SQUASH: Simple QoS-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators

Hiroyuki Usui
husui@andrew.cmu.edu

Lavanya Subramanian
lsubrama@andrew.cmu.edu

Kevin Chang
kevincha@andrew.cmu.edu

Onur Mutlu
onur@cmu.edu

Carnegie Mellon University

Abstract

Modern SoCs integrate multiple CPU cores and Hardware Accelerators (HWAs) that share the same main memory system, causing interference among memory requests from different agents. The result of this interference, if not controlled well, is missed deadlines for HWAs and low CPU performance. State-of-the-art mechanisms designed for CPU-GPU systems strive to meet a target frame rate for GPUs by prioritizing the GPU close to the time when it has to complete a frame. We observe two major problems when such an approach is adapted to a heterogeneous CPU-HWA system. First, HWAs miss deadlines because they are prioritized *only when* they are too close to their deadlines. Second, such an approach does *not* consider the diverse memory access characteristics of different applications running on CPUs and HWAs, leading to low performance for latency-sensitive CPU applications and deadline misses for some HWAs, including GPUs.

In this paper, we propose a Simple QUality of service Aware memory Scheduler for Heterogeneous systems (SQUASH), that overcomes these problems using three key ideas, with the goal of meeting HWAs' deadlines while providing high CPU performance. First, SQUASH prioritizes a HWA when it is not on track to meet its deadline *any time* during a deadline period, instead of prioritizing it only when close to a deadline. Second, SQUASH prioritizes HWAs over memory-intensive CPU applications based on the observation that memory-intensive applications' performance is not sensitive to memory latency. Third, SQUASH treats short-deadline HWAs differently as they are more likely to miss their deadlines and schedules their requests based on worst-case memory access time estimates.

Extensive evaluations across a wide variety of different workloads and systems show that SQUASH achieves significantly better CPU performance than the best previous scheduler while always meeting the deadlines for all HWAs, including GPUs, thereby largely improving frame rates.

1 Introduction

Today's SoCs are heterogeneous architectures that integrate hardware accelerators (HWAs) and CPUs. Special-purpose hardware accelerators are widely used in SoCs, along with general-purpose CPU cores, because of their ability to perform specific operations in a fast and energy-efficient manner. For example, CPU cores and Graphics Processing Units (GPUs) are often integrated in smart phone SoCs [35]. Hard-wired HWAs are implemented in a very wide range of SoCs [45], including smart phones.

In most such SoCs, HWAs share the main memory with CPU cores. Main memory is a heavily contended resource between multiple agents and a critical bottleneck in such systems [35, 45]. This becomes even more of a problem since HWAs need to meet *deadlines*. Therefore, it is important to manage the main memory such that HWAs meet deadlines while CPUs achieve high performance.

Several previous works have explored application-aware memory request scheduling in CPU-only multicore systems [30, 32, 31, 21, 22, 43]. The basic idea is to reorder requests from different CPU cores to achieve high performance and fairness. However, there have been few previous works that have tackled the problem of main memory

management in heterogeneous systems consisting of CPUs and HWAs, with the dual goals of 1) meeting HWAs' deadlines while 2) achieving high CPU performance.

The closest works tackle the problem of memory management in specifically CPU-GPU systems (e.g., [17, 7]). In particular, the state-of-the-art memory scheduling scheme for CPU-GPU systems [17] strives to meet a target frame rate for the GPU while achieving high CPU performance. Its key idea is to prioritize the GPU over the CPU cores *only close to the deadline* when the GPU has to finish a frame. The GPU is either deprioritized or given the same priority as the CPU cores at other times. This scheme does not consider different HWAs than GPUs.

We adapted this state-of-the-art scheme [17] to a heterogeneous system with CPUs and various HWAs, and observed that such an approach when used in a CPU-HWA context, suffers from two major problems. First, it prioritizes a HWA *only when* it is close to its deadlines, thus causing the HWA to potentially miss deadlines. Second, it is *not aware* of the memory access characteristics of the different applications executing on different agents (CPUs or HWAs), thus resulting in *both* HWA deadline misses and low CPU performance.

Our goal, in this work, is to design a memory scheduler that 1) meets HWAs' deadlines and 2) at the same time maximizes CPU performance. To do so, we design a scheduler that takes into account the differences in memory access characteristics and demands of *both* different CPU cores and HWAs. Our *Simple QUality of service Aware memory Scheduler for Heterogeneous systems (SQUASH)* is based on three key ideas.

First, to tackle the problem of HWAs missing their deadlines, SQUASH prioritizes a HWA *any time when it is not on track to meet its deadline* (called *Distributed Priority*), instead of prioritizing it *only* when close to a deadline . Effectively, our mechanism *distributes* the priority of the HWA over its deadline period, instead of clumping it towards the end of the period. This allows each HWA to receive *consistent memory bandwidth* throughout its run time. Second, SQUASH exploits the heterogeneous memory access characteristics of different *CPU* applications and prioritizes HWAs over *memory-intensive* CPU applications *even when* HWAs are on track to meet their deadlines. The reason is that memory-intensive CPU applications' performance is not greatly sensitive to additional memory latency. Hence, prioritizing HWAs over memory-intensive CPU applications enables faster progress for HWAs and reduces the amount of time they are *not* on track to meet their deadlines. This, in turn, reduces the amount of time HWAs are prioritized over *memory-non-intensive* CPU applications that are latency-sensitive, thereby achieving high overall CPU performance. Third, SQUASH exploits the heterogeneous access characteristics of different *HWAs*. We observe that a HWA with a *short deadline period* needs a short burst of high priority *long enough* to ensure its few requests are served, rather than consistent memory bandwidth. SQUASH achieves this by prioritizing such HWAs for a short time period based on their estimated worst-case memory access latency.

This paper makes the following main contributions.

- We identify a new problem: state-of-the-art memory schedulers cannot both satisfy HWAs' QoS requirements and provide high CPU performance.

- We propose SQUASH, a new QoS-aware memory scheduler that *always* meets HWAs' deadlines while greatly improving CPU performance over the best previous scheduler.

- We compare SQUASH to four different memory schedulers across a wide variety of system configurations and workloads. We show that SQUASH improves CPU performance by 10.1% compared to the best-previous scheduler, while always meeting deadlines of all HWAs including GPUs.

2 Background

In this section, we will first provide an overview of heterogeneous SoC architectures and hardware accelerators (HWAs) that are significant components in heterogeneous SoCs. Next, we will provide a brief background on the organization of the DRAM main memory and then describe the closest previous works on main memory management and interference mitigation in heterogeneous SoCs.

2.1 Heterogeneous SoC Architecture

Modern SoCs are heterogeneous architectures that integrate various kinds of processors. Figure 1 is an example of a typical high-end SoC designed for smart phones [35, 20]. The CPU is used to perform general purpose computation. There are multiple kinds of peripheral units such as video-I/O, USB, WLAN controller, and modem. HWAs are
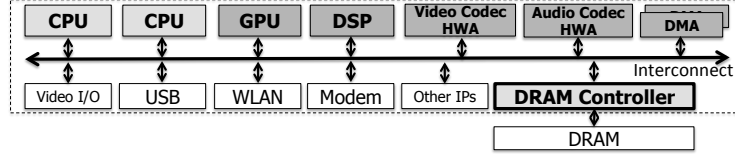
Figure 1: Example heterogeneous SoC architecture

employed to accelerate various functions. For instance, the GPU and Digital Signal Processor (DSP) are optimized for graphics and digital signal processing respectively. Other hard-wired HWAs are employed to perform video and audio coding at low power consumption. Image recognition is another common function for which HWAs are used [45, 41] because image recognition requires a large amount of computation that embedded CPUs cannot perform in a timely fashion [45]. In such a heterogeneous SoC, the DRAM main memory is a critical bottleneck between the CPU cores, HWAs, and DMA engines. Satisfying the memory bandwidth requirements of all these requestors (or, agents) becomes a major challenge. In this work, we will focus on managing memory bandwidth between the CPU cores and HWAs with the goal of meeting deadline requirements for HWAs while improving CPU performance.

2.2 Hardware Accelerator Characteristics

Modern SoCs consist of a wide variety of HWAs as each one is designed to accelerate a specific function. The functions that they accelerate are diverse and the implementations also vary among different HWAs. As an example, we will first describe a typical implementation of a *3x3 horizontal Sobel filter* accelerator [40] (shown in Figure 2), which computes the gradient of an image for image recognition.
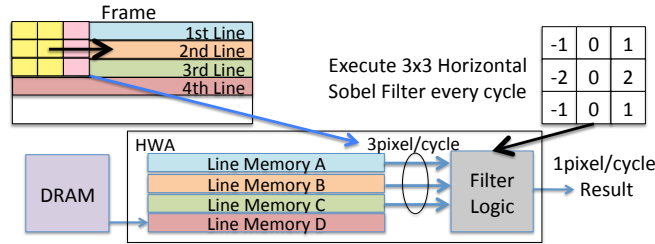

Figure 2: Typical implementation of a Sobel filter HWA

The accelerator executes the Sobel filter on a target VGA (640x480) image, at a target frame rate of 30 frames per second (fps). A typical implementation for the filter uses *line memory* to take advantage of data access locality and hide the memory access latency, as shown in Figure 2. The line memory (consisting of lines A, B, C and D) can hold four lines, each of size 640 pixels, of the target image. The filter operates on three lines, at any point in time, while the next line is being prefetched. For instance, the filter operates on lines A, B, and C, while line D is being prefetched. After the filter finishes processing lines A, B, and C, it operates on lines B, C, and D, while line A is being prefetched. As long as the next line is prefetched while the filter is operating on the three previous lines, memory access latency does not affect performance. To meet a performance target (30 fps), the filtering operation on a set of lines and the fetching of the next line have to be finished within 69.44 μsec ($= 1\ sec/30\ frames/480\ lines$). In this case, the *period* of the HWA is 69.44 μsec and the next line needs to be prefetched by the end of the period (the *deadline*). Missing this deadline causes the filtering logic to stall and drop the frame being processed. As a result, it prevents the system from achieving the performance target.

On the other hand, if the next-line prefetch is finished earlier than the deadline, the prefetch of the line after that cannot be initiated because the line memory can hold only one extra prefetched line. Prefetching more than one line can overwrite the contents of the line that is currently being processed by the filter logic. In order to provision for more capacity to hold prefetched data and hide memory latency better, double buffers (e.g., frame buffers used in GPUs) are implemented in some HWAs.

There are several HWAs with similar architectures employing line/frame buffers, which are widely used in the media processing domain. HWAs for resizing an image [11] or feature extraction [13, 4] use line buffers. HWAs for

acoustic feature extraction [38] use frame buffers. In all these HWAs, computing engines can only access line/frame buffers and data is prefetched into these buffers from main memory.

Regardless of the types of buffers being used, one common attribute across all these HWAs is that the amount of available buffer capacity determines the *deadline*, or *period*, and how much data needs to be sent for each *period*. For instance, in the Sobel filter HWA example described above, the HWA requires continuous bandwidth during each period (640 bytes for every 69.44 *µ*sec). As long as this continuous bandwidth is allocated, the HWA is tolerant of memory latency. On the other hand, finishing the HWA's memory accesses earlier than the deadline is wasteful, especially in a system with other agents such as CPU cores, where the memory bandwidth can be better utilized to achieve higher overall performance for the other agents.

As a result, a major challenge in today's heterogeneous SoCs is to ensure that the HWAs get a consistent share of main memory bandwidth such that their deadlines are met, while allocating enough bandwidth to the CPU cores to achieve high CPU performance. This challenge is not solved by today's memory schedulers which focus on either the HWAs or the CPUs. As we will show in our evaluation (Section 7), the HWA-friendly memory scheduler that achieves almost 100% deadline-met ratio for HWAs has 12% lower performance compared to the CPU-friendly scheduler that attains the highest CPU performance without always meeting the deadlines. **The goal of our work** is *to both meet the HWAs' deadlines and maximize CPU performance*.

2.3 DRAM Main Memory Organization

A typical DRAM main memory system is organized as a hierarchy of channels, ranks, and banks. Each channel has its own address and data bus that operate independently. A channel consists of one or more ranks. A rank, in turn, consists of multiple banks. Each bank can operate independently and in parallel with the other banks. However, all the banks on a channel share the address and data buses of the channel.

Each bank consists of a 2D array (rows and columns) of cells. When a piece of data is accessed from a bank, the entire row containing the piece of data is brought into a bank-internal structure called the *row buffer*. Any subsequent access to other data in the same row can be served from the row buffer itself without incurring the additional latency of accessing the array. Such an access is called a *row hit*. On the other hand, if the subsequent access is to data in a different row, the array needs to be accessed and the new row needs to be brought into the row buffer. Such an access is called a *row miss*. A *row miss* incurs more than 2x the latency of a row hit [31, 37].

2.4 Memory Management in Heterogeneous Systems

Many previous works have tackled the problem of managing memory bandwidth between applications in CPU-only multicore systems [37, 30, 31, 32, 21, 22, 44]. However, few previous works have tackled the problem of memory management in heterogeneous systems, taking into account the memory access characteristics of the different agents. One previous work [42] attempts to satisfy both high system performance and QoS by acknowledging the differences in memory access characteristics between CPU cores and other agents. They observe that CPU cores are latency sensitive, whereas the GPU and Video Processing Unit (VPU) are bandwidth sensitive with high memory latency tolerance. Therefore, they propose to prioritize CPU requests over GPU requests, while attempting to provide sufficient bandwidth to the GPU. However, with such a scheme, the GPU cannot always achieve its performance target when the CPU demands high bandwidth [17].

In order to address this challenge of managing memory bandwidth between the CPU cores and the GPU, a state-of-the-art technique [17] proposed to dynamically adjust memory access priorities between the CPU cores and the GPU. This policy compares current progress in terms of tiles rendered in a frame (Equation 1) against expected progress in terms of time elapsed in a period (Equation 2) and adjusts priorities.

$$CurrentProgress = \frac{Number\ of\ tiles\ rendered}{Number\ of\ tiles\ in\ the\ frame} \tag{1}$$

$$ExpectedProgress = \frac{Time\ elapsed\ in\ the\ current\ frame}{Period\ for\ each\ frame} \tag{2}$$

When *CurrentProgress* is greater than *ExpectedProgress*, the GPU is on track to meet its target frame rate. Hence, GPU requests are given lower priority than CPU requests. On the other hand, if *CurrentProgress* is less than or equal to *ExpectedProgress*, the GPU is not on track to meet its target frame rate. In order to enable the GPU to make better progress, its requests are given the *same* priority as the CPU cores' requests. Only when the *ExpectedProgress* is

greater than an *EmergentThreshold* (=0.90), are the GPU's requests given higher priority than the CPU cores' requests. Such a policy aims to preserve CPU performance, while still giving the GPU highest priority close to the time when a frame needs to be completed, thereby providing better QoS to the GPU than static prioritization policies. However, this policy, when used within the context of a CPU-HWA system, is not adaptive enough, as we will show next, to the diverse characteristics of different CPU applications and HWAs.

3 Motivation and Key Ideas

In this work, we examine heterogeneous systems that consist of multiple HWAs and CPU cores executing applications with very diverse characteristics (e.g., memory intensity and deadline requirements). Although we have a different usage scenario from the previous work that targets CPU-GPU systems by Jeong et al. [17] (discussed in the previous section), we adapt their scheduling policy in [17] and apply it to manage memory bandwidth between CPU cores and HWAs since it is the best previous work we know of in memory scheduling for heterogeneous systems. We call this policy *Dyn-Prio*.

Similar to GPUs' frame rate requirements, HWAs need to meet deadlines. We target HWAs having soft deadlines, such as HWAs for image processing and image recognition. A deadline miss for such HWAs causes frames to be dropped. We re-define *CurrentProgress* and *ExpectedProgress* in Equations 3 and 4, respectively to capture HWAs' deadline requirements.

$$CurrentProgress = \frac{\# \, of \, completed \, memory \, requests/period}{\# \, of \, total \, memory \, requests/period} \tag{3}$$

$$ExpectedProgress = \frac{Time \, elapsed \, in \, current \, period}{Total \, length \, of \, current \, period} \tag{4}$$

CurrentProgress for HWAs is defined as the fraction of the total number of memory requests that have been completed. *ExpectedProgress* for HWAs is defined in terms of the fraction of time elapsed during an execution period. In order to compute *CurrentProgress*, the number of requests served during each period is needed. For several kinds of HWAs, it is possible to precisely know this number due to two reasons. First, as described in Section 2.2, a lot of HWAs for media processing access media data in a streaming manner, resulting in a predictable/prefetch-friendly access stream. Second, when a HWA is implemented with a line-/double-buffered scratchpad, all the data required for the next set of computations need to be prefetched into the scratchpad to meet a target performance because the compute engines can only access the scratchpad. In this scenario, the number of memory requests in a period can be estimated in a fairly straightforward manner based on the amount of data that are required to be prefetched.

We observe that there are two major problems with Dyn-Prio when used in a CPU-HWA context. First, it only prioritizes a HWA over CPU cores close to the HWA's deadline (i.e., after 90% *ExpectedProgress* has been made). Prioritizing HWAs only when the deadline is very close can cause deadline misses because the available memory bandwidth in the remaining time before the deadline may not be able to sustain the required memory request rates of all the HWAs and CPUs. We will explain this problem in greater detail in Section 3.1. Second, Dyn-Prio is designed for a simple heterogeneous CPU-GPU system and is not designed to consider the access characteristics of different applications in a heterogeneous system executing different kinds of applications on different kinds of agents, either on the CPUs or on the HWAs. As we will explain in Section 3.2 and 3.3, application-unawareness misses opportunities to improve system performance because different applications executing on CPUs and HWAs have different latency tolerance and bandwidth requirements.

3.1 Key Idea 1: Distributed Priority

To address the first problem where HWAs sometimes miss their deadlines, we propose a simple modification. A HWA enters a state of urgency and is given the highest priority anytime when its *CurrentProgress* is less than or equal to *ExpectedProgress*. We call such a scheme *Distributed Priority* (*Dist-Prio* for short). Using *Dist-Prio* distributes a HWA's priority over its deadline period, rather than clumping it close to a deadline. This allows HWAs to receive consistent memory bandwidth and make *steady progress* throughout their run time.

To illustrate the benefit of such a policy, we study an example system with two CPU cores and a HWA. Figure 3 shows the execution timelines when each agent (HWA or CPU core) executes alone. In this example, CPU-A has low memory intensity and generates very few memory requests. In contrast, CPU-B has high memory intensity and
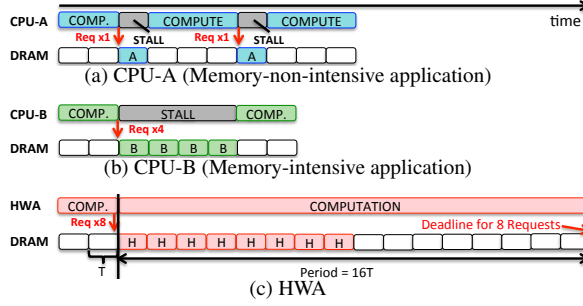
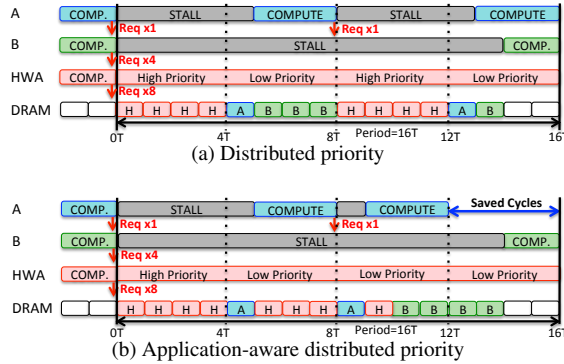Figure 3: Memory service timeline example when each agent is executed alone

Figure 4: Memory service timeline example when all agents execute together

generates more memory requests than `CPU-A` does. `HWA` has double buffers and generates 8 prefetch requests for the data to be processed in the next period. For ease of understanding, we assume all these requests are destined to the same bank and each memory request takes T cycles (no distinction between row hits and misses). The length of the `HWA`'s period is 16T. When a *Dyn-Prio* scheme with an *EmergentThreshold* of 0.9 is employed to schedule these requests, the `HWA` is given highest priority only for the last two time units, starting at time 14T. Until then, the CPU cores' requests are treated on par with the `HWA`'s requests. Such a short amount of time is not sufficient to finish serving the `HWA`'s requests in this example.

Figure 4a illustrates the scheduling order of requests from a *single system* with a HWA (`HWA`) and two CPU cores (`CPU-A` and `CPU-B`) using our proposed *Dist-Prio* scheme. It prioritizes the HWA *any time* when it is not on track to meet its deadline. Among the CPU cores, the low memory-intensity `CPU-A` is prioritized over the high memory-intensity `CPU-B`. At the beginning of the deadline period, since both *CurrentProgress* and *ExpectedProgress* are zero and equal, `HWA` is deemed as urgent and is given higher priority than the CPU cores. Hence, during the first 4T cycles, only `HWA`'s requests are served. After 4T cycles, *CurrentProgress* is 0.5 and *ExpectedProgress* is 0.25. Hence, `HWA` is deemed as not urgent and is given lower priority than the CPU cores. Requests from both CPU cores are served from cycles 4T to 8T. After 8T cycles, since both *CurrentProgress* and *ExpectedProgress* are 0.50, `HWA` is deemed as urgent again and its remaining requests are completed. Hence, Dist-Prio enables the HWA to meet its deadlines while also achieving high CPU performance.

3.2 Key Idea 2: Application-Aware Scheduling for CPUs

We observe that when HWAs are given higher priority than CPU cores, they interfere with all CPU cores' memory requests. For instance, in Figure 4a, during cycles 8T to 12T, `HWA` stalls both `CPU-A` and `CPU-B`. Furthermore, higher the memory intensity of the HWAs, more the memory bandwidth they need to make sufficient progress to meet their deadlines, exacerbating the interference. We propose to tackle this shortcoming based on the observation that memory-intensive applications do not experience significant performance degradation when HWAs are prioritized over them.

Applications with low memory-intensity are more sensitive to memory latency, since these applications generate few memory requests, and quick service of these few requests enables such applications to make good forward

progress.[1] On the other hand, applications with high-memory-intensity often have a large number of outstanding memory requests and spend a significant fraction of their execution time stalling on memory. Therefore, delaying high-memory-intensity applications' requests does not impact their performance significantly. Based on this observation, we propose to prioritize HWAs' memory requests over those of high-memory-intensity applications even when HWAs are making sufficient progress and are *not* in a state of urgency. Such a prioritization scheme reduces the number of cycles when HWAs are deemed *urgent* and prioritized over *memory-non-intensive* CPU applications that are latency-sensitive, thereby improving their performance.

Figure 4b illustrates the benefits of such an application-aware distributed priority scheme for the same set of requests shown in Figure 4a. The request schedule remains the same during the first 4T cycles. At time 4T, the *CurrentProgress* is 0.5 and *ExpectedProgress* is 0.25. Since *CurrentProgress* is greater than *ExpectedProgress*, HWA is not deemed urgent and CPU-A's request is prioritized over HWA's requests. However, HWA is *still* prioritized over CPU-B that has high memory-intensity, enabling faster progress for HWA. As a result, at time 8T, *CurrentProgress* is 0.875, which is greater than *ExpectedProgress*. As such, the HWA is *still* deemed *not urgent*, unlike in the distributed priority case (in Figure 4a). Hence, the latency-sensitive CPU-A's requests are served earlier. Thus, this key idea enables faster progress for CPU-A, as can be seen from Figure 4b, and results in higher CPU performance.

3.3 Key Idea 3: Application-Aware Scheduling for HWAs

Monitoring a HWA's progress and prioritizing it when it is not on track to meet its deadline is an effective mechanism to ensure consistent bandwidth to HWAs that have fairly long periods (infrequent deadlines). However, such a scheme is not effective for HWAs with short periods (frequent deadlines) since it is difficult to ensure that these HWAs are deemed urgent and receive enough priority for sufficiently long periods of time within a short deadline period. Specifically, a short deadline provides little time for all requests and causes the HWAs to be more susceptible to interference from other HWAs and CPU cores. We evaluated a heterogeneous system with two accelerators (HWA-A and HWA-B) that have vastly different period lengths (i.e., 63041 and 5447 cycles) and bandwidth requirements (i.e., 8.32GB/s and 475MB/s) employing our previously two key ideas. Our results show that HWA-A meets all its deadlines whereas HWA-B, on average, misses a deadline every 2000 execution periods.

To help enable better deadline-met ratios for HWAs with short deadlines, we make the following two observations that lead to our third key idea. First, HWAs with short deadline periods can be enabled to meet their deadlines by giving them a short burst of highest priority very close to the deadline. Second, prioritizing short-deadline-period HWAs does not cause much interference to other requestors because these HWAs consume a small amount of bandwidth. Based on these observations, we propose to estimate the *WorstCaseLatency* for a memory access and give a short-deadline-period HWA highest priority for *WorstCaseLatency* ∗ *NumberOfRequests* cycles close to its deadline.

4   Mechanism

In this section, we describe the details of SQUASH, our proposed memory scheduling mechanism to manage memory bandwidth between CPU cores and HWAs, using the three key ideas described in Section 3. First, we describe a scheduling policy to prioritize between HWAs with long deadline periods and CPU applications, with the goal of enabling the long-deadline-period HWAs to meet their deadlines while improving CPU performance (Section 4.1). Second, we describe how SQUASH enables HWAs with short deadline periods to meet their deadlines (Section 4.2). Third, we present a combined scheduling policy for long and short-deadline-period HWAs (Section 4.3). Finally, we describe a modification to our original scheduling policy to probabilistically change priorities between long-deadline-period HWAs and CPU applications to enable higher fairness for memory-intensive CPU applications (Section 4.4), which results in the final SQUASH mechanism.

Overview: SQUASH categorizes HWAs as long and short-deadline-period statically based on their deadline period. A different scheduling policy is employed for each of these two categories, since they have different kinds of bandwidth demand. For the long-deadline-period accelerators (*LDP-HWAs* for short), SQUASH monitors their progress periodically and appropriately prioritizes them, enabling them to get sufficient and consistent bandwidth to meet their deadlines (Section 3.1). For the short-deadline-period accelerators (*SDP-HWAs* for short), SQUASH gives them a short burst of highest priority close to each deadline, based on worst-case access latency calculations

---

[1]This was also observed by some previous works and utilized in the context of multi-core memory scheduling. [21, 22]

(Section 3.3). SQUASH also treats memory-intensive and memory-non-intensive CPU applications differently with respect to their priority over HWAs (Section 3.2).

4.1 Long-Deadline-Period HWAs vs. CPU Applications

To make scheduling decisions between LDP-HWAs and CPU applications, SQUASH employs the *Distributed Priority (Dist-Prio)* scheme as previously described in Section 3.1, monitoring each LDP-HWA's progress every *SchedulingUnit*. SQUASH prioritizes LDP-HWAs over CPUs only when LDP-HWAs become *urgent* under either of the following conditions: 1) $CurrentProgress \leq ExpectedProgress$ or 2) $ExpectedProgress > EmergentThreshold$.

CPU applications' memory-intensity is monitored and applications are classified as memory-non-intensive or memory-intensive periodically based on their calculated memory-intensity using the classification mechanism borrowed from [22]. Note that other mechanisms can also be employed to perform this classification.[2]

Based on this classification, SQUASH schedules requests at the memory controller in the following priority order (lower number indicates higher priority):

1. Urgent HWAs

2. Memory-non-intensive CPU applications

3. Non-urgent HWAs

4. Memory-intensive CPU applications

Based on our first observation in Section 3.2, HWAs' requests are prioritized over memory-intensive CPU applications' requests, even when the HWAs are not deemed urgent since memory-intensive applications are not latency-sensitive.

4.2 Providing QoS to HWAs with Short Deadline Periods

Although using Dist-Prio can provide consistent bandwidth to LDP-HWAs to meet their deadlines, SDP-HWAs do not get enough memory bandwidth to meet their deadlines (as we described in our third key idea in Section 3.3). In order to enable SDP-HWAs to meet their deadlines, we propose to give them a short burst of high priority close to a deadline using estimated worst-case memory latency calculations.

Estimating worst case access latency. In the worst case, all requests from a SDP-HWA would access different rows in the same bank. In this case, all such requests are serialized and each request takes tRC - the minimum time between two DRAM row ACTIVATE operations.[3] Therefore, in order to serve the requests of an SDP-HWA before its deadline, it needs to be deemed urgent and given the highest priority over all other requestors for $tRC * NumberOfRequests$ for each period, which we call it *Urgent Period Length (UPL)*.

For example, when a HWA outputs 16 requests every 2000 ns period and tRC is 50 ns, the HWA is deemed urgent and given the highest priority for 800 ns + α during each period, where α is the waiting time for the in-flight requests to finish. Furthermore, finishing a HWA's requests much earlier than the deadline is wasteful, since doing so does not improve the HWA's performance any further. Hence, this highest priority period can be at the end of the HWA's deadline period. For instance, in the previous example, the HWA is given highest priority $(2000 - (800 + \alpha))$ ns after a deadline period starts.

Handling multiple short-deadline-period HWAs. The scheme discussed above does not consider the scenarios when there are multiple SDP-HWA, which could overlap with each other during the high priority cycles, resulting in deadline misses. We propose to address this concern using the following mechanism:

1. SQUASH calculates the urgent period length (UPL) of each SDP-HWA *x* as:
   $UPL(x) = tRC * TheNumberOfRequests(x)$

2. Among the urgent short-deadline-period HWAs, the HWAs with shorter deadline periods are given higher priority.

---

[2]Also note that even though we borrow the classification mechanism of [22] to categorize memory-intensive and memory-non-intensive applications, the problem we solve and the scheduling policy we devise are very different from those of [22]

[3]Accesses to different rows within the same bank have to be spaced apart by a fixed timing of tRC based on the DRAM specification [16].

3. SQUASH extends the urgent period length of each SDP-HWA x further by taking into account all the SDP-HWAs that have higher priority (i.e., shorter deadline period) than x. This ensures that each HWA is allocated enough cycles for its urgent period. When calculating how long we should extend a SDP-HWA x's *UPL*, we calculate how many deadline periods (N_i) of each higher priority SDP-HWA (i) can overlap with the *UPL* of x: $N_i = \lceil (UPL(x)/Period(i)) \rceil$. We then calculate the total length of high-priority *UPL*, *HP-UPL(i)*, resulting from N_i high-priority deadline periods: $HP\text{-}UPL(i) = N_i * UPL(i)$, which we use to add to the current SDP-HWA's UPL. In summary, the final extension function for each SDP-HWA x is: $UPL(x) = \Sigma_i(HP\text{-}UPL(i)) + UPL(x)$, for all HWAs i that have higher priority than x.

4.3 Overall Scheduling Policy

Combining the mechanisms described in Sections 4.1 and 4.2, SQUASH schedules requests in the following order (lower number indicates higher priority) and priority level within each group is also provided in parentheses:

1. **Urgent HWAs in the short deadline period group** (Higher priority to shorter deadline HWAs)

2. **Urgent HWAs in the long deadline period group** (Higher priority to earlier deadline HWAs)

3. **Non-memory-intensive CPU applications** (Higher priority to lower memory-intensity applications)

4. **Non-urgent HWAs in the long deadline period group** (Higher priority to earlier deadline HWAs)

5. **Memory-intensive CPU applications** (Application priorities are shuffled as in [22])

6. **Non-urgent HWAs in the short and long deadline period group** (Higher priority to earlier deadline HWAs)

The current scheduling order allows HWAs to receive high priority when they becomes urgent (i.e., not meeting their expected progress). This prevents them from missing deadlines due to interference from CPU applications. Memory-intensive CPU applications (*Group*5) are always ranked lower than memory-non-intensive CPU applications (*Group*3) and LDP-HWAs (*Group*2, 4). This can potentially always deprioritize memory-intensive applications when the memory bandwidth is only enough to serve memory-non-intensive applications and HWAs. To ensure memory-intensive applications receive sufficient memory bandwidth to make progress, we employ a clustering mechanism that only allocates a fraction of total memory bandwidth (called *ClusterFactor*) to the memory-non-intensive group [22].

As explained in Section 3.1, the initial state of all HWAs is urgent when using *Dist-Prio*. When HWAs meet their expected progress, they make the transition to the non-urgent state, allocating more bandwidth to CPU applications or other HWAs. Non-urgent SDP-HWAs are always in *Group*6. Non-urgent LDP-HWAs, however, can be in either *Group*4 or *Group*6. They are assigned to *Group*6 only when they first transition to the non-urgent state, but are assigned to *Group*4 when they re-enter the non-urgent state later on. The rationale is that LDP-HWAs do not need to be prioritized over memory-intensive CPU applications (*Group*5) if they are already receiving memory bandwidth such that they continuously meet their expected progress, without ever transitioning back to the urgent state again, throughout the period. This kind of a priority scheme enables LDP-HWAs to make progress while not over-consuming memory bandwidth and enables memory-intensive CPU applications to achieve higher performance.

4.4 Probabilistic Switching of LDP-HWAs' Priorities

By using the scheduling order described in the previous section, we observe that memory-intensive applications experience unfair slowdowns due to interference from non-urgent LDP-HWAs in some workloads. To solve this problem, we propose a mechanism to probabilistically prioritize memory-intensive applications over non-urgent LDP-HWAs, switching priorities between *Group*4 and *Group*5. Each LDP-HWA x has a probability value $Pb(x)$ that is controlled based on its request progress every epoch (*SwitchingUnit*). Algorithm 1 shows how requests are scheduled based on $Pb(x)$. With a probability of $Pb(x)$, memory-intensive applications are prioritized over LDP-HWA x to enable higher fairness. Algorithm 2 shows the periodic adjustment of $Pb(x)$ using empirically determined steps. We use a larger decrement step than the increment step because we want to quickly reduce the priority of memory-intensive applications in order to increase the HWA's bandwidth allocation when it is not making sufficient progress. This probabilistic switching helps ensure that the memory-intensive CPU applications are treated fairly.

Algorithm 1 Scheduling using $Pb(x)$

With a probability $Pb(x)$:
Memory-intensive applications $>$ Long-deadline-period HWA x
With a probability $(1 - Pb(x))$:
Memory-intensive applications $<$ Long-deadline-period HWA x

Algorithm 2 Controlling $Pb(x)$ for LDP-HWAs

Initialization: $Pb(x) = 0$
Every *SwitchingUnit:*
if *CurrentProgress ¿ ExpectedProgress* **then**
    $Pb(x) += Pb_{inc}$ ($Pb_{inc} = 1\%$ in our experiments)
else if *CurrentProgress ¡ ExpectedProgress* **then**
    $Pb(x) -= Pb_{dec}$ ($Pb_{dec} = 5\%$ in our experiments)
else
    $Pb(x) = Pb(x)$
end if

---

5   Implementation and Hardware Cost

SQUASH requires hardware support to monitor HWAs' current and expected progress and schedule memory requests accordingly. To track current progress, the memory controller counts the number of completed requests during a deadline period. If there are multiple memory controllers, they send their recorded counter values to a centralized meta-controller every *SchedulingUnit*, similar to [21, 22]. If HWAs access shared caches, the number of completed requests at the shared caches is sent to the meta-controller. Table 1 lists the major counters required for the meta-controller over a baseline TCM scheduler [22], the state-of-the-art application-aware scheduler for multi-core systems, which we later provide comparison to. The request counters are used to track current progress, whereas the cycle counters are used to compute expected progress. *Pb* is the probability that determines priorities between long-deadline-period HWAs and memory-intensive applications. A 4-byte counter is sufficient to denote each of these quantities. Hence, the total counter overhead is 20 bytes for a long-deadline-period HWA and 12 bytes for a short-deadline-period HWA.

For long-deadline-period HWAs	
Name	**Function**
Curr-Req	Number of requests completed in a deadline period
Total-Req	Total number of requests completed in a deadline period
Curr-Cyc	Number of cycles elapsed in a deadline period
Total-Cyc	Total number of cycles in a deadline period
Pb	Probability when memory-intensive applications
	$>$ long-deadline-period HWAs
For short-deadline-period HWAs	
Name	**Function**
Priority-Cyc	Indicates when the priority is transitioned to high
Curr-Cyc	Number of cycles elapsed in a deadline period
Total-Cyc	Total number of cycles in a deadline period

Table 1: Storage required for SQUASH

Total-Req and *Total-Cyc* are set by the system software based on the specifications of HWAs. If these parameters are fixed for the target HWA, the software sets up these registers at the beginning of execution. If these parameters vary for each period, the software updates them at the beginning of each period. *Curr-Cyc* is incremented every cycle. *Curr-Req* is incremented every time a request is completed (at the respective memory controller). At the end of every *SchedulingUnit*, the meta controller computes *ExpectedProgress* and *CurrentProgress* using these accumulated counts, in order to determine how urgent each long-deadline-period HWA is. For the short-deadline-period HWAs, their state of urgency is determined based on *Priority-Cyc* and *Curr-Cyc*. *Priority-Cyc* is set by the system software based on the HWAs' specifications. This information is used along with *Pb* to determine the scheduling order across all HWAs and CPU applications. Once this priority order is determined, the meta-controller broadcasts the priority to the memory controllers, and the memory controllers schedule requests based on this priority order, similar to other application-aware memory schedulers [30, 31, 21, 22].

6   Methodology

6.1   System Configuration

We use an in-house cycle-level simulator to perform our evaluations. We model a system with eight x86 CPU cores and four HWAs for our main evaluation. To avoid starving CPU cores or HWAs, we allocate half of the memory request buffer entries that hold memory requests to CPU cores and the other half to HWAs. Unless stated otherwise, our system configuration is as shown in Table 2.

CPU	8 Cores, 2.66GHz, 3-wide issues
	128 entry instruction window, 16 MSHRs/core
L1Cache	Private, 2 way, 32 KB, 64 Byte Line
L2Cache	Shared, 16 way, 4 MB, 64 Byte Line
HWA	4 HWAs
DRAM	DDR3-1333 (9-9-9) [28], 300 request buffer entries
	2 channels, 1 rank per channel, 8 banks per rank

Table 2: Configuration of the simulated system

6.2   Workloads for CPUs

We construct 80 multiprogrammed workloads from the SPEC CPU2006 suite [2], TPC [3], and the NAS parallel benchmark suite [1]. We use Pin [26] with PinPoints [34] to extract representative phases. We classify CPU benchmarks into two categories, memory-intensive and memory-non-intensive, based on the number of last-level cache misses per thousand instructions (MPKI). If an application's MPKI is greater than 5, it is classified as a memory-intensive application. Otherwise, it is classified as memory-non-intensive. We then construct five intensity categories of workloads based on the fraction of memory-intensive benchmarks in a workload: 0%, 25%, 50%, 75%, and 100%. Each category consists of 16 workloads.

6.3   Hardware Accelerators

We use five kinds of HWAs designed for image processing and recognition, for our evaluations, as described in Table 3. The target frame rate for the HWAs is 30 fps. The image processing HWA (IMG-HWA) performs filter processing on input RGB images of size 1920x1080. We assume that IMG-HWA performs filter processing on one frame for 1/30 sec with double buffers. Hessian HWA (HES-HWA) and Matching HWA (MAT-HWA) are designed for Augmented Reality (AR) systems [24]. HES-HWA accelerates the fast Hessian detector that is executed in SURF (Speed up Robust Features) [8], which is used to detect interesting points and generate descriptors. MAT-HWA accelerates the operation of matching descriptors generated by SURF against those in a database. The implementation of HES-HWA and MAT-HWA are based on [24]. Their configuration parameters are as shown in Table 3. We evaluate HES-HWA and MAT-HWA for three different configurations. The periods and bandwidth requirements of the HWAs are different depending on the configuration. We assume that the result of the MAT-HWA is output to a register in the HWA. Resize HWA (RSZ-HWA) and Detect HWA (DET-HWA) are used for face detection. Their implementations are based on a library that uses Haar-Like features [47], included in Open-CV [15]. RSZ-HWA shrinks the target frame recursively in order to detect differences in sizes of faces and generates integral images. DET-HWA detects faces included in the resized image. Because the target image is shrunk recursively over each frame, the HWAs' periods are variable. The HES-HWA and DET-HWA are categorized into the short-deadline-period group and the others into the long-deadline-period group.

Based on the implementations of the HWAs, we build trace-generators that simulate memory requests from the HWAs. All HWAs have fixed access patterns throughout the simulation run. We evaluate two mixes of HWAs, Config-A and Config-B, with each CPU workload, as shown in Table 3. Config-B includes HWAs that dynamically change their bandwidth requirements and deadlines over time, which we use to evaluate the adaptivity of different schedulers. We simulate for 200 million CPU cycles. The size of memory requests from HWAs is 64 bytes and the number of outstanding requests from each HWA to the memory is at most 16.

6.4   System with a GPU

In addition to our CPU-HWA evaluations, we also evaluate CPU-GPU and CPU-GPU-HWA systems. The specification of the GPU we model is 800 MHz, 20 cores and 1600 operations/cycle, which is similar to the AMD Radeon 5870

	Period	Bandwidth	Scratchpad
IMG-HWA	33 ms	360 MB/s	double buffer (1 frame x 4)
HES-HWA(32)	2 us	478 MB/s	line buffer (32 lines)
HES-HWA(64)	4 us	329 MB/s	30 lines for computation
HES-HWA(128)	8 us	224 MB/s	2 lines for prefetch
MAT-HWA(30)	23.6 us	8.32 GB/s	double buffer (4 KB x 4)
MAT-HWA(20)	35.4 us	5.55 GB/s	4 KB x 2 for query
MAT-HWA(10)	47.2 us	2.77 GB/s	4 KB x 2 for database
RSZ-HWA	46.5 us - 5183 us	2.07 GB/s - 3.33 GB/s	double buffer (1 frame x 4)
DET-HWA	0.8 us - 9.6 us	1.60 GB/s - 1.86 GB/s	line buffer (26 lines) 24 lines for computation
	Parameters		
HES-HWA(N) [24]	image size: 1920 x 1080, max filter size: 30, N entries operated at the same time		
MAT-HWA(M) [24]	3000 interesting points (64 dimension) per image, matching M images		
RSZ-HWA DET-HWA [15]	image size: 1920 x 1080, scale factor : 1.1, 24 x 24 window		
	Configuration		
Config-A	IMG-HWA x 2, MAT-HWA(30), HES-HWA(32)		
Config-B	MAT-HWA(20), HES-HWA(32), RSZ-HWA, DET-HWA		

Table 3: Configuration of the HWAs

specification [5]. The GPU does not share caches with CPUs. The CPU-GPU-HWA system has four memory channels and four HWAs, whose configuration is the same as Config-A in Section 6.3. The other system parameters are the same as the CPU-HWA system. We collect GPU traces from GPU benchmarks and games, as shown in Table 4, with a proprietary GPU simulator. The target frame rate of all GPU benchmarks is 30 fps. We set the GPU benchmarks' deadline to 33.3 msec (= 1 frame). We measure the number of memory requests included in each trace in advance and use this number to calculate *CurrentProgress*. We simulate 30 CPU-GPU and CPU-GPU-HWA workloads respectively.

Name	Description	Name	Description
Bench	3D mark	Game03	Shooting Game 3
Game01	Shooting Game 1	Game04	Adventure Game
Game02	Shooting Game 2	Game05	Role-playing Game

Table 4: GPU benchmarks

6.5   Performance Metrics

We measure CPU performance with the commonly-used *Weighted Speedup (WS)* [10, 39] metric. We measure fairness using the *Maximum Slowdown* metric [46, 21, 22]. For HWAs, we use two metrics: the *DeadlineMetRatio* and *frame rate* in *fps*, frames per second. We assume that if a deadline is missed in a frame, the corresponding frame is dropped (and we calculate frame rate accordingly).

6.6   Parameters of the Evaluated Schedulers

Unless otherwise stated, for SQUASH, we set the * SchedulingUnit* to 1000 CPU cycles and *SwitchingUnit* to 500 CPU cycles. For TCM [22], we use a *ClusterFactor* of 0.2 and a shuffling interval of 800 cycles and *QuantumLength* of 1M cycles.

7   Evaluation

We compare SQUASH with previously proposed schedulers: 1) FRFCFS [37] and TCM with static priority (FRFCFS-St and TCM-St) where the HWA always has higher priority than all CPU cores and 2) FRFCFS with *Dyn-Prio* (FRFCFS-Dyn), which employs the dynamic priority mechanism [17]. We evaluate two variants of the FRFCFS-Dyn mechanism with different *EmergentThreshold* values. First, we use an *EmergentThreshold* value of 0.9 for all

HWAs (FRFCFS-Dyn0.9), which is designed to achieve high CPU performance. Second, in order to achieve high deadline-met ratios for the HWAs, we sweep the value of the *EmergentThreshold* from 0 to 1.0 at the granularity of 0.1 (see Section 7.3 for more details) and choose a different threshold value shown in Table 5 for each HWA (FRFCFS-DynOpt) such that a deadline-met ratio greater than 99.9% and a frame rate greater than 27 fps (90% of target frame rate) are achieved. For SQUASH, we use an *EmergentThreshold* value of 0.8 for all HWAs.

Config-A			Config-B			
HES	MAT	IMG	HES	MAT	DET	RSZ
0.2	0.2	0.9	0.5	0.4	0.5	0.7

Table 5: EmergentThreshold for FRFCFS-Dyn

Figure 5 shows the average system performance across all 80 workloads, using both Config-A and B. Table 6 shows the deadline-met ratio and frame rate of four types of HWAs. We do not show IMG-HWA because it has a 100% deadline-met ratio with all schedulers.
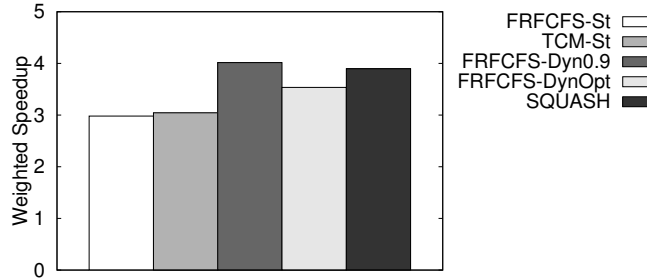

Figure 5: System performance

Scheduling Algorithms	Deadline-Met Ratio (%) / Frame Rate (fps)			
	HES	MAT	RSZ	DET
FRFCFS-St	100 / 30	100 / 30	100 / 30	100 / 30
TCM-St	100 / 30	100 / 30	100 / 30	100 / 30
FRFCFS-Dyn0.9	99.40 / 15.38	46.01 / 15.28	97.98 / 25.19	97.14 / 16.5
FRFCFS-DynOpt	100 / 30	99.997 / 29.72	100 / 30	99.99 / 25.5
SQUASH	100 / 30	100 / 30	100 / 30	100 / 30

Table 6: Deadline-met ratio and frame rate of HWAs

We draw three major observations. First, FRFCFS-St and TCM-St always prioritize HWAs, achieving a 100% deadline-met ratio. However, always prioritizing the HWAs' requests results in low CPU performance. Second, the FRFCFS-Dyn policy either achieves high CPU performance or high deadline-met ratio depending on the value of the *EmergentThreshold*. When *EmergentThreshold* is 0.9, the HWAs are not prioritized much, causing them to miss deadlines. However, CPU performance is high. On the other hand, when we use optimized values of *EmergentThreshold* (FRFCFS-DynOpt), the HWAs are prioritized enabling them to meet almost all their deadlines, but at the cost of CPU performance. Third, SQUASH achieves comparable performance to FRFCFS-Dyn-0.9 and 10.1% better system performance than FRFCFS-DynOpt, while achieving a deadline-met ratio of 100%. We conclude that SQUASH achieves both high CPU performance and 100% QoS for HWAs. In the next section, we present a breakdown of the benefits from the different components of SQUASH.

7.1 Performance Breakdown of SQUASH

In this section, we break down the performance benefits due to the different components of SQUASH. Figure 6 shows the system performance normalized to FRFCFS-DynOpt. The x-axis shows the memory intensities of the workloads. The numbers above the bars of FRFCFS-DynOpt show the absolute values for FRFCFS-DynOpt. We compare four different configurations of SQUASH over FRFCFS-DynOpt: 1) SQ-D (distributed priority on top of TCM for CPU applications), 2) SQ-D+L (SQ-D along with application-aware prioritization between LDP-HWAs and memory-intensive

CPU applications), 3) SQ-D+L+S (SQ-D+L along with worst-case latency based prioritization for SDP-HWAs), and 4) SQ-D+L+S+P (Complete SQUASH mechanism, SQ-D+L+S along with probabilistic prioritization between LDP-HWAs and memory-intensive CPU applications). Table 7 shows the deadline-met ratio for the different mechanisms. Figure 7 shows the bandwidth utilization of different categories of applications when the MAT-HWA has low priority and the fraction of time the MAT-HWA is assigned different priorities.
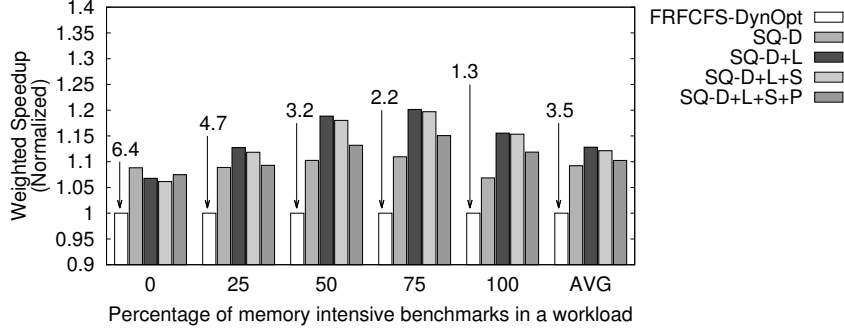

Figure 6: SQUASH performance breakdown for different workload memory intensities

Scheduling Algorithms	Deadline-Met Ratio (%) / Frame Rate (fps)			
	HES	MAT	RSZ	DET
FRFCFS-DynOpt	100 / 30	99.997 / 29.72	100 / 30	99.99 / 25.5
SQ-D	99.999 / 29.875	100 / 30	100 / 30	99.88 / 21
SQ-D+L	99.999 / 29.934	100 / 30	100 / 30	99.86 / 20.44
SQ-D+L+S	100 / 30	100 / 30	100 / 30	100 / 30
SQ-D+L+S+P	100 / 30	100 / 30	100 / 30	100 / 30

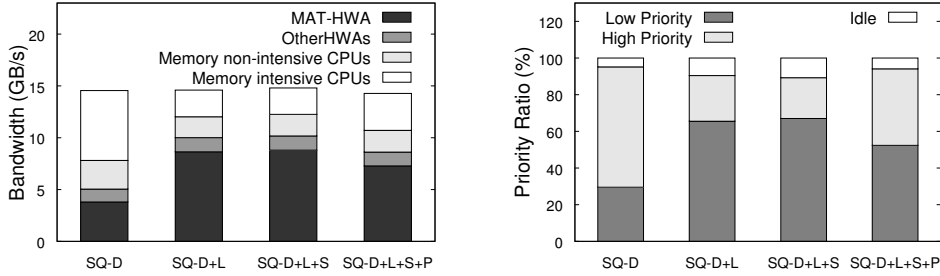Table 7: Deadline-met ratio and frame rate of HWAs for SQUASH components

Figure 7: Bandwidth and priority ratio

We draw four major observations. First, SQ-D improves performance over FRFCFS-DynOpt by 9.2%. However, this improvement comes at the cost of missed deadlines for some HWAs (HES and DET), as shown in Table 7. Second, introducing application-aware prioritization between LDP-HWAs and memory-intensive CPU applications (SQ-D+L) improves performance especially as the memory intensity increases (8.3% maximum over SQ-D). This is because prioritizing HWAs over memory-intensive applications reduces the amount of time HWAs become urgent and interfere with memory-non-intensive CPU applications as shown in Figure 7. However, the SDP-HWAs (HES and DET) still miss some deadlines.

Third, SQ-D+L+S employs worst-case access latency based prioritization for SDP-HWAs, enabling such HWAs to meet their deadlines, while still achieving high performance. However, memory-intensive applications still experience high slowdowns. Fourth, SQ-D+L+S+P tackles this problem by probabilistically changing the prioritization order between memory-intensive CPU applications and LDP-HWAs. This increases the bandwidth allocation of memory-intensive CPU applications, as shown in Figure 7. The result is a 26% average reduction in the maximum slowdown

experienced by any application in a workload, while degrading performance by only 1.7% compared to SQ-D+L+S and achieving 100% deadline-met ratio. We conclude that SQUASH is in achieving high CPU performance, while also meeting the HWAs' deadlines.

7.2 Impact of *EmergentThreshold*

In this section, we study the impact of *EmergentThreshold* on performance and deadline met ratio and the trade offs it enables. Figure 8 shows the average system performance with FRFCFS-Dyn and SQUASH when sweeping *EmergentThreshold* across all 80 workloads using both Config-A and B. We employ the same *EmergentThreshold* value for all HWAs. Tables 8 and 9 show the deadline-met ratio of HWAs with FRFCFS-Dyn and SQUASH respectively.
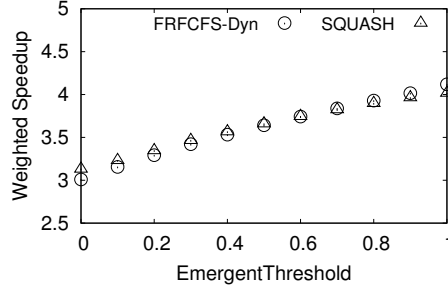

Figure 8: Performance sensitivity to *EmergentThreshold*

| Emergent Threshold | Deadline-Met Ratio (%) | | | | | |
| | Config-A | | Config-B | | | |
	HES	MAT	HES	MAT	DET	RSZ
0-0.1	100	100	100	100	100	100
0.2	100	99.987	100	100	100	100
0.3	99.992	93.740	100	100	100	100
0.4	99.971	73.179	100	100	100	100
0.5	99.945	55.760	99.9996	99.751	99.997	100
0.6	99.905	44.691	99.989	94.697	99.960	100
0.7	99.875	38.097	99.957	86.366	99.733	100
0.8	99.831	34.098	99.906	74.690	99.004	99.886
0.9	99.487	31.385	99.319	60.641	97.149	97.977
1	96.653	27.320	95.798	33.449	88.425	55.773

Table 8: Deadline-met ratio of FRFCFS-Dyn

| Emergent Threshold | Deadline-Met Ratio (%) | | | | | |
| | Config-A | | Config-B | | | |
	HES	MAT	HES	MAT	DET	RSZ
0-0.8	100	100	100	100	100	100
0.9	100	99.997	100	99.993	100	100
1.0	100	68.44	100	75.83	100	95.93

Table 9: Deadline-met ratio of SQUASH

We draw two major conclusions. First, there is a trade-off between system performance and HWA deadline-met ratio, as the *EmergentThreshold* is varied. As the *EmergentThreshold* increases, CPU performance improves at the cost of increase in deadline-met ratio. Second, for a given value of *EmergentThreshold*, SQUASH achieves significantly higher deadline-met ratio than FRFCFS-Dyn, while achieving similar CPU performance, because of distributed priority and application-aware scheduling. Specifically, SQUASH meets all deadlines with an *EmergentThreshold* of 0.8, for Config-A, whereas FRFCFS-Dyn needs an *EmergentThreshold* of 0.1 to meet all deadlines. Furthermore, SQUASH-0.8 achieves 23.5% higher performance than FRFCFS-Dyn-0.1. Based on these observations, we conclude that SQUASH is effective in achieving both high CPU performance and QoS for HWAs.

7.3 Impact of *ClusterFactor*

We study the impact of the *ClusterFactor* used to determine what fraction of total memory bandwidth is allocated to memory-non-intensive CPU applications. Figure 9 shows average CPU performance and fairness with FRFCFS-DynOpt and SQUASH across 80 workloads using Config-A. For SQUASH, we sweep the *ClusterFactor* from 0 to 1.0. All HWAs' deadlines are met for all values of the *ClusterFactor* for SQUASH.
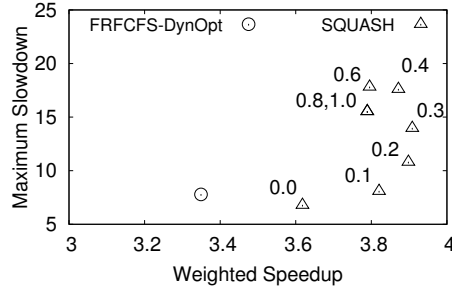

Figure 9: Performance sensitivity to *ClusterFactor*

We draw three major conclusions. First, there is a trade-off between performance and fairness, as the *ClusterFactor* is varied. As the *ClusterFactor* increases, CPU performance improves, but fairness degrades. This is because more CPU applications are classified and prioritized as memory-non-intensive at the cost of degrading the performance of some memory-intensive applications. Second, *ClusterFactor* is an effective knob for trading off CPU performance and fairness. For example, in our main evaluations, we optimize for performance and pick a *ClusterFactor* of 0.2. Instead, if we want to optimize for fairness, we could pick a *ClusterFactor* of 0.1, which still improves performance by 14%, compared to FRFCFS-DynOpt, while degrading fairness by only 3.8% (for Config-A). Third, regardless of the *ClusterFactor*, SQUASH is able to meet all HWAs' deadlines (not shown), since it monitors and assigns enough priority to HWAs based on their progress.

7.4 Effect of HWAs' Memory Intensity

We study the impact of HWAs' memory intensity on a system with 2 MAT-HWAs and 2 HES-HWAs. We vary the memory intensity of the HWAs by varying their parameters in Table 3. As the HWAs' memory intensity increases, CPU performance improvement with SQUASH increases (30.6% maximum) while meeting almost all deadlines (99.99%), when using an *EmergentThreshold* of 0.8. This is because as the memory intensity of HWAs increases, they cause more interference to CPU applications. SQUASH is effective in mitigating this interference.

7.5 Evaluation on systems with GPUs

Figure 10 shows the average CPU performance and frame rate of the MAT-HWA across 30 workloads on a CPU-GPU-HWA system. The other HWAs and the GPU meet all deadlines with all schedulers. For FRFCFS-Dyn, we use an *EmergentThreshold* of 0.9 for the GPU and the threshold values shown in Table 5 for the other HWAs. For SQUASH, we use an *EmergentThreshold* of 0.9 for both the GPU and other HWAs.
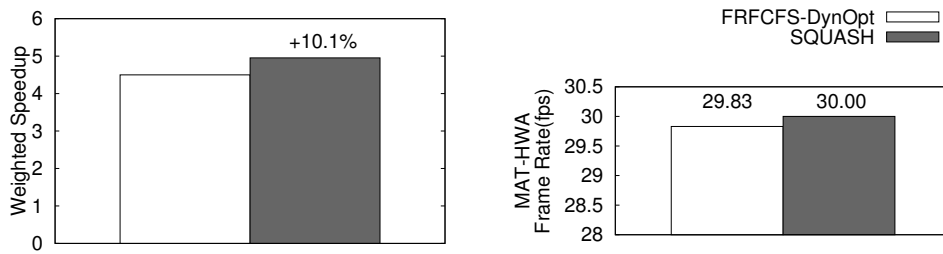

Figure 10: System performance and frame rate

SQUASH achieves 10.1% higher CPU performance than FRFCFS-Dyn, while also achieving a higher frame rate for the MAT-HWA than FRFCFS-Dyn. SQUASH's distributed priority and application-aware scheduling schemes enable higher system performance, while ensuring QoS for the HWAs *and* the GPU. We also evaluate a CPU-GPU system. SQUASH improves CPU performance by 2% over FRFCFS-Dyn, while meeting all deadlines, where FRFCFS-Dyn misses a deadline. We conclude that SQUASH is effective in achieving high system performance and QoS in systems with GPUs as well.

7.6 Sensitivity to System Parameters

7.6.1 Number of Channels.

Figure 11 (left) shows the system performance with different number of channels across 25 workloads (executing 90M cycles) using HWA Config-A (all other parameters are the same as baseline). All HWAs meet all deadlines with all schedulers as there is ample bandwidth. The key conclusion is that as the number of channels decreases, memory contention increases, resulting in increased benefits from SQUASH.
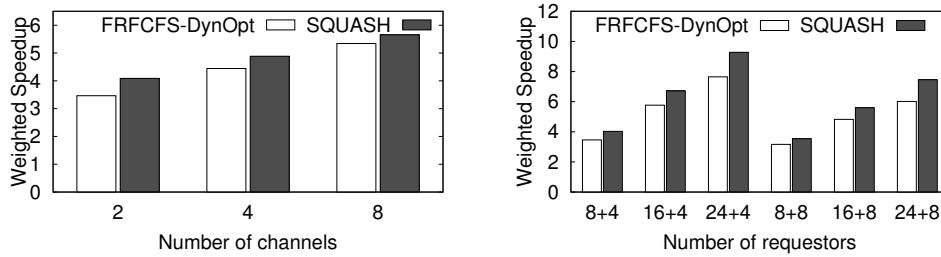

Figure 11: Performance sensitivity to system parameters

7.6.2 Number of Cores.

Figures 11 (right) and 12 show the same performance metrics for the same schedulers as the previous section when using different number of CPU cores (from 8 to 24) and HWAs[4] (4 or 8). We draw three conclusions. First, SQUASH *always* improves CPU performance over FRFCFS-DynOpt. Second, as the number of requestors increases, there is more contention between HWAs and CPU applications, providing more opportunity for SQUASH, which achieves greater performance improvement (24.0% maximum). Finally, SQUASH meets all deadlines for all HWAs.
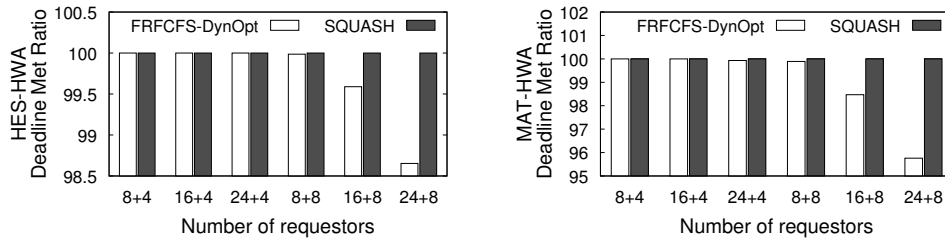

Figure 12: Deadline-met ratio sensitivity to core count

7.6.3 *Scheduling Unit* and *Switching Unit*.

We sweep the *SchedulingUnit* (Section 4.1) from 1000 to 5000 cycles and *SwitchingUnit* (Section 4.4) from 500 to 2000 cycles (*SwitchingUnit* < *SchedulingUnit*). We observe two trends. First, as the *SchedulingUnit* increases, system performance decreases because once a HWA is classified as urgent, it interferes with CPU cores for a longer

---

[4]The 4-HWA configuration is the same as Config-A. The 8-HWA configuration consists of IMG-HWA x2, MAT-HWA(10) x1, MAT-HWA(20) x1, HES-HWA(32) x1, HES-HWA(128) x1, RSZ-HWA x1, and DET-HWA x1.

time. Second, a smaller *SwitchingUnit* provides better fairness, since fine-grained switching of the probability *Pb* enables memory-intensive applications to have higher priority for longer time periods. Based on these observations, we empirically pick a *SchedulingUnit* of 1000 cycles and *SwitchingUnit* of 500 cycles.

8 Related Work

Memory scheduling. We have already compared SQUASH both qualitatively and quantitatively to the state-of-the-art QoS-aware memory scheduler for CPU-GPU systems proposed by Jeong et al. [17]. When this scheduler is adapted to the CPU-HWA context, SQUASH outperforms it in terms of both system performance and deadline-met ratio.

Ausavarungnirun et al. [7] propose Staged Memory Scheduling (SMS) to improve system performance and fairness in a heterogeneous system CPU-GPU system. Unlike SQUASH, SMS does not explicitly attempt to provide QoS to the GPU and aims to only optimize overall performance and fairness.

Most other previously proposed memory schedulers (e.g., [37, 30, 31, 32, 21, 22, 44, 14] have been designed with the goal of improving system performance and fairness in CPU-only multicore systems. These works do not consider the memory access characteristics and needs of other requestors such as HWAs. In contrast, SQUASH is specifically designed to provide high system performance and QoS in heterogeneous systems with CPU cores and HWAs.

Lee et al. [23] propose a quality-aware memory controller that aims to satisfy latency and bandwidth requirements of different requestors, in a best-effort manner. Latency-sensitive requestors are always given higher priority over bandwidth-sensitive requestors. Hence, it might not be possible to meet potential deadline requirements of bandwidth-sensitive requestors with such a mechanism.

Other previous works [6, 36, 33, 48, 27, 19] have proposed to build memory controllers that provide support to guarantee real-time access latency constraints for each master. The PRET DRAM Controller [36] partitions DRAM into multiple resources that are accessed in a periodic pipeline fashion. Wu et al. [48] propose to strictly prioritize real-time threads over non real-time threads. Macian et al. [27] bound the maximum access latency by scheduling in a round-robin manner. Other works [6, 33] group a series of accesses to all banks and schedule at the group unit. All these works aim to bound the worst case latency by scheduling requests in a fixed predictable order. As a result, they waste significant amount of memory bandwidth and do not achieve high system performance.

Source throttling. Memguard [12] guarantees worst case bandwidth to each core by regulating the number of injected requests from each core. Ebrahimi et al. [9] propose to limit the number of memory requests of requestors to improve fairness in CPU-only systems. Other previous works [42, 17] propose to throttle the number of outstanding GPU requests for CPU-GPU systems, in order to mitigate interference to CPU applications. These schemes are complementary to the memory scheduling approach taken by SQUASH and can be employed in conjunction with SQUASH to achieve better interference mitigation.

Memory channel/bank partitioning. Previous works [29, 18, 25] propose to mitigate interference by mapping data of applications that significantly interfere with each other to different channels/banks. Our memory scheduling approach can be combined with a channel/bank partitioning approach to achieve higher system performance and QoS for HWAs.

9 Conclusion

We introduce a simple QoS-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators, SQUASH, with the goal of enabling hardware accelerators (HWAs) to meet their deadlines while providing high CPU performance. Our experimental evaluations across a wide variety of workloads and systems show that SQUASH meets HWAs' deadlines and improves their frame rates while also greatly improving CPU performance, compared to the state-of-the-art techniques. We conclude that SQUASH can be an efficient and effective memory scheduling substrate for current and future heterogeneous SoCs, which will require increasingly more predictable and at the same time high performance memory systems.

References

[1] *NAS Parallel Benchmark Suite*. http://www.nas.nasa.gov/publications/npb.html. 11

[2] *SPEC CPU2006*. http://www.spec.org/spec2006. 11

[3] *Transaction Processing Performance Council*. http://www.tpc.org/. 11

[4] L. Acasandrei and A. Barriga. AMBA bus hardware accelerator IP for Viola-Jones face detection. *Computers Digital Techniques, IET*, 7(5), September 2013. 4

[5] Advanced Micro Devices. AMD Radeon HD 5870 Graphics. 12

[6] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A Predictable SDRAM Memory Controller. In *CODES+ISSS*, 2007. 18

[7] R. Ausavarungnirun, K. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged Memory Scheduling: Achieving high performance and scalability in heterogeneous systems. In *ISCA*, 2012. 2, 18

[8] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool. SURF: Speeded Up Robust Features. In *CVIU*, 2008. 11

[9] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS*, 2010. 18

[10] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, (3), 2008. 12

[11] P. N. Gour, S. Narumanchi, S. Saurav, and S. Singh. Hardware accelerator for real-time image resizing. In *VLSI Design and Test, 18th International Symposium on*, 2014. 4

[12] Y. Heechul, Y. Gang, P. Rodolfo, C. Marco, and S. Lui. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, 2013. 18

[13] F.-C. Huang, S.-Y. Huang, J.-W. Ker, and Y.-C. Chen. High-Performance SIFT Hardware Accelerator for Real-Time Image Feature Extraction. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(3), March 2012. 4

[14] I. Hur and C. Lin. Adaptive History-Based Memory Schedulers. In *MICRO*, 2004. 18

[15] Itseez. Open Source Computer Vision. 11, 12

[16] JEDEC. Standard No. 79-3. DDR3 SDRAM STANDARD, 2010. 8

[17] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. A QoS-aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *DAC-49*, 2012. 2, 4, 5, 12, 18

[18] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM locality and parallelism in shared memory CMP systems. In *HPCA*, 2012. 18

[19] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Roitzsch. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014. 18

[20] W. Kim, H. Chung, H.-D. Cho, and Y. Kim. Enjoy the Ultimate WQXGA Solution with Exynos 5 Dual. *Samsung Electronics White Paper*, 2012. 2

[21] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010. 1, 4, 7, 10, 12, 18

[22] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010. 1, 4, 7, 8, 9, 10, 12, 18

[23] K.-B. Lee, T.-C. Lin, and C.-W. Jen. An efficient quality-aware memory controller for multimedia platform SoC. *IEEE Transactions on Circuits and Systems for Video Technology*, May 2005. 18

[24] S. E. Lee, Y. Zhang, Z. Fang, S. Srinivasan, R. Iyer, and D. Newell. Accelerating mobile augmented reality on a handheld platform. In *ICCD*, 2009. 11, 12

[25] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, 2012. 18

[26] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005. 11

[27] C. Macian, S. Dharmapurikar, and J. Lockwood. Beyond performance: secure and fair memory management for multiple systems on a chip. In *FPT*, 2003. 18

[28] Micron. 1Gb: x4, x8, x16 DDR3 SDRAM Features. 11

[29] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *MICRO*, 2011. 18

[30] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007. 1, 4, 10, 18

[31] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008. 1, 4, 10, 18

[32] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO*, 2006. 1, 4, 18

[33] M. Paolieri, E. Quiones, F. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded Systems Letters*, Dec 2009. 18

[34] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO*, 2004. 11

[35] Qualcomm. Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age. *Qualcomm White Paper*, 2011. 1, 2

[36] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS*, 2011. 18

[37] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000. 4, 12, 18

[38] I. Schmadecke and H. Blume. Hardware-accelerator design for energy-efficient acoustic feature extraction. In *Consumer Electronics (GCCE), 2013 IEEE 2nd Global Conference on*, 2013. 4

[39] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000. 12

[40] I. Sobel. An isotropic 3x3 image gradient operator. In *Machine Vision for Three-Dimensional Scenes*, pages 376–379. Academic Press, 1990. 3

[41] G. P. Stein, I. Gat, and G. Hayon. Challenges and Solutions for Bundling Multiple DAS Applications on a Single Hardware platform. In *V.I.S.I.O.N.*, 2008. 3

[42] A. Stevens. QoS for High-Performance and Power-Efficient HD Multimedia. *ARM White Paper*, 2010. 4, 18

[43] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. The blacklisting memory scheduler: Achieving high performance and fairness at low cost. In *ICCD*, 2014. 1

[44] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA*, 2013. 4, 18

[45] Y. Tanabe, M. Sumiyoshi, M. Nishiyama, I. Yamazaki, S. Fujii, K. Kimura, T. Aoyama, M. Banno, H. Hayashi, and T. Miyamori. A 464GOPS 620GOPS/W heterogeneous multi-core SoC for image-recognition applications. In *ISSCC*, 2012. 1, 3

[46] H. Vandierendonck and A. Seznec. Fairness metrics for multi-threaded processors. *IEEE CAL*, February 2011. 12

[47] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, 2001. 11

[48] L. Wu and W. Zhang. Time-predictable DRAM access scheduling algorithms for real-time multicore processors. In *Southeastcon*, 2013. 18