E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management

Parul Sohal*, Rohan Tabish†, Ulrich Drepper‡ and Renato Mancuso*
*Boston University, USA, {psohal, rmancuso}@bu.edu
†University of Illinois at Urbana-Champaign, USA, rtabish@illinois.edu
‡Red Hat, drepper@redhat.com

***Abstract*—The proliferation of multi-core, accelerator-enabled embedded systems has introduced new opportunities to consolidate real-time systems of increasing complexity. But the road to build confidence on the temporal behavior of co-running applications has presented formidable challenges. Most prominently, the main memory subsystem represents a performance bottleneck for both CPUs and accelerators. And industry-viable frameworks for full-system main memory management and performance analysis are past due.**

In this paper, we propose our *Envelope-aWare Predictive model*, or E-WarP for short. E-WarP is a methodology and technological framework to: (1) analyze the memory demand of applications following a profile-driven approach; (2) make realistic predictions on the temporal behavior of workload deployed on CPUs and accelerators; and (3) perform saturation-aware system consolidation. This work aims at providing the technological foundations as well as the theoretical grassroots for truly workload-aware analysis of real-time systems. We provide a full implementation of our techniques on a commercial platform (NXP S32V234) and make two key observations. First, we achieve, on average, a 6% over-prediction on the runtime of bandwidth-regulated applications. Second, we experimentally validate that the calculated bounds hold if the main memory subsystem operates below saturation.

I. Introduction

The proliferation of inexpensive and high-performance multi-core embedded platforms have been enthusiastically embraced by the industry. These are seen as an opportunity to migrate away from system designs with many interconnected single-core chips; to consolidate all the application workload onto a few systems-on-chip (SoC) with multiple CPUs and accelerators. And while the transition has been smooth for general purpose workload, the same cannot be stated for safety-critical systems.

It is well known that contention over shared hardware resources leads to substantial violation of temporal properties when workload developed and tested in isolation is consolidated on the same multi-core platform. Effects like shared cache contention [1], [2], DRAM bank conflicts [3], [4], contention at the DDR controller [5] have significantly slowed down the adoption of multi-core solutions in the safety-critical domain. The presence of performance *interference channels* has been acknowledged by certification authorities [6], that have mandated methodologies to "account and bound" the temporal effect of interference channels for the certification of avionic systems.

The last decade has produced seminal results [7] on techniques to manage contention at the different levels of the memory hierarchy. But unfortunately, there is a substantial lack of frameworks and methodologies that can be applied system-wise to: (1) take into account realistic applications, (2) consider that processing workload does not occur only on CPUs; accelerators (e.g. DMAs, video-encoders, GPUs) are also fundamental components in real systems, and (3) that can be deployed on existing platforms while ensuring that the models assumed to derive analytical guarantees are in match with the true behavior of the hardware.

In concurrent activity of CPUs and accelerators, main memory is the performance bottleneck. Thus we set our focus on the problem of contention in the DRAM subsystem. DRAM bandwidth management [8], [9] is a promising grassroots technique to exert control over main memory contention. Many works have studied the behavior of applications in multi-core systems under main memory bandwidth regulation [4], [10], [11]. But the overwhelming focus of these works has been put on formulating an increasingly more refined model of the DRAM subsystem [4], [11] to reduce the pessimism in the timing analysis. On the other hand, the behavior of applications is abstracted away with only a few parameters, for instance, to summarize the worst-case end-to-end number of cache misses [4], [5], [12].

In this paper, we propose a focus-shift. We introduce a comprehensive framework of techniques called *Envelope-aWare Predictive model*, or E-WarP for short. In E-WarP, accurate predictions on the worst-case execution time (WCET) of co-running applications are made following a profile-driven approach. Profiling represents a substantial refinement of measurement-driven approaches, where fine-grained knowledge of the interaction between applications and the platform is collected and leveraged. Conversely, we treat the DRAM subsystem, as much as possible, as a black-box. By shifting our emphasis on a more precise representation of memory bandwidth requirements of applications and by ensuring that the DRAM subsystem operates below its saturation threshold, we demonstrate that highly accurate predictions on the behavior of tasks operating on CPUs and accelerators can be made.

We stress upfront that we do not construct a formal model of the DRAM subsystem, nor formulate provable guarantees. The correctness of our approach is corroborated by a full-system evaluation, which provides evidence that the work presented is practical for industrial applications. Furthermore, our profile-driven approach enables a better understanding of the important aspects that have traditionally received little attention. Precise regulation overheads, impact of burst size on DRAM utilization, and the unexpected presence of memory instructions that bypass regulation are some examples. The proposed E-WarP framework can be used to integrate multi-core, accelerators-enabled real-time systems in all those domains where a measurement-based approach was deemed acceptable for single-core systems.

In summary, this paper makes the following contributions. (1) It introduces the E-WarP model where the time-varying demand for main memory resources is characterized via envelopes. (2) It introduces key requirements and design principles for profile-driven approaches. (3) It considers the integration of broadly implementable techniques for DRAM bandwidth regulation of CPUs and accelerators. (4) It describes how to leverage memory enveloping to perform accurate WCET predictions under regulation for both CPU and accelerator workload. (5) It provides a technique to reason on the saturation level of the DRAM subsystem. (6) Lastly, it proposes a full-system implementation and evaluation that includes a low-overhead profiler and an augmented partitioning hypervisor.

II. Related Work

There has been a plethora of research works [4], [5], [10], [13] that aimed at providing hard real-time guarantees for tasks running on multi-core systems. A common denominator in these works is that they consider the worst-case number of main memory transactions (LLC misses) for tasks in isolation [5], [10]–[12]; then compute an upper-bound on memory interference when multiple applications run in parallel. This type of analysis has been proposed with various degrees of refinement on different DRAM/CPU models. For instance, in [4] the authors assume that there is only one outstanding request per CPU; while [5] focuses on the First-Ready First-Come First-Served (FR-FCFS) DRAM scheduling policy. Compared to this line of work, E-WarP is substantially different because its premise is to rely on high-accuracy observations of the memory demands of applications, while treating the DRAM subsystem mostly as a black-box.

Other works such as [8], [9], [14] focus on implementable mechanisms to regulate/throttle the bandwidth of other low criticality tasks with the goal of reducing contention and improving performance isolation. The first work in this sense was [8], where budget-based bandwidth enforcement is proposed. The work in [9] builds on this technique by allowing high-priority tasks to acquire a "bandwidth lock" on the memory controller. These techniques have also been shown to be implementable at the hypervisor level [14], [15]. Recently, there have been important efforts to control, account, and ultimately integrate the behavior of accelerators into real-time systems. The work in [16] lays the groundwork for managing hardware accelerator defined in FPGA, while [17] touches on the topic of non-CPU components regulated via platform-specific throttling mechanisms. In many ways, E-WarP builds on top of the seminal results achieved in this context and complements the CPU-centric management by integrating traditional accelerators (e.g., DMAs, GPUs) in the picture.

Finally, the need for a DRAM controller capable of enforcing bandwidth partitioning and traffic prioritization has been expressed in multiple papers [18]–[21]. We acknowledge the important design principles proposed in said works. However, as we strive for immediate industrial applicability, we restrict ourselves to commercial-off-the-shelf platforms.

In summary, our work sets itself apart because it proposes a novel profile-driven methodology to characterize the behavior of applications that execute on CPUs and accelerators. It then combines (1) CPU-centric bandwidth regulation techniques with (2) broadly available hardware support for regulation of non-CPU masters. In doing so, key relationships between extracted bandwidth and saturation of the DRAM subsystem are derived. Finally, a full-system integration is proposed where we demonstrate that E-WarP is practical in real systems.

III. System Model and Assumptions

We consider a heterogeneous multi-core system with accelerators and traditional CPUs. A hardware accelerator can be any module capable of initiating transactions to the main memory. DMA engines, GPUs, video encoders/decoders, audio sequencers, network interfaces, are some notable examples. We use m to indicate the number of CPUs present in the system and the index $k \in \{1, \dots, m\}$ to refer to a given CPU_k. The system also features a accelerators indexed using l, with $l \in \{1, \dots, a\}$. The l-th accelerator is indicated with ACC_l. We use *processing element* (PE) whenever what stated applies to both CPU and accelerator.

We make a restriction, namely the *single driver assumption*, on how accelerators are used in our system. We assume that there exists a single CPU task that acts as the driver for a given accelerator. I.e., it must hold that for ACC_l there exists at most one CPU task acting as the driver. The assumption allows us to abstract away the differences in the preemption model of accelerators. The single driver assumption is accurate only in a subset of possible system designs, but it allows us to keep our focus on how accelerators interact with the main memory. For the same reason, we make the assumption that caches [15], [22] and DRAM banks [3], [4] are statically partitioned on a per-core basis to ensure that the load generated by each application toward main memory does not change when multiple applications execute in parallel.

We assume that only one main memory controller is used by all the tasks under analysis. This is referred to as the "DDR controller", or the "DRAM controller". If more than one controller exists, the techniques presented in this work can be extended by partitioning tasks to memory controllers, and then considering each sub-system independently. We assume that the traffic originated by CPU and accelerators towards main memory can be regulated. We use budget-based periodic regulation (MemGuard [8]) to manage traffic from the CPU; we leverage standard ARM QoS support that is broadly available in modern ARM-based SoCs to regulate traffic from accelerators (see Section VI-B). Lastly, the bandwidth at the interconnect should be greater than the bandwidth of both memory controllers.

IV. E-WarP Task Model

The E-WarP task model incorporates the relationship between a task's progress and its demand for main memory. This relationship, expressed via *cumulative memory envelopes*, is captured for each task in isolation. It is leveraged to derive precise predictions on the behavior of the task under regulation. Section VIII is dedicated to constructing memory envelopes following a profile-driven approach.

We consider a set of n sporadic, deadline-constrained real-time tasks scheduled according to fixed-priority. The generic task τ_i is statically assigned to execute on a given CPU_k — partitioned fixed-priority scheduling. A task τ_i is a tuple of the following form: $\tau_i = \{T_i, D_i, C_i, \mathcal{M}_i\}$. T_i represents the minimum inter-arrival time between two jobs of the same task, D_i is the relative deadline of task τ_i, and C_i captures the worst-case execution time (WCET) of τ_i in isolation and without memory bandwidth regulation. The $\mathcal{M}_i$ parameter is a super-set of memory envelopes, one per each PE that the task uses. Each memory envelope $M_j \in \mathcal{M}_i$ is of the form $\{R_j, \sigma_j(1), \dots, \sigma_j(L_i)\}$. Here, L_i is simply the number of $\sigma_j(h)$ elements that compose the envelope, while each

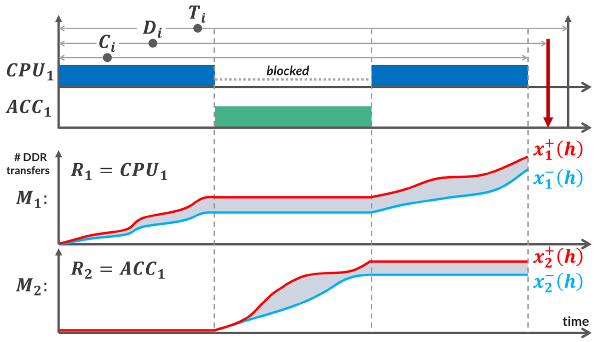

Fig. 1. Overview of main parameters in the E-WarP model for a generic task τ_i executing on CPU_1 and ACC_1.

$\sigma_j(h)$ captures the activity of the task over a fixed small time interval δ. It follows that $L_i = \lceil C_i/\delta \rceil$. The generic $\sigma_j(h)$ has the structure $\{x_j^+(h), x_j^-(h)\}$, where $x_j^+(h)$ (resp., $x_j^-(h)$) captures the upper-bound (resp., lower-bound) on the cumulative number of memory transactions at time $h \cdot \delta$ for the task in isolation and without regulation. Lastly, R_j keeps track of the PE (CPU_k or ACC_l) on which the transactions are executed. Figure 1 provides a visualization of the main parameters in the E-WarP model.

Note that in principle the E-WarP model is capable of encapsulating the behavior of input-dependent applications. That is, as long as profiling is performed over a set of input vectors that exercise those execution paths leading to the worst-case memory envelope. To identify a set of representative input vectors, one could use approaches like symcretic execution [23].

V. Transparent Profiling

In the E-WarP methodology, the profiler plays a key role to (1) define the memory envelopes of applications, (2) study the saturation point of the DDR subsystem, and (3) differentiate between the behavior in main memory of different PEs. To precisely measure these quantities, the profiler must be designed to satisfy the ***transparency requirement***. Under this requirement, the task under observation is not (or minimally) impacted by the activity of the profiler. On the other hand, the higher the granularity of the profiler, the smaller will be the pessimism on WCET estimations. This corresponds to the ***fine-granularity requirement***. Indeed, an extremely coarse profiler degenerates into a model where the entire activity of the task is summarized by a single value for the worst-case number of transactions.

The definition of a good profiler is challenging because the two requirements are opposing objectives. Indeed, to achieve fine granularity, the profiler needs to frequently sample DDR performance and to keep in memory a complete history of the acquired samples. Thus, a fine-grained profiler acts as a *memory bomb*. We briefly outline the principles according to which a profiler that satisfies both requirements can be designed and implemented. We describe our software-only implementation in Section IX.

To satisfy the fine-granularity requirement, the target platform must provide a performance monitoring interface to sample key metrics of the DDR activity. The closer the interface is to where the transactions are served, the higher the accuracy of the resulting profile. Modern embedded platforms include extensive facilities for performance monitoring [24]. Some of these interfaces, such as the ARM Performance Monitoring Unit (PMU), are broadly known and supported in software. Often times, however, there exist better interfaces that operate much closer to main memory. A few notable examples are discussed in the following paragraph.

The P- and T-series of NXP embedded platforms [25], [26] have been extensively studied in the literature [3], [10], [12], [27], [28]. These platforms include an Event Processing Unit (EPU) and a DDR debug subsystem. The DDR debug subsystem can be configured to generate a trace of events at the DDR controller(s) that includes performed reads, writes, DRAM refreshes, DRAM row hit/miss events, and so on. The trace can be processed on chip to create custom event counters.

The DDR trace can also be stored in memory for later retrieval [29], [30]. The Xilinx Zynq UltraScale+ family platforms [31] that are surging in popularity in the recent years [32], [33] include an AXI Performance Monitor (APM) that is interposed between the interconnect and the DDR and that well fits our requirement. The APM can measure the exact number of bytes read/written, as well as their max/min latency over a user-specified sampling interval [31]. Unsurprisingly, support for fine-grained monitoring close to memory resources is not limited to embedded platforms. Intel has recently introduced its family of memory monitoring and management techniques under the name of Resource Director Technology (RDT) [34]. RDT includes support to monitor the memory bandwidth extracted by CPUs via the Memory Bandwidth Monitoring (MBM) interface [35]. On top of the families of platforms mentioned above, yet another example is the NXP S32V234 (NXP S32V family) platform [36] targeted in our implementation.

To ensure transparency, the platform must allow storing the profiled samples without introducing spurious DRAM traffic. Fortunately, modern embedded platforms feature heterogeneous memory subsystems, with two common features that can be leveraged. (1) The presence of fast on-chip scratchpad memories (SPM); and (2) the existence of multiple DDR controllers. Both are valid alternatives. But the limited size of SPMs restricts the length of profiled application, and/or the granularity of the profile. The NXP P- and T-series family of platforms, the Xilinx Zynq UltraScale+ SoCs, the NXP S32V and S32G family of platforms all define both multiple DDR controllers and on-chip SPMs. A key takeaway is that fine-grained transparent profiling is possible today in a range of modern platforms. A sound implementation requires careful consideration of platforms-specific features and the flow of data within the memory hierarchy. Nonetheless, this level of knowledge of the underlying hardware is not uncommon in the development cycle of safety-critical systems.

VI. System-wide Bandwidth Regulation

To achieve system-wide control over memory bandwidth allocation, with the goal of keeping the DRAM subsystem below its saturation threshold, we combine two different mechanisms for CPUs and accelerators, respectively.

A. Regulating CPU Memory Traffic

The first mechanism is budget-based periodic regulation following the MemGuard [8] approach to regulate CPU memory traffic. MemGuard defines a global *regulation period* P and a per-core budget of last-level cache line refills Q_k. The performance measurement unit (PMU) is used to keep track of per-core cache line refills since the beginning of the current P. A local interrupt is delivered by the PMU to stop the core until

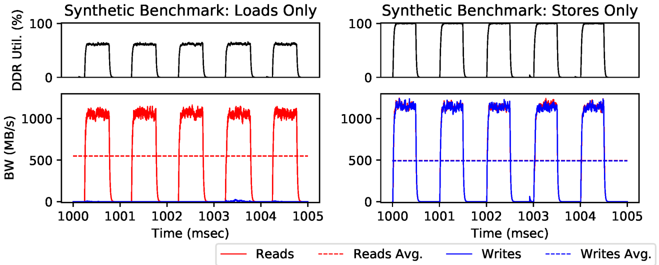

Fig. 2. Profile of a MemGuard-regulated synthetic task performing only loads (left) or only stores (right) under the same regulation budget.

the next P interval if Q_k refills have occurred. All the cores have their budget synchronously replenished at the beginning of the next regulation interval.

Two important considerations need to be made. Consider a single CPU_k active in the system. First, until CPU_k hits Q_k, it alone can potentially drive the DDR controller to 100% utilization. In other words, MemGuard guarantees no regulation at a time scale that is smaller than P. However, when Q_k is selected appropriately if CPU_k performs back-to-back transactions, it will eventually be regulated/stopped until the next regulation period. Thus, the DDR utilization observed over a time period, not shorter than P can be kept below 100%. We statically set P = 1 ms.

Second, the same number of cache refills can result in very different data exchanges with main memory because of write-back CPU caches. When a load or store instruction causes a cache miss, a read transaction is initiated to load the cache block from main memory. If the line being evicted from cache was dirty, then it is written back to memory with a write transaction. Thus, MemGuard only directly regulates read transactions. This phenomenon is shown in Figure 2 which was produced by our profiler. The figure depicts the behavior in terms of read/write bandwidth extracted by a synthetic load- (left) and store-intensive (right) benchmark regulated with the same budget. Note how the same budget can impact the DDR utilization differently, depending on the exact state of the cache. It follows that a safe approach is to construct memory envelopes of read transactions (parameters $x_j^+(h)$ and $x_j^-(h)$), and always assume that the impact on DDR utilization is that of read+write transactions.

B. Regulating the Memory Traffic of Accelerators

To enforce regulation on accelerators, we leverage traffic shaping via ARM QoS support. ARM QoS has been substantially ignored by past literature on real-time systems. Yet, it encompasses a rich set of functionalities implemented in many popular high-performance embedded platforms to manage memory traffic at the level of bus masters. As the first work to leverage and integrate QoS support, we hereby provide some essential background.

Modern ARM-based platforms rely on the Advanced eXtensible Interface 4.0 (AXI4) [37] to establish on-chip data communication channels between processing elements, I/O devices, and memory modules. Each AXI segment defines 5 channels. Read (resp., write) requests are initiated by the master through the **AR** (resp., **AW**) channel. The master provides the data to be written through the **W** channel. The slave responds with read data on the **R** channel, and with write acknowledgments on the **B** channel.

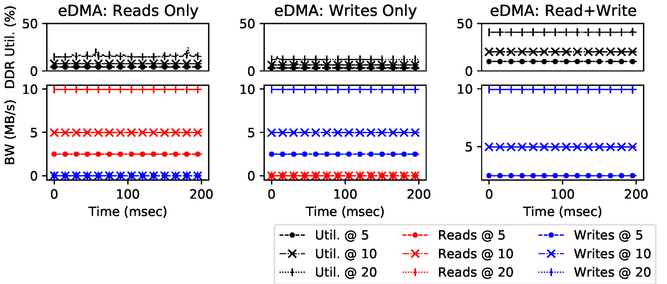

Fig. 3. Profile of QoS-regulated DMA traffic performing only reads (left), only writes (center), or reads+writes (right). Each plot depicts regulation at three enforced rates: 5, 10, and 20.

If the platform includes an interconnect with QoS extensions, such as the ARM QoS-301 [38] or ARM QoS-400 [39], then traffic shaping is also supported and enforced on individual requests that traverse the interconnect. Shaping is supported for individual master ports attached to the main interconnect. In all the instances of QoS-enabled platforms we have studied, all the CPUs are treated as a single master. Hence, QoS support cannot be used to regulate the traffic of individual CPUs, but only to control the aggregated traffic of all the CPUs. Conversely, QoS-based regulation is well suited to regulate traffic from accelerators.

Besides a static bus priority for each regulated master, one can specify a set of parameters to enforce traffic shaping separately for traffic on the **AR** (read requests) and **AW** (write requests) channels. Focusing on the **AR**, the `ar_r` parameter controls the rate of read requests; the `ar_b` the accepted burst size, and `ar_p` the peak rate of read transactions within the allowed burst size. Setting `ar_b` =1 enforces strict regulation at the rate selected by `ar_r`, which is the way QoS is used in this work. The value of `ar_r` can be any 12-bit value greater than 0 [38]. The resulting inter-transaction gap can be computed as: $t_{ar} = (2^{12}/\texttt{ar\_r})/f_{clk}$, where f_{clk} is the reference clock. In our platform, f_{clk} corresponds to the DDR clock (0.5 GHz). The same applies to write requests. In this work, we always set `ar_r` = `aw_r`, and refer to this value as "QoS level" with the notation Q_l for each regulated ACC_l. Equation 1 can be used to compute the bandwidth in MB/s given Q_l, and the size w in bytes of each transaction initiated by ACC_l.

$$b_{qos,w} = \frac{w \cdot Q_l \cdot f_{clk}}{2^{32}} \tag{1}$$

Figure 3 was obtained by programming the Enhanced DMA (eDMA) on our platform at different QoS levels, and by performing only reads (left), only writes (center), or reads+writes (right). For the eDMA, $w = 4$ bytes. Firstly, unlike MemGuard, QoS regulation operates with a transaction-level granularity (flat lines). Secondly, the read+write traffic once again achieves higher DDR utilization at the same QoS level.

C. DDR Saturation Model under Regulation

We propose a linear model that describes how the different active PEs subject to (MemGuard or QoS) regulation impact the utilization U_{tot} of the DDR subsystem. The proposed model is simple and the actual values of the parameters are described in Section X-C. The model is given in Equation 2:

$$U_{tot} = \sum_{CPU_k} \left(U^{\alpha}_{mg} \cdot \frac{Q_k \cdot L_s}{2^{20} \cdot P} + U^{\beta}_{mg} \right) + \sum_{ACC_l} \left(U^{\alpha}_{qos,w} \cdot Q_l + U^{\beta}_{qos,w} \right) \tag{2}$$

In the equation, CPUs and accelerators are treated differently because they are differently regulated. For CPUs, we first convert the MemGuard budget to the corresponding bandwidth in MB/s — where P is expressed in seconds and L_s represents the size of a cache line in bytes. Then a linear slope U^{α}_{mg} is applied and the initial offset U^{β}_{mg} is added to find the contribution of each CPU_k to the total utilization. For accelerators, instead, we convert directly from QoS level to contribution in utilization with similar parameters $U^{\alpha}_{qos,w}$ and $U^{\beta}_{qos,w}$. These parameters depend on the transfer size in bytes, w, that masters are capable of transferring with each read/write request — recall that ARM QoS only enforces a minimum inter-arrival time on memory requests, regardless of their size.

VII. From Profiles to E-WarP Tasks

In order to instantiate the E-WarP model, the starting point is the profiles acquired on the task under analysis in isolation. Indeed, a large number of runs and corresponding profiles are required to build confidence on the worst-case behavior, like in traditional single-core measurement-based WCET estimation. The profiles are then integrated to build the task envelopes $\mathcal{M}_i$ for the task under analysis. If a task executes on multiple processing elements, then multiple sets of profiles need to be acquired, one per each processing element R_j used by the task. We hereby focus on the definition of the generic M_j for processing element R_j.

Let us first consider a single run and resulting acquired raw profile. A profile is an ordered collection of samples $\{s_r(1), s_r(2), \ldots\}$. Each sample collected by the profiler captures the activity of the task under analysis during an interval of length δ. The smaller the parameter, the more accurate the E-WarP model will be. Moreover, for the model to produce valid predictions on the task's WCET under regulation, it must hold that $\delta < P$. We hereby consider that $\delta << P$ and evaluate how to find a suitable lower limit for δ in Section X-B.

We use the notation $s_r(h)$ to refer to the h-th sample in the r-th run. Each sample collected by the profiler carries the following information. (1) s^r_r number of bytes read during δ; (2) the s^w_r number of bytes written during δ. The profile also contains (3) the $s^u_r \in [0, 1]$ utilization of the DDR controller during the δ time window. The latter information is not stored in the task envelopes, but it is useful to study the saturation point of the DDR controller, as studied in Section X-C.

Algorithm 1 constructs the envelope M_j and also returns the observed task's WCET in isolation from an arbitrary set of runs $\mathcal{R}$ sorted by shortest-to-longest. The logic of the algorithm is simple: (1) we expand the length of the envelope M_j if longer runs are observed (Lines 11-17); and (2) we keep track of the highest and lowest cumulative number of transactions in each run (Lines 18-19). Note that Algorithm 1 only considers read traffic in the profiles, which is the correct way of deriving envelopes when R_j is a CPU. To apply the algorithm to accelerator tasks, it is enough to replace Line 10 with: $x_r \leftarrow \max(s^r_r(h), s^w_r(h))$, to only keep track of the type of traffic that constitutes the bottleneck.

Algorithm 1 Envelope M_j from profiler runs

```
 1: function GetEnvelope($\tau_i, R_j, \mathcal{R}$)
 2:     $L_i \leftarrow 0$
 3:     $M_j \leftarrow \{R_j\}$                      ▷ The first element is the proc. element
 4:     for $r \leftarrow 1, |\mathcal{R}|$ do                              ▷ Consider each run
 5:         $x_r \leftarrow 0$                   ▷ Cumulative num. of transfers in run $r$
 6:         $h \leftarrow 1$                                      ▷ Current sample index
 7:         $L_r \leftarrow 0$
 8:         for $\exists s_r(h), h \leftarrow h + 1$ do
 9:             $L_r \leftarrow L_r + 1$                          ▷ Track length of the run
10:             $x_r \leftarrow x_r + s^r_r(h)$
11:             if $L_r > L_i$ then
12:                 $L_i \leftarrow L_r$                          ▷ Remember longest run
13:                 $x^+_j(h) \leftarrow \max(x^+_j(h-1), x_r)$
14:                 $x^-_j(h) \leftarrow x_r$
15:                 $\sigma_j(h) \leftarrow \{x^+_j(h), x^-_j(h)\}$
16:                 $M_j \leftarrow M_j + \{\sigma_j(h)\}$
17:             else
18:                 if $x_r > x^+_j(h)$ then $x^+_j(h) \leftarrow x_r$
19:                 if $x_r < x^-_j(h)$ then $x^-_j(h) \leftarrow x_r$
20:     return $M_j, L_i \cdot \delta$                   ▷ Return envelope and WCET
```

VIII. Predicting WCETs from Regulation Levels

In this section, we describe how to predict the WCET of tasks for which a memory envelope has been constructed according to Section VII. The key idea is to mimic the behavior of budget-based regulation (for CPU envelopes) or QoS-based regulation (for accelerator envelopes) as we move through the envelope.

Let us first consider CPU envelopes. Given a generic envelope M_j where $R_j = CPU_k$, we use Algorithm 2 to predict the WCET of the task when CPU_k is assigned MemGuard budget Q_k. To be correct in practice, an extra overhead introduced by MemGuard needs to be taken into account. There are two types of overhead involved. The first, namely t_{ovh} is the upper-bound on the extra time overhead introduced by each periodic budget replenishment. Each activation of MemGuard might also pollute some of the cache partition of the application under analysis, leading to extra memory transactions x_{ovh} being budgeted to the task, compared to when it operates without regulation. We incorporate this overhead as a restriction on the budget given to the core under analysis. Hence, Algorithm 2 considers $Q'_k = Q_k - x_{ovh}$.

Intuition: Algorithm 2 returns the predicted WCET by keeping track of the additional time t_{add} due to regulation at quota Q'_k. During every regulation period of length P, the algorithm performs multiple steps through the profile samples. At each step, from a memory bandwidth perspective, the worst-case is when (1) the behavior of the application has followed the lower envelope, i.e. when at the generic sample $h - 1$ its cumulative number of memory transactions is exactly $x^-_j(h-1)$ (Line 16); and (2) at sample h the cumulative number of memory transactions jumps to $x^+_j(h)$. If this difference is greater than Q'_k, (Line 12) then we increase the overall regulation stall. But in doing that, we remember that at least Q'_k transactions were performed by increasing the value of x_{off} which is always considered instead of $x^-_j(\cdot)$ ($\cdot$ refers to an arbitrary sample) when $x_{off} > x^-_j(\cdot)$. This prevents the algorithm from being overly pessimistic. Indeed, by tracking x_{off}, the algorithm captures the worst-case progress of the application as a trajectory somewhere between $x^+_j(h)$ and $x^-_j(h)$.

Correctness: To understand why the algorithm is safe, lets take a closer look. Consider the easy case where the upper-envelope is equal to the lower envelope, i.e. $\forall h, x^+_j(h) = x^-_j(h)$. In this case, it is enough to keep tracking the progression of transactions. If within a regulation period P we observe more transactions than Q'_k, then the extra regulation time is added to the WCET (Lines 12-15). Conversely, if the budget is not exceeded, it is replenished and counting transactions restarts (Lines 9-11). In this case, transactions might suffer a t_{stall} time due to contention, which is accounted (Line 10). This parameter makes the calculation generic and applicable to in-order micro-architectures. In our observations, no visible stall was measured when the saturation of the DDR is kept below 100%, hence we considered $t_{stall} = 0$. Moreover, any carry-in due to misalignments between δ and P needs to be taken into account — see Line 11.

In the more general case, i.e. when $x_j^+(h) \neq x_j^-(h)$, one must consider the case where the task might have been idle (in terms of DDR activity) and then suddenly performs $x_j^+(h) - x_j^-(h-1)$ transactions. If the jump incurs regulation, we add the regulation time but also shift up the lower envelope by Q'_k, always preventing it from exceeding $x_j^+(h)$ — see Lines 15-16.

Algorithm 2 Predict WCET for CPU Envelope

```
1: function GetWCET_CPU($\tau_i, M_j, CPU_k$)
2:     $t_{add} \leftarrow P$                                   ▷ Track time added by regulation, add tail
3:     $x_{off} \leftarrow 0$                                   ▷ Tracks offset of lower envelope
4:     $t_s \leftarrow 0$                                       ▷ Start time of regulation period
5:     $x_s \leftarrow 0$                                       ▷ Transactions at beginning of regul. period
6:     $h \leftarrow 1$
7:     for $\exists \sigma_j(h), h \leftarrow h + 1$ do
8:         $t \leftarrow \delta \cdot h$                        ▷ Advance time
9:         if $t - t_s \geq P$ then                             ▷ No regulation
10:            $t_{add} \leftarrow t_{add} + t_{stall} \cdot x_s + t_{ovh}$    ▷ Add stall due to contention
11:            $t_s \leftarrow t - ((t - t_s) - P)$             ▷ New beginning of regulation period.
12:        if $x_j^+(h) - x_s \geq Q'_k$ then                   ▷ Budget exceeded
13:            $t_{add} \leftarrow t_{add} + P - (t - t_s) + t_{ovh}$    ▷ Add regulation stall
14:            $t_s \leftarrow t$
15:            $x_{off} \leftarrow \max(x_{off}, x_j^-(h)) + Q'_k$    ▷ Track offset on lower env.
16:        $x_s \leftarrow \min(x_j^+(h), \max(x_j^-(h), x_{off}))$ ▷ New initial number of trans.
17:    return $t_{wcet} = t + t_{add}$                          ▷ Predicted WCET
```

To compute WCET predictions on envelopes defined on accelerators, i.e. when $R_j = ACC_l$, we follow a similar yet different strategy. The root of the difference is that QoS-based regulation is performed at the granularity of individual transactions. Hence, computing the length of the envelope with QoS regulation can be done by reusing principles from Network Calculus [40]. QoS-regulation corresponds to traffic shaping under a service curve $\beta(t) =$ with rate equal to the QoS level Q_l = ar_r = aw_r; where the maximum difference between $x_j^+(h)$ and $x_j^-(h)$ is taken as the initial burstiness (b) of the arrival curve $\alpha(t)$, and the sub-additive closure of the upper-envelope as the arrival curve.

In the network calculus framework, shaping is computed as the min-plus algebra convolution between the arrival curve and the service curve. However, the computation on the WCET of the envelope under QoS regulation can be performed in linear time following a strategy similar to Algorithm 2. We omit the full algorithm due to space constraints.

If a task τ_i runs only on a CPU_k, then the new WCET $C_i(Q_k)$ under regulation with budget Q_k can be computed by invoking Algorithm 2. In this case, schedulability can be checked using the traditional partitioned fixed-priority scheduling with preemptions, as long as preemptions are restricted to occur only at the boundaries of regulation periods. If preemptions can occur, however, care must be taken in adding the additional overhead in terms of extra memory transactions performed by τ_i due to cache-related preemption delay (CRPD) [41], [42].

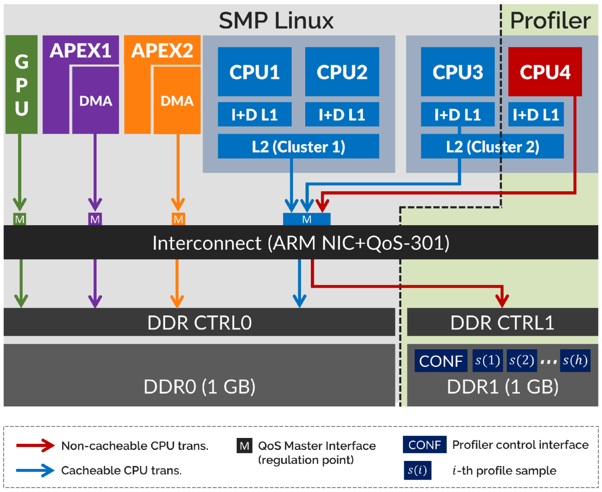

Fig. 4. Block diagram of the main PEs and memory modules in the NXP S32V234 platform. The division between computation and profiling sub-shell is highlighted.

However, if a task assigned to CPU_k also uses an accelerator ACC_l, then we assume it will be blocked on CPU_k while it executes on ACC_l. From a CPU scheduling point of view, the time it takes for ACC_l to return control to CPU_k is a self-suspension interval. τ_i's worst-case response time can be computed leveraging the results in [43]. To compute the overall WCET of τ_i $C_i(Q_k, Q_l)$ subject to regulation on CPU_k with budget Q_k and with ACC_l subject to QoS-based regulation at level Q_l, the following needs to be computed. First, we compute the stall due to regulation on CPU_k as $t_k^{stall} = C_i(Q_k) - C_i$ computed using Algorithm 2; next, we compute $t_l^{stall} = C_i(Q_l) - C_i$ using the equivalent of Algorithm 2 for QoS regulation. Finally, $C_i(Q_k, Q_l) = C_i + t_k^{stall} + t_l^{stall}$.

IX. Implementation

We have performed a full-system implementation that includes a low-overhead, high-accuracy profiler, and a partitioning hypervisor augmented to support ARM QoS features. Our implementation was carried out on the NXP S32V234 [36] embedded platform. The main hardware blocks are presented in Figure 4. The SoC features 4 ARM Cortex A53 CPUs operating at a clock frequency of 1 GHz ($f_{cpu} = 1 \times 10^9$ Hz) divided into two clusters. Each core has a private 32 KB+32 KB I+D L1 cache, and a 256 KB L2 cache is present in each cluster. Because this platform is designed for vision applications, it also integrates two accelerators. The first is a programmable GC3000 GPU [44] and the second is the APEX-CL Image Cognition Processor, or APEX for short. The device contains two identical instances of the APEX engine, namely APEX0 and APEX1. This accelerator promises to deliver “high-performance parallel processing capability” [36]. The APEX are highly complex processing subsystems that include scalar and vector processing units, local scratchpad memories, and DMA engines. We focus on the APEX in our evaluation as a realistic instance of a high-performance accelerator.

The platform features two DDR controllers, namely DDR0 and DDR1, that operate independently on two separate portions of DRAM memory of 1 GB each. The controllers operate at $f_{clk} = 0.5$ GHz and have a bus width of 32 bits. Importantly, each controller exposes a set of memory-mapped performance counters that report: (1) the number of DDR cycles elapsed tot_ddr_cyc; (2) the number of busy DDR cycles busy_ddr_cyc; (3) the total number of bytes transferred in read (rd_bytes) and (4) in write (wr_bytes) transactions. The DDR profiling interface also allows defining a filter on the source of traffic (e.g. CPU cluster 1, APEX1, etc.) that is applied when counting read/write bytes. To differentiate between the traffic coming from different masters, counters (3) and (4) can be programmed to only filter the traffic coming from a specific master(s) based on their AXI-ID.

The last component that requires some introduction is the interconnect. The S32V234 system uses a standard ARM NIC-301 [45] with ARM QoS-301 [38] extensions. The QoS extension of the NIC is where traffic regulation is performed on traffic that traverses the interconnect towards DDR. ARM QoS extensions are surprisingly, broadly available in many current-generation ARM-based platforms. When we started this work, we were surprised to discover that little-to-no software support or research literature was available on these modules. So we had to implement our own to carry out this research. The NIC+QoS-301 provides a memory-mapped interface to control the regulation parameters on a per-master basis. Regulation interfaces are depicted as colored squares on top of the NIC in Figure 4. Because the traffic from all the CPUs arrives through the same master interface, QoS regulation cannot be used to regulate individual CPUs, but only the total traffic from all the CPUs. Conversely, it allows one to set individual regulation regimes for each of the APEX, for the GPU (see Figure 4), for the DMAs, for the network interface and the I/O sub-shell (not shown).

We use the Jailhouse partitioning hypervisor [46] to partition resources in our system. Jailhouse is the ideal choice for this type of implementations because it does not perform scheduling of virtual CPUs (VCPUs), it is lightweight and easy to port/modify, includes support for cache coloring and DRAM bank partitioning [47], and is open-source. It also includes libraries to define bare-metal guest-OS that can be launched directly on a subset of the CPUs. Unfortunately, Jailhouse was not ported to the NXP S32V234 platform at the time we started this work. Our first implementation tasks concerned writing a layer of SoC-dependent code to port Jailhouse onto the target platform. Doing so required a few modifications to the stock boot-loader(u-boot), and to the CPU hotplug support in the Linux kernel[1]. It also involved writing a driver for the LINFlexD device in the S32 that controls the console outputs.

Next, we integrated into our porting an implementation of MemGuard originally proposed in the context of the HERCULES project [48]. We also implemented from scratch a platform-independent support for ARM QoS features, along with the platform-specific code to setup QoS regulation in the S32V234 system. With the implemented support, system designers can set multiple QoS parameters for multiple masters in a single hypercall, making the interface suitable for efficient online dynamic QoS management.

Finally, we implemented a profiler that is comprised of two parts: a low-level profiler, profvm, and a user-space control toolkit, profctl. First, profvm is a small-footprint bare-metal guest-OS that can be loaded by Jailhouse. To meet the stringent accuracy and transparency requirements of our profiler, we proceeded as follows. When loaded, profvm takes exclusive ownership of a single CPU (CPU4), and of an entire DDR controller (DDR1). Our profvm uses the dedicated 1 GB of DRAM memory for two purposes. (1) It exposes a shared command&control interface; and (2) when active, stores a sequence of samples of DDR0 activity. The other three CPUs are assigned to Linux in SMP mode and are used to run the user-space applications that need to be profiled. When active, profvm performs periodic sampling of DDR0 at a configurable sampling rate expressed in CPU clock cycles. Each sample collected in DDR1 contains the values, and the time of sampling, of: (1) CPU cycles counter, (2) value of tot_ddr_cyc, (3) value of busy_ddr_cyc, (4) value of rd_bytes and (5) wr_bytes. The ratio between (3) and (2) provides the instantaneous utilization of the DDR subsystems. Some porting was also required to ensure that the APEX driver does not attempt to use any memory space in DDR1. This is because the out-of-the-box drivers execute APEX code from the memory space of DDR1 controller.

Second, to facilitate profile acquisition, the profctl toolkit is provided. It takes care of all the low-level coordination with the profvm module; launches the benchmark(s) to be profiled; and at the end of the experiment gathers samples from DDR1 to save them to disk for later analysis[2]. Multiple parameters can be configured directly from profctl, most prominently sampling period, and filter on individual masters.

Even with all the changes mentioned above, two important features are needed to port E-WarP to another hardware. (1) Profiling: The requirements for such a profiling tool are discussed in detail in Section V. (2) Bandwidth Control: MemGuard is a widely-implementable technique and ARM QoS extensions are drop-in modules (ARM QoS-310/QoS-400) bound to increase in popularity. Another tool, ARM Memory System Resource Partitioning and Monitoring (MPAM) [49] combines shared cache, memory, and interconnect bandwidth management.

X. Validation and Evaluation

In this section, we first build a set of experiments to identify key parameters in our system. Next, we discuss how we instantiated the E-WarP model on real-world applications and evaluate the WCET predictions under regulation. Then an in depth analysis of QoS-based controls for accelerators is provided. Finally, we present a full-system integration where all the applications analyzed in isolation on the CPU and the accelerators are deployed to run in parallel.

A. Experimental Setup

We used the NXP S32V234 [36] platform introduced in Section IX. A combination of synthetic and real benchmarks are used to gain insight into the platform. The synthetic benchmarks used to stress/evaluate specific parameters of our platform are described in the corresponding subsections. For our real benchmarks, we use a subset of the benchmarks in the San-Diego Vision Benchmarks (SD-VBS) suite [50]. Because we are interested in applications that are DRAM-bound, the selection was performed by taking all the benchmarks that operate on images. These come with different input sizes, but we have excluded the FullHD inputs which lead to impractically long runtimes. We instead focus on the next two largest sizes, i.e. VGA and CIF. The full list of selected benchmarks is reported in Table II.

In terms of accelerators, we focus on the APEX engine included in the S32 platform. The S32 features two independent

[1] This was required to overcome the lack of a PSCI firmware provided by the vendor to control CPU shutdown.

[2] https://github.com/rntmancuso/jailhouse-rt

APEX accelerators. Each accelerator is fully programmable and includes a high-performance parallel processing unit (APU) for vector and scalar operations, a DMA, and internal scratchpad memories to operate on data/image tiles. The ARM QoS control interface instantiated on this platform allows setting regulation parameters on the main bus independently for the two APEX engines. The selection of benchmarks available for this unit is limited to the examples released by the manufacturer. We were able to fully integrate the APEX within our profiling infrastructure. But the benchmarks we observed insisted on the processing capabilities of the engine as opposed to generating a lot of DDR traffic. We focus our evaluation on the most DRAM-intensive one we found, i.e., the "Region of Interest" (RoI) benchmark. The RoI benchmark processes different parts of the image on APEX.

For consistency, we always activate the Jailhouse hypervisor. As most of our experiments involve the use of the presented profiler, the `profvm` bare-metal VM is generally loaded (unless specified otherwise) and pinned to core 4. Linux v4.19 is deployed on the other 3 CPUs. Some minor modifications to the kernel were performed to port Jailhouse and to enable support for the APEX. The kernel is compiled in full-tickless (`NO_HZ_FULL`) mode. All the benchmarks are always deployed using the `SCHED_FIFO` scheduler and with explicit pinning to CPUs. We use the `profctl` to synchronously launch multiple benchmarks in parallel and to coordinate profiling and collection of execution times. All the min/max/avg statistics were calculated on 30 runs for each configuration, to remain statistically significant.

B. Profiler Transparency and Accuracy

As a first experiment, we evaluate how well the proposed profiler satisfies the transparency and accuracy requirements.

The accuracy was evaluated along two sub-dimensions. First, we evaluated how closely the obtained profile matches the expected number of read/write bytes in a synthetic benchmark of known characteristics. To limit the number of spurious DDR transactions in the experiment, we (1) program the platform DMA (eDMA) engine to transfer a known number of bytes; (2) leverage the filtering capabilities of our profiler to only capture eDMA transactions. The resulting profiles cumulative number of read/write bytes were in perfect match with the synthetic benchmark.

Next, we want to find a suitable value for δ that directly relates to the profiler's accuracy. To do so, we varied the configuration of `profvm`'s sampling period and selected the smallest number of CPU clock cycles that leads to a measurement error no larger than ±2 clock cycles[3] with 99.99% confidence over 100,000 consecutive measurements. Setting 1,500 clock cycles as the sampling period of `profvm` satisfies this specification. This value was used in all the experiments. With this setting, each acquired sample captures the behavior of the DDR subsystem within a $1.5\mu s$ window. The profiler operates $1,500\times$ faster than MemGuard, so it holds that $\delta << P$.

Lastly, we evaluated the impact of the profiler on all the selected SD-VBS applications. We first capture the runtime of a benchmark executing without the `profvm` loaded in the system. The runtime is then compared to the case where `profvm` is loaded and configured to collect the profile of the application under analysis. On average across all cases, we observed a runtime increase of 0.33%, with a maximum of 1.67%. Since the profiler is designed to bypass the shared cache and only interact with a private DDR controller, the overhead necessarily arises at the shared interconnect. Because the profiler is not required in production, this overhead will not affect the final applications and all the WCET predictions will still be safe.

C. DRAM Controller Saturation

In this section, we study the saturation of the DDR controller under MemGuard and QoS regulation with the goal of establishing appropriate values for the U^{α}_{mg}, U^{β}_{mg}, $U^{\alpha}_{qos,w}$, $U^{\beta}_{qos,w}$ parameters discussed in the previous sections.

1) MemGuard Regulation: We first establish a relationship between MemGuard budget assigned to a CPU, the resulting bandwidth extracted from the DRAM, and the measured DRAM utilization. Because we are interested in an upper-bound on the utilization, it is important to design an experiment where the DDR utilization is maximized at the selected budget. It is already clear from Figure 2 that performing stores achieves higher utilization at the same level of budget. Furthermore, following the analysis in [51] we want to make sure that each DRAM transaction performed by our benchmark results in a DRAM row miss.

With this in mind, we consider the mapping of physical addresses to DRAM coordinates (banks/rows/columns), and design the USTRESS synthetic benchmark. USTRESS allocates in user-space a 2 MB buffer that is contiguous in physical memory leveraging standard support for huge-pages (`MAP_HUGETLB`). It then performs the first store on column 0 and row 0. The next store is performed 2^{15} bytes away — because the first row bit is bit 15. This pattern keeps all the accesses on bank 0. Once we reach the last accessible row, we set the column offset to 64 bytes and restart from row 0 to fetch the second cache line in the first row. We proceed by scanning all the rows (inner loop) and then increasing the column offset (outer loop) until reaching the last accessible column of the last row. This pattern not only always accesses a closed row in the same bank, but it also bypasses the cache and ensures that no prefetching is performed because subsequent accesses cross the 4 KB page boundary.

We then profile USTRESS subject to variable regulation enforced with MemGuard. We compare the theoretical bandwidth that should be extracted with what is observed in the profiles. Simultaneously studying the trend of DDR utilization as returned by the profiles. The results are shown in Figure 5. As predicted by our model in Equation 2 for cache line size $L_s = 64$ bytes, the utilization grows linearly as the extracted bandwidth increases. At bandwidth 950 MB/s (budget = 15565) the controller is running at 97% utilization. At the next budget value we considered (budget = 16384), 100% utilization is reached, and the observed bandwidth starts to level-off and deviate from the linear trend. Hence we consider 950 MB/s to be a safe bound on the cumulative budget that can be extracted by the CPUs without saturating the DDR controller. By finding the angular coefficient and $y-$intercept of the utilization trend before saturation, we can set $U^{\alpha}_{mg} = 6.23856 \times 10^{-3}$ and $U^{\beta}_{mg} = 6.68742 \times 10^{-2}$.

2) QoS-based Regulation: We conducted a similar analysis to the previous case, but this time we use two accelerators, to study the relationship between extracted bandwidth and DDR utilization with 4-byte and 128-byte transfers, respectively.

[3] The DRAM operates at half the frequency of the CPUs.

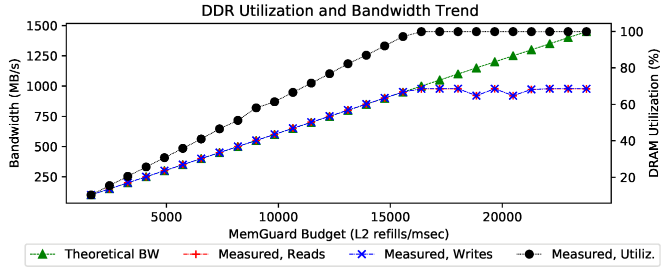

Fig. 5. DDR controller saturation analysis. Bandwidth grows lineraly for reads and writes and close to theoretical value as long as the utilization remains below 100%.

TABLE I
MEMGUARD BUDGETS (Q_k) AND QOS LEVELS (Q_l) WITH THE CORROSPONDING BANDWIDTH AND UTILIZATION

Setting		Bandwidth (MB/s)		DDR Utilization (%)	
MemGuard	QoS	MemGuard	QoS	MemGuard	QoS
492	5	30.03	74.51	3.14	15.68
819	10	49.99	149.01	5.18	30.73
1475	20	90.03	298.02	9.27	60.83
2130	40	130.00	596.05	13.36	121.02
4096	80	250.00	1192.09	25.62	241.41
5734	100	349.98	1490.12	35.84	301.61
7373	160	450.01	2384.19	46.06	482.2
9830	320	599.98	4768.37	61.39	963.76

First, we use the eDMA module to generate read+write access patterns at various levels of QoS regulation. The eDMA is configured to generate 32-bit-wide transfers. Unfortunately, the eDMA is not very efficient and hence it cannot bring the DDR to 100% utilization. Indeed, the eDMA does not incur any slowdown when regulated at QoS level 40 and above. Hence, we use the regulation levels between 5 and 20 to establish the linear relationship between QoS levels and utilization. The behavior of the eDMA at these regulation levels was already shown in Figure 3 — see the rightmost subplot for the read+write case. Once again we observe a linear trend between QoS levels and DDR utilization, and we identify the following parameters: $U^{\alpha}_{qos,4} = 2.05867$ and $U^{\beta}_{qos,4} = -0.383333$. We repeat the same type of experiment using the APEX engine which transfers 128-bytes with each transaction. This allows us to set the parameters $U^{\alpha}_{qos,128} = 3.00978$ and $U^{\beta}_{qos,128} = 0.632288$.

We provide in Table I a recap on the relationship between MemGuard budgets, QoS values, and upper-bounds on extracted bandwidth and the impact on DDR utilization.

D. MemGuard Regulation — Practical Quirks

Before moving on to instantiating our E-WarP on real applications, a couple of aspects need to be clarified. These are CPU regulation overheads and limitations to what can be regulated. Both these aspects have been overlooked in previous works because they were hard to evaluate. We were able to user our profiler to evaluate these.

In terms of overhead, we mentioned that MemGuard introduces two types of overheads, i.e. t_{ovh} and x_{ovh}. We designed two synthetic tasks to evaluate these quantities. To evaluate t_{ovh}, we implemented a task that defines a buffer smaller than the L1 cache size, and that continuously samples the CPU cycle counter, storing the difference between two successive samples in the buffer. Because the benchmark does not generate DDR traffic, it will not be regulated by MemGuard. It will however be interrupted by the periodic interrupt used for budget replenishment. To discover the end-to-end overhead, we then look for discontinuities in the sampled time deltas. With this, we measured the overhead of our Jailhouse implementation of MemGuard to be up to 450 cycles. This is also in line with [52] and we set $t_{ovh} = 450/1.0\ GHz = 4.5 \times 10^{-4}$ ms.

To compute x_{ovh}, we rely on the profiler. We created a benchmark that allocates a buffer of the same size as the last-level cache — 256 KB. Like in USTRESS, the buffer is placed contiguously in physical memory to control cache-set conflicts. When this benchmark is profiled, we observe small spikes of memory transactions at the periodicity of MemGuard activations. By counting these transactions on a per-period basis, we computed $x_{ovh} = 35$ transactions.

Another phenomenon we observed by analyzing the profiles of some of our benchmarks is unregulated CPU activity. MemGuard, as well as later implementations like the one in [9], rely on the `L2_CACHE_REFILL` event to count transactions. Clearly, a CPU can perform DRAM transactions that are not counted by this event by accessing non-cacheable memory, or when performing cache maintenance operations — e.g. a cache flush. Fortunately, these operations are not common in user-space applications. But there exists a class of instruction routinely used in user-space applications that behave in a similar way. Instruction like `STM` (in ARM aarch32) and `STP` (in ARM aarch64) that might be treated as write-no-allocate, which bypass the cache and generate DRAM write transactions. Common operations such as `memset` are implemented using these instructions. We have modified our benchmarks to avoid the use of the problematic instructions.

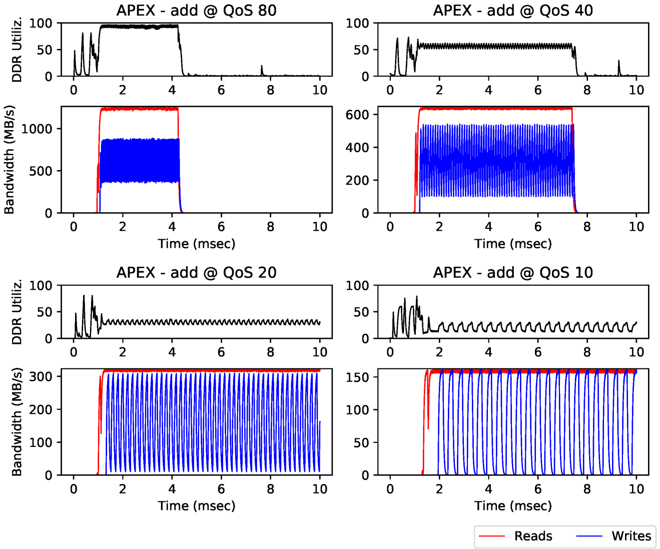

Fig. 6. Profile of QoS-regulated APEX computation over the ADD benchmark. From left-to-right, top-to-bottom, the regulation levels are 80, 40, 20, and 10.

E. QoS Regulation — Adherence to Single-bottleneck Model

As we mentioned in Section VI-B, we assume that at any point in time, the workload executing on an accelerator is bottle-necked by either read or write operations. This is trivially true if the accelerator first copies a block of inputs, computes the result locally, and then writes back the result to memory.

But some accelerators might perform stream-processing, with interleaved reads and writes, where this assumption is less obvious. If the assumption holds, when the traffic that represents the bottleneck is regulated, the other type of traffic will also slow down. This assumption allows us to reason on a single upper-envelope curve and to predict the effect of QoS regulation on the combined read/write traffic. We hereby validate this assumption.

TABLE II
CPU BENCHMARKS — PREDICTION AND OVERESTIMATION

B.mark	In	Budgets									
		492		983		1475		1966		2458	
		Pr. (s)	+ %	Pr. (s)	+ %	Pr. (s)	+ %	Pr. (s)	+ %	Pr. (s)	+ %
sift	cif	5.28	22.51 %	3.19	13.32 %	2.6	8.53 %	2.31	7.44 %	2.15	6.46 %
	vga	13.91	9.45 %	8.74	4.55 %	7.23	3.08 %	6.5	2.16 %	6.08	2.9 %
disparity	cif	10.68	5.53 %	5.3	2.57 %	3.57	1.74 %	2.72	2.07 %	2.21	2.69 %
	vga	29.11	4.49 %	14.39	1.65 %	9.65	1.07 %	7.32	0.74 %	5.93	1.13 %
mser	cif	1.79	1.72 %	0.91	0.98 %	0.63	0.94 %	0.49	0.38 %	0.42	0.5 %
	vga	8.23	18.82 %	4.14	15.41 %	2.83	13.49 %	2.19	12.76 %	1.82	13.24 %
tracking	cif	2.13	8.9 %	1.16	4.81 %	0.85	3.46 %	0.71	2.47 %	0.63	2.01 %
	vga	7.2	23.69 %	3.92	18.9 %	2.89	16.51 %	2.39	14.92 %	2.13	14.34 %
localiz.	cif	1.16	10.9 %	1.02	3.82 %	0.99	2.12 %	0.97	1.5 %	0.97	1.73 %
	vga	4.48	5.28 %	3.7	2.02 %	3.5	1.8 %	3.41	1.1 %	3.38	0.83 %
tex_synth	cif	0.43	14.16 %	0.3	7.27 %	0.26	5.13 %	0.24	2.92 %	0.24	2.62 %
	vga	1.42	10.35 %	1.1	3.04 %	1.01	1.51 %	0.98	0.62 %	0.97	0.55 %
stitch	cif	1.42	9.69 %	1.09	3.38 %	1	2.11 %	0.96	1.27 %	0.93	0.74 %
svm	cif	1.73	6.19 %	1.58	1.65 %	1.55	1.07 %	1.53	0.97 %	1.53	0.87 %
						MAX	23.69%	MIN	0.38%	**AVG**	**5.71%**

We study the profile of a simple benchmark, namely ADD, deployed on the APEX accelerator, and performing streaming vector additions. Because the benchmark reads in input, two operands for each unit of output, we expect the read bandwidth to be the bottleneck. Figure 6 displays the activity in DDR of this benchmark at different levels of QoS regulation. First, we note that at QoS 80 (top-left) the APEX is able to saturate the DDR and that, as predicted, it is bottle-necked by read operations. As we lower the QoS level to 40, 20, and 10 it can be noted that (1) the extracted write bandwidth also drops; and (2) that the overall length of the operation is dictated by the bottleneck traffic.

While all the benchmarks we observed on the target platform behaved in a similar way, we do not exclude that workload violating this assumption could be defined. The model presented in this paper does not directly apply to such cases.

F. E-WarP Instantiation and Prediction — CPU Tasks

Having identified key system parameters and having assessed the validity of fundamental assumptions, we present the results obtained by instantiating the E-WarP model on the CPU-only SD-VBS applications in this section. In all the predictions presented in this section, 30 runs/profiles of execution were obtained for each benchmark in isolation and without regulation. Then, we produce a prediction for each budget in Table I using Algorithm 2. We then run the benchmark under regulation at each of the selected values, and compare our predictions against the maximum runtime observed under regulation.

The full list of results is summarized in Table II. For each benchmark/input size we report the predicted time in seconds ("Pred. (s)") column and the overestimation percentage ("Incr. (%)") compared to the longest run observed under the considered budget. We only report numbers for the lower values of budgets because they are where predictions become worse. However, we have carried out our predictions on the full range of considered budgets and confirmed that the obtained WCET always upper-bounds the experimental observations.

We visualize the memory envelope obtained on the application that led to higher overestimation, namely TRACKING with input VGA in Figure 7. Differently from most of the other applications we studied, the upper and lower envelopes create a visible gap, which forces the prediction to be more pessimistic. As part of future work we want to explore how much these curves diverge, and how that affects predictions, when different inputs are provided in different runs.

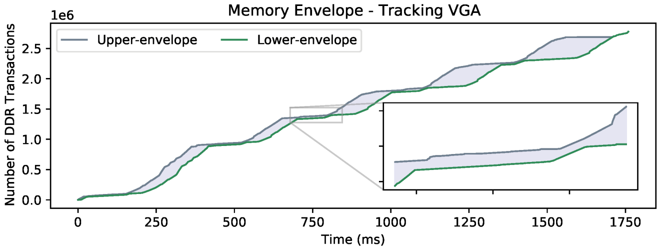

Fig. 7. Upper- and lower-envelope computed over 30 runs of the tracking benchmark.

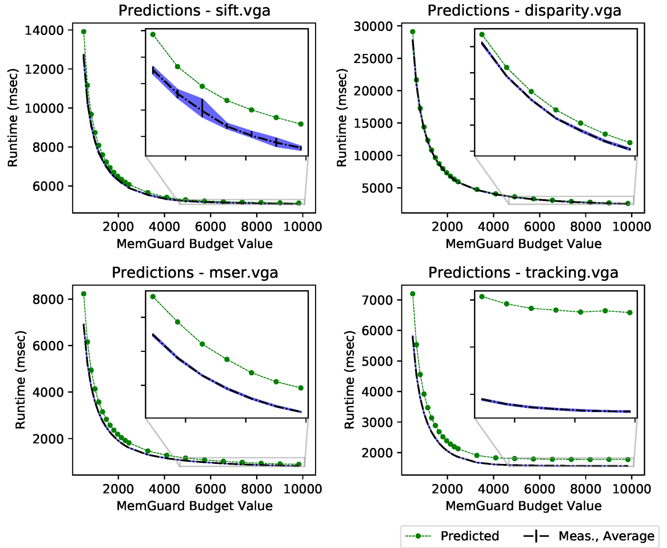

Fig. 8. Prediction v/s measured runtime for SIFT, DISPARITY, MSER, and TRACKING benchmarks with VGA input size. Zoomed in where curve flattens.

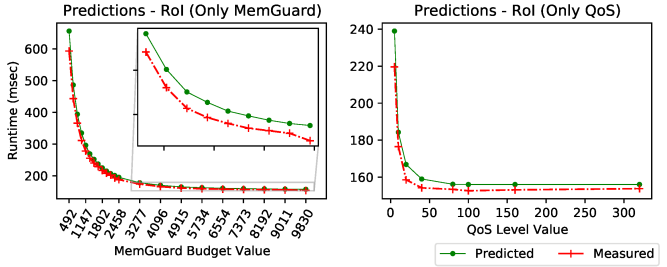

Fig. 9. Comparison of measured runtime of ROI benchmark under MemGuard-only regulation for CPUs (left) and QoS-only regulation for APEX (right).

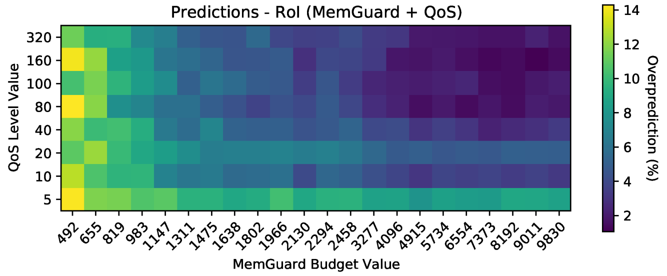

Fig. 10. Comparison of measured runtime of ROI benchmark under both MemGuard- (CPUs) and QoS-based regulation (APEX).

In order to understand how precisely the runtime of CPU tasks can be predicted following the E-WarP methodology, we refer to Figure 8. Here, we plot the trend of our predictions against the timing observed in the actual runs. The blue area represents the min-max error range around the average. We provide insets in each sub-plot to zoom in on the portions that are otherwise harder to appreciate. All in all, what stands out is that our prediction remains extremely close to the observed runtimes, with an average over-prediction below 6%

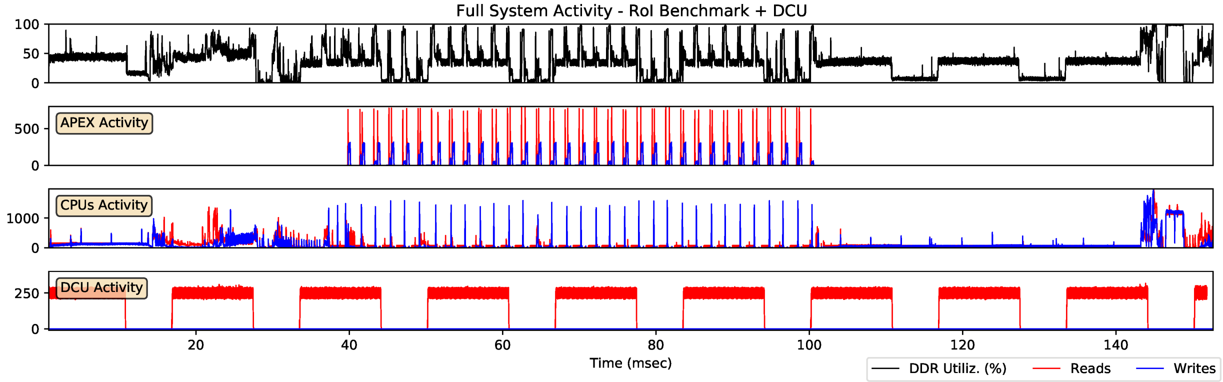

Fig. 11. Full system DDR activity. ROI benchmark in execution. Utilization on top; APEX, CPUs and DCU activity in the other sub-plots.

for budgets between 492 and 2415 (Table II); and with max an average over-prediction of 13.6% and 3%, respectively, for budgets in the mid-range 4915—9830 (not shown in Table II).

G. E-WarP Instantiation and Prediction — Accelerator Tasks

We now consider the most memory-intensive benchmark, i.e. ROI, available for the APEX accelerator. With ROI, we perform the combined analysis for a task that runs in intermittent phases on the CPU and the APEX. The profile of CPU and APEX activity in DRAM is depicted in Figure 11. The figure was obtained by acquiring three profiles of ROI, in each profile, we filter traffic by the AXI-ID either APEX, the CPU, or the Display Control Unit (DCU). More details about the DCU are provided in the next section.

We first perform prediction of the runtime of the benchmark when no QoS-based regulation is enforced on transactions from the APEX block, while the CPU activity of the benchmark is regulated at different levels considered thus far. The results for this experiment are reported in the left plot of Figure 9. Once again, we observed that the predictions remain very close to the maximum measured runtimes, with 10.58% max over-prediction at the lowest budget and 4.15% on average across all the considered budget values.

We then studied our prediction on the same benchmark but when only the APEX is regulated using QoS, while the CPU is not throttled. The results for the QoS values considered in Table I are reported in the right sub-plot in Figure 9. Once again, with E-WarP we are able to predict with good accuracy the total execution time of the benchmark. The maximum over-prediction was 8.82%, while the average sat at 3.62%.

Next, we evaluate the accuracy of E-WarP predictions over all the possible pairs of QoS values and MemGuard budgets in Table II. We visualize the results in terms of over-prediction as a heat-map in Figure 10, where CPU regulation levels and APEX regulation levels are varied on the x- and y-axis, respectively. As expected, highest levels of throttling for both CPU and APEX lead to the largest over-estimations — around 14%, bottom-left corner. Conversely, over-estimation is below 2% for high QoS and budget values (top-right corner).

H. Full System Integration

In our last experiments, we tie everything together. We consider a production-like system with CPU applications running on all the cores, and hence without the profiler. We use one of the cores (CPU 1) to execute the ROI benchmark; we execute the MSER (or TRACKING) benchmark on (CPU 3) with VGA input; and instantiate two memory bombs continuously performing DDR transactions on CPUs 2 and 4.

First, we need to determine suitable budget and QoS levels, since the unconstrained system easily drives the DDR to 100% utilization. In this system, when all the drivers from the manufacturer have been loaded, the DCU becomes active. The role of the DCU is to transfer display frames from the frame-buffer in DDR, to the display port. The DCU is active even without a display connected to the I/O port. Our profiler was able to reveal the presence of this spurious activity, which is visible in the bottom plot of Figure 11, which underlines the importance of having a tool like the one proposed in this work when working on modern complex embedded platforms. While it is possible to disable the DCU, we believe that an active DCU makes for a more realistic setup. Hence, we conduct our experiments by simply accounting for its impact on the utilization of the DDR subsystem.

We measure the upper-bound on DDR utilization caused by the DCU at 36%. Unfortunately, the DCU cannot be regulated using QoS. To reduce the number of parameters, we also set the QoS level for the APEX at 10, so that the APEX can increase the DDR utilization by at most 30.73%. From Section VI-C, we know that a safe utilization is 97%. Hence the cores need to be assigned MemGuard budgets so that they increase the DDR utilization by no more than 30.27%, which corresponds to a total budget of 4915 (about 300 MB/s). With 4 active CPUs, and by performing even division of this quota, we expect that the DDR remains below the saturation threshold as long as the individual CPU budgets remain below 1228.

In Figure 12 (resp., Figure 13), we plot what happens to the runtime of the CPU tasks, i.e. ROI and MSER (resp., TRACKING) with VGA input as we increase the budgets on the CPUs. The black solid line tracks the predicted DDR utilization, with the 100% threshold marked with dashed line. Solid blue lines are used to plot the maximum observed runtime of the MSER (resp, TRACKING), with our predictions depicted in the same color and dashed lines. The same convention using red lines is used to plot the runtime of the ROI benchmark. The areas under the blue/red curves captures the difference between observed maximum and average runtimes. Three main characteristics stand out in the figures. (1) In both, the maximum runtimes correctly remain below the predicted WCETs until 100% DDR utilization is reached, which confirms the validity of the E-WarP approach. (2) Once the saturation point is exceeded, the behavior of the system is highly unpredictable, with our benchmarks experiencing large swings in execution times that are not mitigated by increasing the CPU budgets. (3) In the system with TRACKING, the benchmarks behave erratically slightly later than the predicted saturation point. This is possible because the proposed utilization model has to be conservative to be safe.

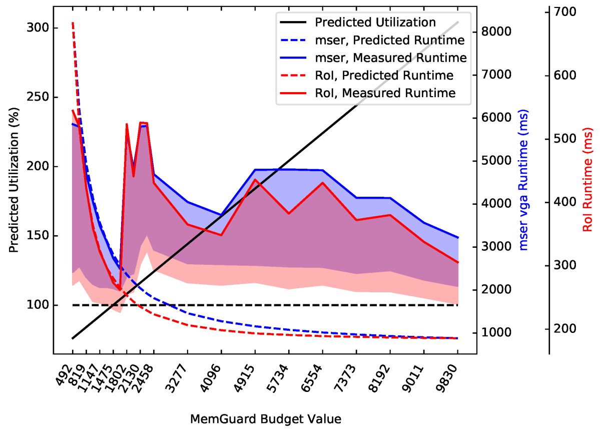

Fig. 12. Full system setup with RoI, MSER and two bandwidth intensive synthetic applications applications on CPU 1, 2, 3, and 4, respectively.

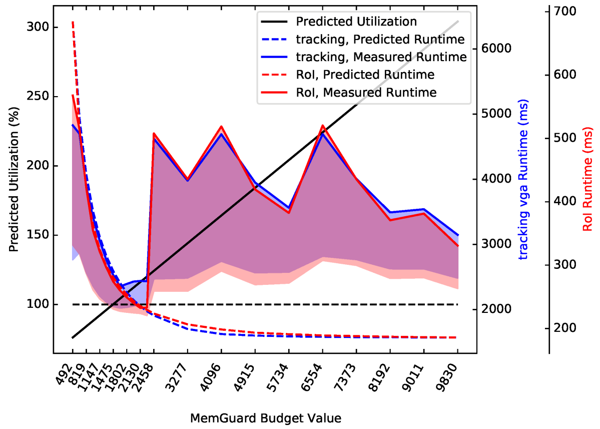

Fig. 13. Full system setup with RoI, TRACKING and two bandwidth intensive synthetic applications applications on CPU 1, 2, 3, and 4, respectively.

XI. Conclusion and Future Work

This work presented E-WarP, a framework of technologies to profile and bound the temporal behavior of workload on CPUs and accelerators. E-WarP achieves full-system memory bandwidth management by integrating two broadly available regulation mechanisms. We design and implement a fine-granularity, transparent profiler. We show how to build relationships between regulation levels and DDR saturation. Finally, we experimentally demonstrate that the formulated WCET predictions hold as long as the main memory subsystem remains below its saturation threshold.

E-WarP is meant to be a stepping stone for profile-driven real-time application analysis with realistic upper-bounds on application runtimes. It enables important future research avenues in directions that include: (1) optimally setting regulation parameters leveraging the convexity of the E-WarP's predictions; (2) performing WCET impact-aware dynamic regulation control in the OS; and (3) integrating our profile-driven approach with formal DRAM models for provable performance guarantees.

Acknowledgments

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. The work was also supported through the Red Hat Research program. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Outstanding paper award: Making shared caches more predictable on multicore platforms," in *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 157–167.

[2] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: a dynamic cache partitioning system using page coloring," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014, pp. 381–392.

[3] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 155–166.

[4] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 145–154.

[5] R. Pellizzoni and H. Yun, "Memory Servers for Multicore Systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–12.

[6] C. A. S. Team, "Multi-core Processors Position Paper," November 2016, accessed on 07.01.2020.

[7] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems," *ACM Comput. Surv.*, vol. 52, no. 3, Jun. 2019. [Online]. Available: https://doi.org/10.1145/3323212

[8] 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)*, 2013, pp. 55–64.

[9] H. Yun, W. Ali, S. Gondi, and S. Biswas, "BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms," *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1247–1252, 2017.

[10] A. Agrawal, R. Mancuso, R. Pellizzoni, and G. Fohler, "Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 230–241.

[11] M. Hassan and R. Pellizzoni, "Analysis of Memory-Contention in Heterogeneous COTS MPSoCs ," in *(ECRTS2020)*, 2020.

[12] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "Wcet(m) estimation in multi-core systems using single core equivalence," in *2015 27th Euromicro Conference on Real-Time Systems*, 2015, pp. 174–183.

[13] G. Yao, H. Yun, Z. P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha, "Schedulability Analysis for Memory Bandwidth Regulated Multicore Real-Time Systems," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 601–614, 2016.

[14] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, 2018, pp. 1651–1657.

[15] H. Kim and R. Rajkumar, "Real-time cache management for multi-core virtualization," in *2016 International Conference on Embedded Software (EMSOFT)*, 2016, pp. 1–10.

[16] M. Pagani, A. Balsini, A. Biondi, M. Marinoni, and G. Buttazzo, "A Linux-based support for developing real-time applications on heterogeneous platforms with dynamic FPGA reconfiguration," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*, 2017, pp. 96–101.

[17] P. Houdek, M. Sojka, and Z. Hanzálek, "Towards predictable execution model on ARM-based heterogeneous platforms," in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*. IEEE, 2017, pp. 1297–1302.

[18] Y. Li, K. Akesson, and K. Goossens, "Architecture and analysis of a dynamically-scheduled real-time memory controller," *Real-Time Systems*, vol. 52, no. 5, p. 675–729, 9 2016.

[19] D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 274–279.

[20] P. K. Valsan and H. Yun, "MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore Based Embedded Systems," in *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, 2015, pp. 86–93.

[21] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable SDRAM memory controller," in *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007, pp. 251–256.

[22] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, Philadelphia, PA, USA, April 2013, conference, pp. 45–54.

[23] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 31–36. [Online]. Available: https://doi.org/10.1145/2642937.2642951

[24] R. Neill, A. Drebes, and A. Pop, "Fuse: Accurate multiplexing of hardware performance counters across executions," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, pp. 1–26, 2017.

[25] NXP, "P-Series in QorIQ Processing Platforms," accessed on 07.01.2020.

[26] ——, "T-Series in QorIQ Processing Platforms," accessed on 07.01.2020.

[27] J. Freitag, S. Uhrig, and T. Ungerer, "Virtual Timing Isolation for Mixed-Criticality Systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 13:1–13:23. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/8990

[28] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch, "Contention-aware dynamic memory bandwidth isolation with predictability in COTS multicores: An avionics case study," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[29] NXP, "P4080 Multicore Communication Processor Reference Manual," September 2015, accessed on 07.01.2020.

[30] ——, "QorIQ T2080 Reference Manual," Novemebr 2016, accessed on 07.01.2020.

[31] Xilinx, "ZCU102 User Guide," November 2016, accessed on 07.01.2020.

[32] S. Roozkhosh and R. Mancuso, "The potential of programmable logic in the middle: cache bleaching," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 296–309.

[33] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing mixed criticality applications on modern heterogeneous MPSoC platforms," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[34] Intel, "Resource Director Technology Refrence Manual," March 2019, accessed on 07.01.2020.

[35] K. T. Nguyen, "Introduction to Memory Bandwidth Monitoring in the Intel® Xeon® Processor," February 2016, accessed on 07.01.2020.

[36] NXP, "S32V234 Reference Manual," Jnauary 2020, accessed on 07.01.2020.

[37] Xilinx, "AXI4 Refrence Guide," July 2017, accessed on 07.01.2020.

[38] ARM, "ARM® CoreLink™ QoS-301 Network Interconnect Advanced Quality of Service," 2011, accessed on 07.01.2020.

[39] ——, "ARM® CoreLink™ QoS-400 Network Interconnect Advanced Quality of Service," 2013, accessed on 07.01.2020.

[40] P. T. Jean-Yves Bpusec, "Network Calculus: A Theory of Deterministic Queuing Systems for the Internel," December 2019, accessed on 07.01.2020.

[41] S. Altmeyer and C. M. Burguière, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707–719, 2011.

[42] S. Altmeyer, C. Maiza, and J. Reineke, "Resilience analysis: tightening the CRPD bound for set-associative caches," *ACM Sigplan Notices*, vol. 45, no. 4, pp. 153–162, 2010.

[43] G. Nelissen, J. Fonseca, G. Raravi, and V. Nélis, "Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks," in *2015 27th Euromicro Conference on Real-Time Systems*, 2015, pp. 80–89.

[44] Vivante, "Vega Cores for 3D," accessed on 07.01.2020. [Online]. Available: http://www.vivantecorp.com/index.php/en/technology/3d.html

[45] ARM, "AMBA Network Interconncet(NIC-301) Technical Reference Manual," 2010, accessed on 07.01.2020.

[46] J. Kiszka, V. Sinitsin, H. Schild, and contributors, "Jailhouse Hypervisor," accessed on 07.01.2020. [Online]. Available: ttps://github.com/siemens/jailhouse

[47] M. S. T. Kloda, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems," in *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2019)*, Montreal, Canada, April 2019, conference, pp. 1–14.

[48] C. Scirdino, L. Cuomoand, M. Solieri, and M. Sojka, "HERCULES: High-Performance Real-Time Architectures for Low-Power Embedded Systems," December 2018, accessed on 07.01.2020.

[49] Arm, "Arm Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring(MPAM), for Armv8-A," 2018-2020, accessed on 10.16.2020.

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

[51] M. Hassan, "Reduced latency DRAM for multi-core safety-critical real-time systems," *Real-Time Systems*, pp. 1–36, 2019.

[52] C. Dall, S.-W. Li, J. T. Lim, J. Nieh, and G. Koloventzos, "ARM virtualization: performance and architectural implications," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 304–316.