time no matter the value of Q; hence, we can set $Q = 1$. Therefore, in the rest of the proof assume:

$$t < e^1_{\max} - e^1_{\max}/M + \Big(\sum_{i=1...\mathcal{C}} R_i \cdot e^1_i\Big)/M. \quad (20)$$

Furthermore, by correctness of the WCET estimation in Algorithm 1, it must also hold:

$$t \geq e^1_{\max} - e^1_{\max}/M + \Big(\sum_{i=1...\mathcal{C}} R_i \cdot e^0_i\Big)/M, \quad (21)$$

that is, the remaining time cannot be smaller than the worst case execution time under no interference.

We now show that the LP problem at Equations 14-19 can be solved. Specifically, we show that there is at least one valid assignment to variables x_i, y_i and $t^{mem}_{\max}$ that satisfies all constraints. Note that substituting $x_i + y_i$ with R_i in the right hand side of Equation 15 yields the right hand side of Equation 20, while substituting with 0 yields the right hand side of Equation 21. Hence, we can find a variable assignment with $0 \leq x_i + y_i \leq R_i$ for all i that satisfies Equation 15. By setting $y_i = 0$, the assignment is guranteed to satisfy Equations 17-19. Furthermore, for a sufficiently high value of $t^{mem}_{\max}$, the assignment must also satisfy Equation 16. In summary, all constraints are satisfied, hence such assignment is valid (albeit not optimal). Since a solution exists, let us use $\widehat{t^{mem}_{\max}}$ to denote the optimal value of $t^{mem}_{\max}$ computed by solving the LP problem.

Next, let us use function $f(t^{mem})$ to denote the value of the objective function $G^e(Q, \{N_i\}, sync)$ computed by solving the LP problem at Equations 8-12 for a given value of t^{mem} in Equation 9, with $N_i = R_i$. Since increasing t^{mem} relaxes the problem, f must be non-decreasing; furthermore because the problem is linear, f must be continuous (more specifically, it is piecewise linear). Now since $\widehat{t^{mem}_{\max}}$ is the minimum of the problem at Equations 14-19, it follows that for $t^{mem}_{\max} < \widehat{t^{mem}_{\max}}$, for all valid assignments of variables x_i and y_i according to Equations 16-19, the right hand side of Equation 15 must be strictly less than t. Since the same constraints are used in

Equations 9-12, and the right hand side of Equation 15 is the same as Equation 8, this means that for $t^{mem} < \widehat{t^{mem}_{\max}}$, it must hold $f(t^{mem}) < t$. Similarly, for $t^{mem}_{\max} = \widehat{t^{mem}_{\max}}$, there is a valid assignments of variables x_i and y_i such that the right hand side of Equation 15 is equal to t; this means that for $t^{mem} = \widehat{t^{mem}_{\max}}$, it must hold $f(t^{mem}) \geq t$. From continuity of f, we then obtain $f(\widehat{t^{mem}_{\max}}) = t$: and since function f computes an upper bound to the execution time of the kernel, it follows that for a memory time $\widehat{t^{mem}_{\max}}$, the kernel will complete within t time units.

To complete the proof, it suffices to show that we can compute a valid value of Q in the range $[0, 1]$ based on $\widehat{t^{mem}_{\max}}$. To this end, note that function $t^{mem}(t, Q, 1)$ is non-decreasing and piecewise linear in Q, with $t^{mem}(t, 0, 1) = 0$ and $t^{mem}(t, 1, 1) = t$. Now note that since $x_i \geq 0$, it must hold $\widehat{t^{mem}_{\max}} \geq 0$; and since the left hand side of Equation 16 is strictly smaller than the right hand side of Equation 15, it must hold $\widehat{t^{mem}_{\max}} < t$. Hence, we can find Q by solving $t^{mem}(t, Q, 1) = \widehat{t^{mem}_{\max}}$. ■

A. Improved Allocation

It is interesting to note that the strategy presented in Algorithm 2, which we call the *FAIR* allocation, consists in splitting the memory time fairly among all remaining regulation periods, i.e., as if the same value of Q was assigned to all future periods. However, remember that the algorithm is executed again at the beginning of each period; hence, if the kernel accumulates more slack while executing during the next period, running *FAIR* again will result in a higher value of Q. In practice, as we show in Section VII-C, this means that under *FAIR*, the budget assignment increases over time. A more aggressive allocation could instead compute the worst-case finish time of the kernel by increasing only the budget of the next period, while assuming that all other periods are kept to the nominal budget $\overline{Q}$; such *GREEDY* allocation then results in higher budget values for the first regulation periods of the kernel execution.

Under *GREEDY*, the budget for the next regulation period is computed by first determining the maximum memory time during the next period: this is equal to t^{mem}_{max}, as computed at Line 4 of Algorithm 2, minus the memory time of all other periods under nominal budget, which is $\max\big(0, t^{mem}(\overline{Q}, t - T, 1)\big)$. Since the memory time in a single regulation period is by definition equal to $Q \cdot T$, we then obtain:

$$Q = \min\Big(1, \big(t^{mem}_{max} - \max(0, t^{mem}(\overline{Q}, t-T, 1))\big)/T\Big), \quad (22)$$

which replaces Line 5 in Algorithm 2. Finally, as we will show in Section VII-C, a downside of the *GREEDY* allocation is that the budget ratio Q can change widely between successive regulation periods, possibly leading to a rather unfair bandwidth allocation for co-running BE applications. We thus propose a third allocation scheme, which we call *SMOOTH*, which modifies the *GREEDY* allocation by applying a simple filter of the form:

$$Q_n = \max\Big(\min\big(Q_G, a \cdot Q_G + (1-a) \cdot Q_{n-1}\big), Q_F\Big), \quad (23)$$

Algorithm 3: Measure Block Execution Time

```
1  if threadId.x == 0 then
2      clk = getclock( )
3      if blockId.x > M then
4          read co-running thread block IDs in IDlist
5          find i such that TimeAr[i].ID is not in IDlist
6          Duration = clk - TimerAr[i].clk
7          Write Duration to main memory array
8      else
9          i = blockId.x
10 TimeAr[i].ID = blockId.x
11 TimeAr[i].clk = clk
```

where Q_{n-1}, Q_n are the budgets selected for the previous and current regulation period, respectively, while Q_G and Q_F are the budgets computed by *GREEDY* and *FAIR* for the current period. Note that the $\max$ term ensures that the budget selected by the filter is never lower than the *FAIR* one. Based on our evaluation, we experimentally set a value $a = 0.3$ for the smoothing parameter.

VI. Implementation

In this section, we first illustrate how to instrument the kernel code to measure the execution time of thread blocks, and then we explain how we implement the dynamic budget computation.

A. Kernel Instrumentation

As we explained in Section IV-A, our goal is to measure the execution time of each thread block. As discussed in Section II, whenever a block finishes, another block is assigned to the GPU for execution immediately; hence, the finish time of the terminated block is the start time of the newly assigned block. The execution time of the terminated block is then computed as the difference between finish and start time.

Algorithm 3 shows the pseudo-code of our implementation. We allocate two data structures. An array of size N_κ in main memory is used by the GPU to store the computed execution time of each thread block; after the kernel finishes executing, we read the array content from the CPU and use it as input to the clustering process. TimeAr is an array of size M allocated in GPU memory; each element of the array is a structure comprising the ID and the start time of an active thread block. Based on Line 1, the instrumentation code is executed by the first thread of each block; since threads in each block are selected in order, this guarantees that the thread execution coincides with the start time of the block. The thread reads the current clock time at Line 2, as well as the list of co-running thread blocks at Line 4; in CUDA, this information is exported by the %ctaid register. The list of co-running blocks is then matched based on IDs to the content of TimeAr to determine which thread block has finished (note this is done only after the first M blocks have started), and the execution time of the finished block is computed (Lines 5-7). Finally, the ID and start time of the new block is saved in TimeAr.

Note that this scheme cannot measure the execution time for the M thread blocks of the kernel that finish last, since no new

block will start after their termination. Hence, we only use data for $N_\kappa - M$ blocks to perform the clustering. After clustering, we then rerun the kernel with a modified instrumentation, where we estimate the execution time of each of the last M thread blocks as the difference between the start time of the last and first thread in the block. While such measurement is much less precise than what obtained by Algorithm 3, we found it sufficient to classify each thread block in one of the previously obtained clusters.

B. On-line Budget Computation

Because of the nuances involved in making necessary CUDA library calls from a Linux kernel module, we do not implement our on-line budget computation algorithm at the OS kernel level. Instead, we implement it in a user-level high priority real-time process which runs concurrently to the GPU kernels on the real-time CPU core. At the kernel level, we use BWLOCK++ Linux kernel module [20] to enforce the computed budget values for regulating the memory bandwidth of best-effort CPU cores. For this purpose, we implement a shared-memory based communication mechanism between the kernel module and the user-space budget-computation process. Concretely, for each regulation period, the user-space process calculates a new budget value which is then written to a predefined shared-memory area. The kernel module reads the budget value from the shared-memory and enforces it in the current regulation interval. Note that the budget-computation process takes some time to perform the required computation and pass the information to BWLOCK++; for this reason, we synchronize it to start computation a fixed amount of time before the start of each regulation period of BWLOCK++. We note that the resulting budget computation is still safe albeit more pessimistic: the extra time might cause us to miss some completed thread blocks, but this would lead to higher values of $\{R_i\}$ and hence a higher WCET estimate and a lower computed Q. However, due to the pessimism, it is theoretically possible to compute a value of Q that is less than the nominal budget $\overline{Q}$, in which case we can still safely set $Q = \overline{Q}$.

Similarly to Section VI-A, we allocate a data structure in main memory that can be accessed by both the GPU and the user-level process. We instrument the kernel code so that it writes the current clock value at the beginning of the first thread block, which we take as the starting time $\bar{t}$ of the kernel itself; note that on our platform, the GPU and CPU share the same clock timer, so there is no need for timer synchronization and the budget-computation process can directly use the value passed by the GPU to compute the elapsed time $t^{reg} - \bar{t}$ for the kernel. Each successive thread block writes to main memory the IDlist of concurrent blocks, which is used by the budget-computation process to determine the remaining blocks $\{R_i\}$.

To efficiently compute $\{R_i\}$, we use the concept of a *cluster interval* $[i, j]$, that is, a sequence of thread blocks where all blocks with IDs in $[i, j]$ belong to the same cluster. The budget-computation process is provided with a table, computed off-line, where each line, corresponding to a cluster interval, stores the initial ID for the interval, as well as the number of remaining blocks in all successive intervals. At run-time, the process then matches the IDlist written by the most recently started thread block with the interval table to determine the number of not-yet-completed thread blocks for each cluster; note this is possible because blocks are activated in ID order, hence, if block ID i is executing, then we know that all blocks with ID less than i must either be executing (hence in IDlist) or have finished.

Algorithm 4: Approximated $t^{mem}_{\max}$ Computation

Input: $t, \{R_i\}, \{e^0_i\}, \{e^1_i\}, e^y_{\max}$, with clusters ordered by increasing $w_i = e^1_i/(e^1_i - e^0_i)$
Output: Maximum feasible memory time $t^{mem}_{\max}$

$\forall z_i, i = 1...\mathcal{C} : z_i = 0$
for $j = 1...\mathcal{C}$ **do**
  $z_j = R_j \cdot (e^1_j - e^0_j)/M$
  if $\sum_{i=1...\mathcal{C}} z_i \geq t - e^1_{\max} + \frac{e^1_{\max}}{M} - \frac{\left(\sum_{i=1...\mathcal{C}} R_i \cdot e^0_i\right) + e^y_{\max}}{M}$ **then**
    $z_j = t - e^1_{\max} + \frac{e^1_{\max}}{M} - \frac{\left(\sum_{i=1...\mathcal{C}} R_i \cdot e^0_i\right) + e^y_{\max}}{M} - \sum_{i=1...\mathcal{C}, i \neq j} z_i$
    break
end
return $\left(\sum_{i=1...\mathcal{C}} z_i \cdot w_i\right)$

Finally, we discuss how to efficiently implement the budget computation in Algorithm 2. The algorithm requires us to solve a LP problem; while the complexity of the problem is polynomial in the number of clusters, we would prefer linear complexity for on-line implementation. Hence, we next discuss how we can simplify the problem by adopting an overapproximation of the WCET. Specifically, instead of solving the LP problem at Equations 14-19, we solve the following modified problem:

$$\min t^{mem}_{\max} \tag{24}$$

$$t = e^1_{\max} - \frac{e^1_{\max}}{M} + \frac{\left(\sum_{i=1...\mathcal{C}} x_i \cdot e^1_i + (R_i - x_i) \cdot e^0_i\right) + e^y_{\max}}{M} \tag{25}$$

$$\frac{\left(\sum_{i=1...\mathcal{C}} x_i \cdot e^1_i\right)}{M} \leq t^{mem}_{\max} \tag{26}$$

$$\forall i = 1...\mathcal{C} : x_i \leq R_i \tag{27}$$

$$\forall i = 1...\mathcal{C} : x_i \geq 0 \tag{28}$$

where:

$$e^y_{\max} = M \cdot (\lceil t/P \rceil + 1) \cdot \max_{i=1...C} (e^1_i - e^0_i). \tag{29}$$

Note that since the sum of the y_i variables is constrained by $M \cdot (\lceil t/P \rceil + 1)$ in Equation 17, the computed $e^y_{\max}$ upper bounds the contribution of the variables (that is, $(\sum_{i=1...\mathcal{C}} y_i \cdot e^1_i - y_i \cdot e^0_i)$) in Equation 15. Therefore, to satisfy Equation 25, the LP problem at Equations 24-28 will yield a lower value of the x_i variables, and thus of $t^{mem}_{\max}$ and Q, compared to Equations 14-19. Hence, the obtained Q is safe.

The key benefit of such approximation is that we can solve the new LP problem in a greedy manner. Define $z_i = x_i \cdot (e^1_i -$

$e_i^0)/M$ and $w_i = e_i^1/(e_i^1 - e_i^0)$; then the problem is rewritten as:

$$\min t_{\max}^{mem} \tag{30}$$

$$\sum_{i=1...\mathcal{C}} z_i = t - e_{\max}^1 + \frac{e_{\max}^1}{M} - \frac{\left(\sum_{i=1...\mathcal{C}} R_i \cdot e_i^0\right) + e_{\max}^y}{M} \tag{31}$$

$$\left(\sum_{i=1...\mathcal{C}} z_i \cdot w_i\right) \leq t_{\max}^{mem} \tag{32}$$

$$\forall i = 1...\mathcal{C} : 0 \leq z_i \leq R_i \cdot (e_i^1 - e_i^0)/M \tag{33}$$

Note that here, we are minimizing the weighted sum of the z_i variables, under the constraint that the sum of the variables is equal to a constant. Hence, the optimal solution should increase the value of z_i for clusters with smaller weights before clusters with higher weights, resulting in Algorithm 4. The algorithm first sets all z_i to zero. Then, it iterates over all z_j in order of the lowest weight w_j to the highest, and at each step it attempts to set z_j to its highest possible value (based on Equation 33) of $R_j \cdot (e_j^1 - e_j^0)/M$. If the resulting sum of z_i exceeds the right hand side of Equation 31, then z_j is recomputed so that Equation 31 holds, and the iteration terminates. Finally, $t_{\max}^{mem}$ is computed based on Equation 32.

VII. Evaluation

We use NVIDIA's Jetson TX-2 as our evaluation platform. The Jetson TX-2 board contains a heterogeneous multicore CPU cluster (4 Cortex A-57 + 2 Denver cores) and an integrated GPU. On the software side, we use NVIDIA's default Linux kernel (v4.4.38) and patch it with the changes required for memory bandwidth throttling of best-effort tasks through BWLOCK++ kernel module. As per our system model in Section III, we designate 3 Cortex-A57 cores as BE cores and one as the real-time CPU core; the real-time CPU core is not regulated whereas we use BWLOCK++ to regulate the LLC miss events (`L2_DCACHE_REFILL` in Cortex-A57 TRM [21]) of best-effort CPU cores. Please note that we only use the Cortex cores and disable the Denver cores in all our experiments because the latter lack support of necessary performance monitoring counters required by the memory throttling framework of BWLOCK++. Similarly to [4], we use the Parboil suite [7] as GPU benchmarks. Table II details benchmarks' characteristics which will be discussed throughout this section. Note that each benchmark invokes the same kernel multiple times, possibly with a different input set size; hence, the block clusters remain the same, but the number of thread blocks per cluster and the number of cluster intervals change based on the data size. Due to space limitations and since we expect different invocation of the same kernel to behave similarly, we report results for the first kernel invocation in each benchmark.

We employ bandwidth benchmark from IsolBench suite [8] as our synthetic memory-intensive CPU application. The bandwidth benchmark linearly accesses a 1-D array of configurable size and the sequential write pattern of this benchmark is known to cause worst-case interference on several multicore platforms [22]. In our case, we configure bandwidth benchmark to generate LLC misses; thus creating memory level interference. We also use this benchmark to determine the $BW^{\max}$ value for our TX-2 platform. For this purpose, we run three instances of bandwidth benchmark on the 3 Cortex-A57 cores. Our measurement shows that the maximum cumulative memory bandwidth of the three bandwidth benchmarks is ∼3.9 GB/s. In terms of LLC misses, this corresponds to ∼60,000 events per regulation interval of 1-msec. Divided over 3 best-effort CPU cores, this is equal to 20,000 LLC misses in each regulation interval which corresponds to $Q = 1$ i.e., maximum possible interference from each best-effort core.

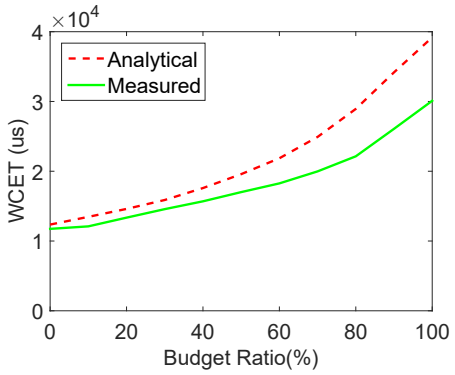


Fig. 2: Analytical WCET vs Measured WCET for *histo*

A. Testing the Interference Hypothesis

We validated Hypothesis 1 through extensive testing. Specifically, we synchronized the execution of three copies of the synthetic bandwidth benchmark running on the three BE cores, and modified the benchmark code to randomly vary: 1) the offsets, relative to the beginning of the regulation period, at which each BE core starts execution; 2) the ratio of read and write operations; 3) the time separation between LLC misses, controlled by inserting a variable number of NOP instructions between reads or writes. We then executed each kernel for several hours and recorded its worst case execution time. In all cases, we found that the WCET is maximized for a 100% write ratio with no NOP added and equal offsets for all cores, which matches the hypothesis.

B. Clustering And WCET Estimation

Table II shows the number of thread blocks per benchmark, as well as the clustering results, in terms of number of clusters, intervals (see Section VI-B), and the worst goodness of fit (i.e., the maximum ratio) over all clusters of each benchmark. The numbers of both clusters and intervals is small for all benchmarks, leading to small memory space overhead for the data structures in Section VI, and low run-time for the on-line algorithm. Per-cluster results are available in Section IV-A for *histo*, and in online appendix [23] for the other benchmarks. All collected measurement data and instrumentation code are available at [24].

To estimate the tightness of the hybrid WCET bounds, we run the following experiment: we first run each kernel one million time for varying values of Q, without any instrumentation and together with the synthetic bandwidth

Benchmark	memory-bound						compute-bound				
	histo	sad	bfs	spmv	stencil	lbm	cutcp	mri_q	mri_gridding	tpacf	sgemm
Number of thread blocks	37,627	59,136	82,318	31,624	31,952	63,627	7,446	19,975	21,587	28,191	15,775
Number of clusters	3	6	6	3	4	5	3	3	4	5	4
Goodness of fit	0.76	0.45	0.53	0.68	0.86	0.66	0.51	0.62	0.47	0.79	0.69
Cluster intervals	14	21	29	17	19	25	10	12	15	22	20
WCET overestim. at $Q = 0$ (%)	5.26	10.04	0.58	6.04	5.49	2.05	4.51	4.23	1.68	2.23	3.31
WCET overestim. at $Q = 1$ (%)	30.20	36.75	12.77	10.09	7.52	34.56	7.22	8.54	4.93	5.60	7.59
Nominal budget (%)	13	11	15.5	26	25.5	21	100	100	100	100	100
Adjusted nominal budget (%)	12.22	10.22	14.78	25.22	24.78	20.22	100	100	100	100	100

TABLE II: Benchmark Characterization

Benchmark	histo	sad	bfs	spmv	stencil	lbm
FAIR	61%	88%	46%	129%	115%	95%
GREEDY	57%	76%	44%	127%	113%	93%
SMOOTH	62%	92%	47%	131%	116%	97%

TABLE III: Improvement over *NOMINAL* in Number of Memory Requests Issued by Synthetic BE Tasks

Benchmark	histo	sad	bfs	spmv	stencil	lbm
Improvement over *NOMINAL* in Number of Memory Requests by BE Tasks						
FAIR, 462.libquantum	106%	193%	78%	167%	143%	121%
GREEDY, 462.libquantum	104%	187%	74%	166%	142%	119%
SMOOTH, 462.libquantum	108%	195%	79%	169%	145%	123%
FAIR, 403.gcc	35%	41%	23%	35%	26%	29%
GREEDY, 403.gcc	34%	38%	21%	38%	22%	25%
SMOOTH, 403.gcc	37%	42%	24%	40%	27%	29%
FAIR, 458.sjeng	13%	12%	7%	0%	0%	3%
GREEDY, 458.sjeng	11%	11%	6%	0%	0%	4%
SMOOTH, 458.sjeng	13%	13%	8%	0%	0%	6%
FAIR, Q mean (%)	39.37	47.65	34.30	72.31	67.86	58.60
GREEDY, Q mean (%)	38.94	47.54	33.96	72.30	67.76	58.18
SMOOTH, Q mean (%)	39.58	47.87	34.79	72.45	68.63	59.34
FAIR, Q std (%)	12.89	20.00	9.67	21.24	20.67	14.78
GREEDY, Q std (%)	20.77	14.04	9.69	14.01	10.78	11.26
SMOOTH, Q std (%)	8.71	8.72	3.62	13.65	10.01	7.10

TABLE IV: Performance Results, SPEC BE Tasks

benchmark, and determine its worst-case measured execution time. To reduce variability, we further force the kernel to start synchronously with the regulation period, i.e. $sync = 1$. We then compare such measured WCET with the analytical WCET obtained through Algorithm 1, and report in Table II the overestimation ratio at both $Q = 0$ and $Q = 1$ for all benchmarks; while Figure 2 shows the detailed WCET plots for the *histo* benchmark as a function of Q. We point out that the overestimation is due to three factors: 1) the measured execution does not represent the real worst-case, as the bandwidth benchmark performs all memory requests at the beginning of each regulation period; hence, it might fail to align memory requests with the most interference-sensitive thread blocks. 2) Our clustering approach leads to some overapproximation of the real WCET of each thread block. 3) Finally, the instrumentation adds some timing overhead; however, this is small, at most 1.5% for the tested benchmarks (measured by running each benchmarks in isolation with and without instrumentation

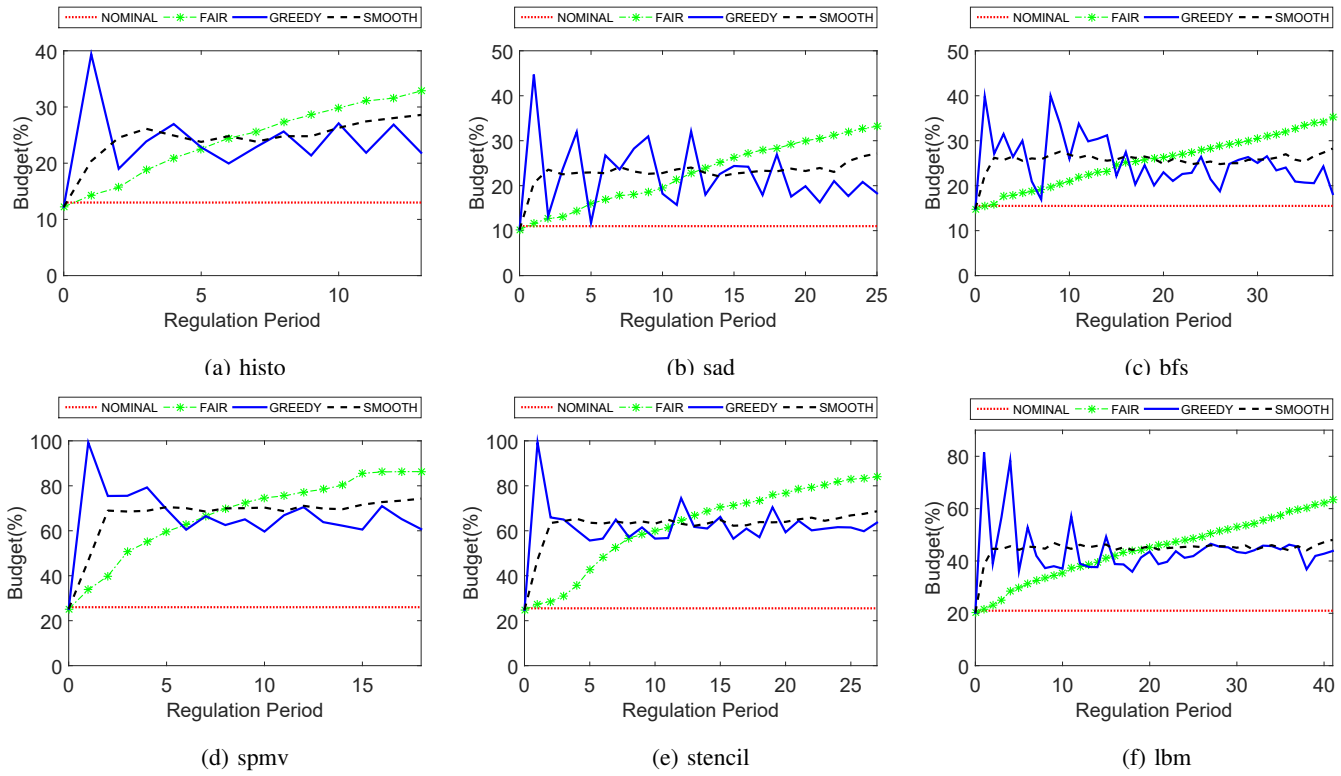
C. Dynamic Budget Allocation

Finally, we evaluate the three dynamic budget allocation schemes in Section V against the *NOMINAL* allocation, where the same nominal budget is used for all regulation periods of a given kernel. To determine the nominal budget $\overline{Q}$ for each kernel κ in a uniform manner, we decided to use the following procedure: we pick the maximum $\overline{Q}$ such that the slowdown $\big(G^e(\overline{Q}, \{N_i\}, 0) - G^e(0, \{N_i\})\big)/G^e(0, \{N_i\})$ for κ is equal to 10%. The obtained values are listed in Table II. We note that the benchmarks in Parboil can be divided in two categories: the first 6 benchmarks are *memory-bound*, and show nominal budgets below 30%. The other 5 benchmarks are *compute-bound*, and are assigned the maximal nominal budget of 100% (in fact, the slowdown of all such benchmarks is below 5% for $\overline{Q} = 1$). Therefore, we conclude that dynamic allocation is not useful for compute-bound applications, and proceed to provide results for the memory-bound benchmarks only.

Under *NOMINAL*, the kernel is run without instrumentation and no extra CPU process. For *FAIR, GREEDY* and *SMOOTH*, as discussed in Section VI we need to instrument each kernel, and run a user-level CPU process to perform the on-line budget computation. We experimentally determined that the process can cause at most 470 LLC misses during each regulation period; since these misses cause extra interference to the GPU, we have to adjust the budget assigned to BE cores for the dynamic schemes by subtracting such LLC amount for every regulation period. This leads to a lower adjusted nominal budget for the first regulation period of each kernel, see Table II. As for the execution time of the budget-computation process, we measured a worst-case time of 10us to compute $\{R_i\}$, and 10us to execute Algorithm 2. Also accounting for the time to communicate with the BWLOCK++ kernel module, we configured the process to start 50us before the beginning of each regulation period.

Figure 3 provides graphs for all benchmarks, where the BE cores execute the synthetic bandwidth benchmark. Specifically, we report results in terms of assigned budget ratio Q for each allocation scheme and regulation period over a single run of each kernel. Note that the graphs for the 4 allocation schemes always finish at the same time, meaning that each kernel executes for a constant number of regulation periods. For the same scenario, Table III shows the performance improvement for each dynamic scheme, in terms of memory requests issued by the BE cores during the execution of the kernel, averaged over a million runs and normalized based on *NOMINAL*. As we can see, the performance improvement of the three schemes is similar, but they behave very differently in terms of budget allocation over time, with *SMOOTH* achieving by far the most uniform distribution. We also note that the performance improvement is significant, ranging from 44% to 131% based on Table III, which is surprising since we use memory-intensive BE benchmarks; this is both due to WCET overestimation, and because the benchmarks fail to cause the actual worst case, especially with $sync = 0$ in this scenario.

Finally, we repeat the same experiments while running a benchmark from the SPEC2006 suite [9] on each BE



(a) histo (b) sad (c) bfs

(d) spmv (e) stencil (f) lbm

Fig. 3: Budget Distribution over Time

core. We selected three benchmarks with different memory intensiveness [2]: 462.libquantum (high), 403.gcc (medium), and 458.sjeng (low). Table IV represents the results in terms of performance improvement over *NOMINAL* for each SPEC benchmark, as well as mean and standard deviation of the assigned budget ratio Q over all regulation periods. We find that for 462.libquantum, the performance improvement is higher compared to the synthetic benchmarks; this is because the other two benchmarks stress the memory less, thus resulting in less interference and more slack for the GPU kernel. The three dynamic allocation schemes again perform similarly, with the exception of the standard deviation, which is significantly smaller for *SMOOTH*.

VIII. Related Work

Due to increased interest in GPU for accelerating parallel real-time applications, many real-time scheduling frameworks for GPU have been proposed in recent years. Since our focus is on memory management, we restrict our attention to work on mitigating resource interference. In [13], the authors show how to partition GPU memory resources, including cache and main memory, to enforce strong isolation between concurrent kernels. However, the approach is highly platform-specific, requiring a great deal of reverse engineering, it is focused on discreet GPUs rather than integrated CPU-GPU SoCs, and does not protect the GPU from CPU interference. SiGAMMA [1] introduced a method to protect real-time CPU tasks from GPU tasks. Their method preempts the GPU kernel by employing a spinning GPU kernel with high-priority to protect critical real-time CPU applications. A compiler-based technique to make GPU code PREM-compliant is introduced in [25]. Under PREM [26], each task has distinct computation and memory phases. The approach in [3], [25] ensures that the CPU does not perform memory accesses during the GPU memory phases, therefore eliminating memory contention by construction. However, it needs significant code restructuring, and can suffer significant overhead from the required fine-grained CPU-GPU synchronization. The most closely related work, as discussed in Section II, is BWLOCK++ [4], which we employ to throttle best-effort cores.

Due to its complexity, WCET analysis for GPU kernels has received less attention compared to CPU analysis. A static analysis approach is introduced in [27], but it assumes a specific behavior of the warp scheduler that is not respected by commercial systems. The approach in [28] also employs static analysis, but with more relaxed assumptions. However, it can not handle cache stalls, and thus cannot be used in our context. A robust measurement-based probabilistic timing analysis is introduced in [6]. Similar to our approach, WCET estimation is based on collecting a trace of independent measurements. However, the approach in [6] is applied at the level of the whole kernel, and thus cannot be used to estimate run-time progress. Finally, a hybrid analysis approach is introduced in [5]. Here, the authors collect measurement traces at the level of individual warps, and then analytically compose the traces to derive the WCET of the whole kernel. Our approach also uses a hybrid analysis, but we apply it at the coarser level of thread blocks, since we find that analyzing traces at the warp level induces too much overhead for run-time implementation.

Run-time slack reclamation for GPU kernels has previously been investigated in Merlot [29]. However, there are three fundamental differences compared to our approach. First, run-

time progress estimation is based on dividing the kernel into a sequence of intervals, whose granularity is too large for our memory regulation framework. Second, it requires hardware modifications to the GPU, while our solution is compatible with commercial hardware, albeit it relies on code instrumentation. Third, the accumulated slack is used to save energy by slowing down the GPU, rather than improving the performance of BE cores.

IX. Conclusion and Future Work

Bounding interference effects is essential for the certification of multicore real-time systems. Bandwidth throttling is an effective mechanism to protect real-time application from main memory interference, but it can lead to significant performance penalties for best-effort applications. To mitigate such performance impact, in this paper we have proposed a dynamic throttling scheme which adjusts the bandwidth budget assigned to best effort cores by exploiting the slack accumulated by a real-time GPU kernel. In particular, we have shown how to use the number of completed thread blocks to estimate the progress of the kernel. Our scheme significantly increases the throughput of memory-intensive, best-effort applications, up to 195% in our evaluation.

As future work, we would like to extend our approach to cover not only GPU segments, but also memory copies and CPU segments. In particular, we believe that hybrid CPU analysis [30], [31] can be adapted to estimate CPU progress in a way compatible with our framework. We also plan to relax the presented system model to allow more general configurations, in particular, allowing other real-time tasks to run concurrently to the GPU kernel on one or more cores.

Acknowledgements

This research has been supported in part by CMC Microsystems, the NSERC, and NSF CNS 1815959.

References

[1] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "Sigamma: Server based integrated gpu arbitration mechanism for memory accesses," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, 2017, pp. 48–57.

[2] 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.

[3] B. Forsberg, A. Marongiu, and L. Benini, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 318–321.

[4] W. Ali and H. Yun, "Protecting real-time gpu kernels on integrated cpu-gpu soc platforms," in *30th EUROMICRO Conference on Real-Time Systems (ECRTS'18)*, 2018, pp. 3:1–3:2.

[5] A. Betts and A. Donaldson, "Estimating the wcet of gpu-accelerated applications using hybrid analysis," in *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 193–202.

[6] K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar, "Measurement-based probabilistic timing analysis for graphics processor units," in *International Conference on Architecture of Computing Systems*. Springer, 2016, pp. 223–236.

[7] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep., 2012.

[8] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–12.

[9] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[10] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 483–496, 2017.

[11] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for gpu with preemption support," in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 119–130.

[12] H. Zhou, S. Bateni, and C. Liu, "Sˆ 3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 190–201.

[13] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional gpus: Software-based compute and memory bandwidth reservation for gpus," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 29–41.

[14] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions on parallel and distributed systems*, vol. 20, no. 4, pp. 553–566, 2009.

[15] Y. Dodge, *The concise encyclopedia of statistics*. Springer Science & Business Media, 2008.

[16] M. G. 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. 1–12.

[17] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, 2018.

[18] H. Yun, R. Pellizzon, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 2015, pp. 184–195.

[19] S. Altmeyer, R. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *RTNS*, 2015, pp. 129–138.

[20] "BWLOCK++ Github Repository," https://github.com/wali-ku/BWLOCK-GPU/tree/master/kernel_module.

[21] ARM Inc., "ARM Cortex-A57 MPCore Processor Technical Reference Manual," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488c/BIIBJJGG.html.

[22] 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.

[23] H. Aghilinasab, W. Ali, H. Yun, and R. Pellizzoni, "Appendix to: Dynamic Memory Bandwidth Allocation For Real-Time GPU-Based SOC Platforms," http://hdl.handle.net/10012/16054.

[24] "GPU Bandwidth Allocation Gitlab Repository," https://git.uwaterloo.ca/haghilin/gpu_bandwidth_allocation.

[25] B. Forsberg, L. Benini, and A. Marongiu, "Heprem: Enabling predictable gpu execution on heterogeneous soc," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 539–544.

[26] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 269–279.

[27] Y. Huangfu and W. Zhang, "Static wcet analysis of gpus with predictable warp scheduling," in *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2017, pp. 101–108.

[28] K. Berezovskyi, K. Bletsas, and B. Andersson, "Makespan computation for gpu threads running on a single streaming multiprocessor," in *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 2012, pp. 277–286.

[29] M. H. Santriaji and H. Hoffmann, "Merlot: Architectural support for energy-efficient real-time processing in gpus," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 214–226.

[30] "RapiTime," https:https://www.rapitasystems.com/products/ra pitime, 2013.

[31] A. Betts, "Hybrid measurement-based wcet analysis using instrumentation point graphs," Ph.D. dissertation, Citeseer, 2008.