DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators

HIROYUKI USUI, LAVANYA SUBRAMANIAN, KEVIN KAI-WEI CHANG, and ONUR MUTLU, Carnegie Mellon University

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 it is not controlled well, is missed deadlines for HWAs and low CPU performance. Few previous works have tackled this problem. 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* 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 article, we propose a Deadline-Aware memory Scheduler for Heterogeneous systems (DASH), which overcomes these problems using three key ideas, with the goal of meeting HWAs' deadlines while providing high CPU performance. First, DASH prioritizes an 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, DASH prioritizes HWAs over memory-intensive CPU applications based on the observation that memory-intensive applications' performance is not sensitive to memory latency. Third, DASH 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 DASH 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.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**; *System on a chip*

Additional Key Words and Phrases: Hardware accelerators, main memory, scheduling, performance, deadline, memory controller, GPUs, multicore, real-time systems, heterogeneous systems, system-on-a-chip

ACM Reference Format:
Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. 2015. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. ACM Trans. Archit. Code Optim. 12, 4, Article 65 (December 2015), 28 pages.
DOI: http://dx.doi.org/10.1145/2847255

We acknowledge the generous support from our industrial partners: Google, Intel, Microsoft, Nvidia, Samsung, VMware. This work is supported in part by NSF grants 0953246, 1212962, 1065112, 1320531, and 1409723; the Semiconductor Research Corporation; Toshiba Corporation; and the Intel Science and Technology Center on Cloud Computing.
Authors' addresses: H. Usui, Toshiba Corporation, 580-1, Horikawa-Cho, Saiwai-ku, Kawasaki 212-8520, Japan; email: hiroyuki1.usui@toshiba.co.jp; L. Subramanian, K. Chang, and O. Mutlu, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh PA 15213; emails: lavanya.cmu@gmail.com, kevincha@cmu.edu, onur@cmu.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 1544-3566/2015/12-ART65 $15.00
DOI: http://dx.doi.org/10.1145/2847255

1. INTRODUCTION

Today's SoCs are heterogeneous architectures that integrate hardware accelerators (HWAs) and CPUs. Special-purpose HWAs 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 [Chandramoorthy et al. 2015]. For example, CPU cores and Graphics Processing Units (GPUs) are often integrated in smartphone SoCs [Qualcomm 2011]. Hard-wired HWAs are implemented in a very wide range of SoCs [Tanabe et al. 2012; Stein et al. 2008; Tanabe et al. 2015], including smartphones.

To meet their target performance (e.g., frame rates), HWAs need to meet *deadlines*. However, in most SoCs, HWAs share the main memory with CPU cores and main memory bandwidth is a heavily contended resource between these different agents [Stevens 2010; Yedlapalli et al. 2014; Nachiappan et al. 2014]. Requests to main memory from CPU cores and HWAs contend for the limited memory bandwidth. Unmanaged memory bandwidth contention prevents HWAs from meeting their deadlines and also degrades CPU performance. Therefore, it is important to manage the main memory bandwidth such that HWAs meet their deadlines and performance targets, while CPU cores also achieve high performance.

Several previous works have explored application-aware memory scheduling in CPU-only multicore systems [Mutlu and Moscibroda 2007; Nesbit et al. 2006; Mutlu and Moscibroda 2008; Kim et al. 2010a, 2010b; Subramanian et al. 2014; Moscibroda and Mutlu 2008; Zhao et al. 2014]. 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 state-of-the-art work [Jeong et al. 2012a] that has these two goals proposes a memory scheduling scheme that aims to enable only one specific type of HWA, *GPU*, to meet its target deadline (i.e., frame rate) while achieving high CPU performance. Its key idea is to prioritize the GPU over the CPU cores *only close to a deadline*, that is, when the GPU has to finish rendering a frame. At other times, when the GPU is *not on track* to meet a deadline, the proposed scheduler assigns the GPU *the same priority* as the CPU cores. On the other hand, it assigns the GPU lower priority than the CPU cores when the GPU is on track to meet a deadline. However, this work does not consider many other kinds of HWAs, which we evaluate in our heterogeneous systems.

We adapted this state-of-the-art scheme [Jeong et al. 2012a] to a heterogeneous system with CPUs and various types of HWAs and observed that such an approach, when used in a CPU-HWA context, suffers from two major problems. First, it prioritizes an HWA over the CPU cores *only when* the HWA is close to a deadline, and during other times, it gives the HWA *the same priority* as the CPU cores, at best, 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) 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 *Deadline Aware memory Scheduler for Heterogeneous systems (DASH)* is based on three key ideas.

First, to tackle the problem of HWAs missing their deadlines, DASH prioritizes an 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 or giving it the same priority as the CPU cores. This allows each HWA to receive *consistent memory bandwidth*

throughout its runtime. Second, DASH exploits the heterogeneous memory access characteristics of different *CPU* applications and prioritizes HWAs over *memory-intensive* CPU applications, which are not greatly sensitive to additional memory latency, *even when* HWAs are on track to meet their deadlines. This reduces the amount of time HWAs are prioritized over *memory-nonintensive* CPU applications that are latency sensitive, thereby achieving high overall CPU performance. Third, DASH exploits the heterogeneous access characteristics of different *HWAs*. We observe that an HWA with a *short deadline period* needs a short burst of high priority *long enough* to ensure that its few requests are served, rather than consistent memory bandwidth. DASH achieves this by prioritizing such HWAs for a short time period based on their estimated worst-case memory access latency.

This article 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 DASH, a new deadline-aware memory scheduler that *always* meets HWAs' deadlines while greatly improving CPU performance over the best previous scheduler.
—We compare DASH to four different memory schedulers across a wide variety of system configurations and workloads. We show that DASH improves CPU performance by 9.5% compared to the best previous scheduler, while always meeting deadlines for all HWAs and GPUs.

2. BACKGROUND

In this section, we first provide an overview of heterogeneous SoC architectures and HWAs that are significant components of heterogeneous SoCs. Next, we provide a brief background on the organization of 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 SoC designed for smartphones [Qualcomm 2011; Kim et al. 2012]. The CPU is used to perform general-purpose computation. HWAs are employed to accelerate various functions. For instance, the GPU is optimized for graphics. 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 [Tanabe et al. 2012; Stein et al. 2008; Tanabe et al. 2015] because image recognition requires a large amount of computation.

At design time, an HWA designer calculates a deadline for each execution of the HWA from the high-level performance target (e.g., frame rate) and designs the HWA such that its computation finishes ahead of the deadline. However, in an SoC, not only HWAs' computation but also all their memory accesses need to be finished ahead of the deadline. It is difficult to estimate memory access bandwidth and latency in the design phase, because the DRAM main memory is shared between multiple HWAs [Tanabe et al. 2015; Nachiappan et al. 2014; Stevens 2010] and CPU cores. Therefore, satisfying the memory requirements of all these requestors (or agents) becomes a major challenge. In this work, we focus on managing memory bandwidth between the CPU cores and HWAs with the goal of meeting deadline requirements for HWAs while improving CPU performance.

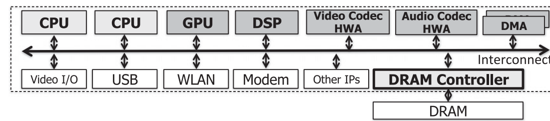

Fig. 1. Example heterogeneous SoC architecture.

2.2. Hardware Accelerator Characteristics

Modern SoCs consist of a wide variety of HWAs, which accelerate specific functions. The functions that they accelerate are diverse and the implementations vary among different HWAs. As an example, we first describe a 3×3 *horizontal Sobel filter* HWA [Sobel 1990] (shown in Figure 2), which computes the gradient of an image.

The accelerator executes the Sobel filter on a target 640x480 image, at a target frame rate of 30 frames per second (fps). A typical implementation of 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 prefetching of the next line have to be finished within 69.44μsec ($= 1\ sec/30\ \ frames/480\ lines$). In this case, the *period* of the HWA is 69.44μsec 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. To provision for more capacity to hold prefetched data, double buffers (e.g., frame buffers) 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 [Gour et al. 2014] or feature extraction [Huang et al. 2012; Acasandrei and Barriga 2013] use line buffers. HWAs for acoustic feature extraction [Schmadecke and Blume 2013] use frame buffers. In all these HWAs, computing engines can access only line/frame buffers and data is prefetched into these buffers from main memory. We employ and evaluate some of these HWAs that utilize line/frame buffers in our heterogeneous systems. We provide the detailed description of these HWAs in Section 6.3.

One common attribute across all these HWAs is that the amount of buffer capacity determines the *deadline*, or *period*, and how much data needs to be accessed from memory for each *period*. For instance, in the Sobel filter HWA example described previously, the HWA requires 640 bytes for every 69.44μsec. 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 not beneficial for the HWA. Hence, statically prioritizing HWAs is often wasteful, especially in a system with other agents such as CPUs, where the memory bandwidth can be better utilized to achieve higher performance for the other agents.

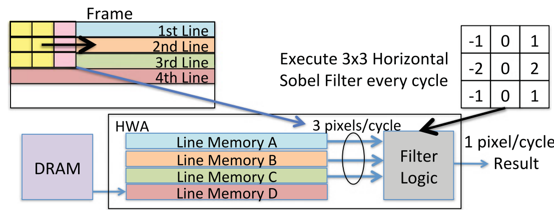

Fig. 2. Typical implementation of a Sobel filter HWA.

As a result, a major challenge in today's heterogeneous SoCs is how to ensure that the HWAs get consistent 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 that focus on either the HWAs or the CPUs. As we will show in our evaluations (Section 7), an HWA-friendly memory scheduler that achieves almost a 100% *deadline-met ratio* for HWAs has 12% lower performance compared to a CPU-friendly memory scheduler that attains the highest CPU performance without always meeting the HWAs' deadlines. **The goal of our work** is *to both meet the HWAs' deadlines and attain high 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 2 times the latency of a row hit [Mutlu and Moscibroda 2008; Rixner et al. 2000]. For more detail on DRAM organization and internals, we refer the reader to Kim et al. [2012], Lee et al. [2013], Liu et al. [2012a], Lee et al. [2015], Seshadri et al. [2013, 2015], Chang et al. [2014], and Khan et al. [2014].

2.4. Managing Memory Bandwidth in Heterogeneous Systems

Many previous works have tackled the problem of managing memory bandwidth between applications in CPU-only multicore systems [Mutlu and Moscibroda 2007, 2008; Nesbit et al. 2006; Kim et al. 2010a, 2010b; Subramanian et al. 2013, 2015b, 2014; Moscibroda and Mutlu 2008; Muralidhara et al. 2011; Zhao et al. 2014; Das et al. 2013; Ebrahimi et al. 2010; Kim et al. 2014]. 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 [Stevens 2010] observes that CPU cores are latency sensitive, whereas the GPU is bandwidth sensitive with high memory latency tolerance. Therefore, they propose to prioritize CPU requests over GPU requests. However, with such a scheme, the GPU cannot always achieve its performance target when the CPU demands high memory bandwidth [Jeong et al. 2012a].

To address this challenge of managing memory bandwidth between the CPU cores and the GPUs, a state-of-the-art technique [Jeong et al. 2012a] proposes to dynamically adjust memory access priorities between the CPU cores and the GPU. This policy compares current progress in terms of the fraction 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. To enable the GPU to make faster progress, its requests are given the *same* priority as CPU requests. Only when the *ExpectedProgress* is greater than an *EmergentThreshold* (=0.90) is the GPU given higher priority than the CPU. Such a policy aims to preserve CPU performance while still giving the GPU the highest priority close to the end of a frame. However, this policy, when used within the context of a CPU-HWA system, is not adaptive enough 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 kinds of 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 best previous work [Jeong et al. 2012a] that only targets CPU-GPU systems (discussed in the previous section), we adapt their scheduling policy and evaluate its efficacy on heterogeneous systems with CPU cores and a wide variety of HWAs. We call this adapted policy *Dyn-Prio* and now briefly describe how we design Dyn-Prio such that it can be used with various types of HWAs.

Similar to GPUs' frame rate requirements, HWAs need to meet deadlines. We target HWAs having soft deadlines, such as HWAs for image processing and recognition. A deadline miss for such HWAs causes frames to be dropped. We redefine *CurrentProgress* and *ExpectedProgress* originally shown in Equations (1) and (2) as shown in Equations (3) and (4), respectively, to capture various 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. The progress is monitored every period (*SchedulingUnit*). 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 for two reasons. First, as described in Section 2.2, a lot of HWAs for media processing access data in a streaming manner [Gour et al. 2014; Huang et al. 2012; Acasandrei and Barriga 2013], resulting in a predictable/prefetch-friendly access stream. Second, when an HWA is implemented with a line-/double-buffer, all

the data required for the next set of computations need to be prefetched into the buffer ahead of a deadline. 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 is required to be prefetched.

We observe that there are two major problems with the Dyn-Prio policy when it is used in a CPU-HWA context. First, it prioritizes an HWA over CPU cores *only* when it is 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 CPU cores. We will explain this problem in greater detail in Section 3.1. Second, Dyn-Prio is designed for a simple 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 (CPUs or HWAs). As we will explain in Sections 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 miss their deadlines, we propose a simple modification. An HWA enters a state of urgency and is given the highest priority any time 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 an 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 runtime.

To illustrate the benefits of such a policy, we study an example system with two CPU cores and an 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 few memory requests. In contrast, `CPU-B` has high memory intensity and generates more memory requests than `CPU-A` does. `HWA` has double buffers and generates 10 prefetch requests during each 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 20T. If 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 18T. 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 (since 10 requests require 10T units of time to be serviced, but *Dyn-Prio* leaves only 2T units before the deadline for them). Hence, the *Dyn-Prio* scheme would violate the HWA's deadline requirements.

Figure 4(a) illustrates the scheduling order of requests from a system with an HWA (`HWA`) and two CPU cores (`CPU-A` and `CPU-B`) using our proposed *Dist-Prio* scheme with a *SchedulingUnit* of 4T. 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 0.4 (4/10) and *ExpectedProgress* is 0.2 (4/20). 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 0.4, `HWA` is deemed as

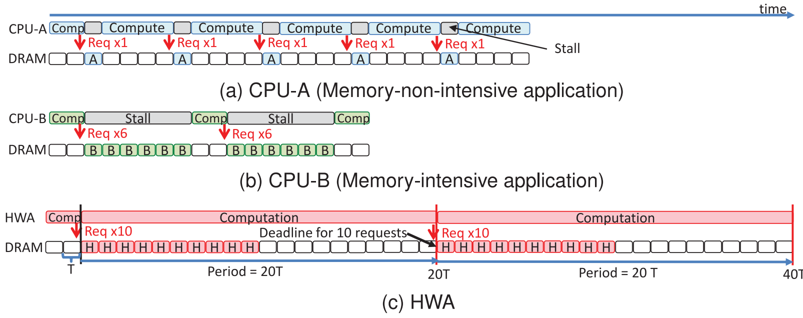

Fig. 3. Memory service timeline example when each agent is executed alone.

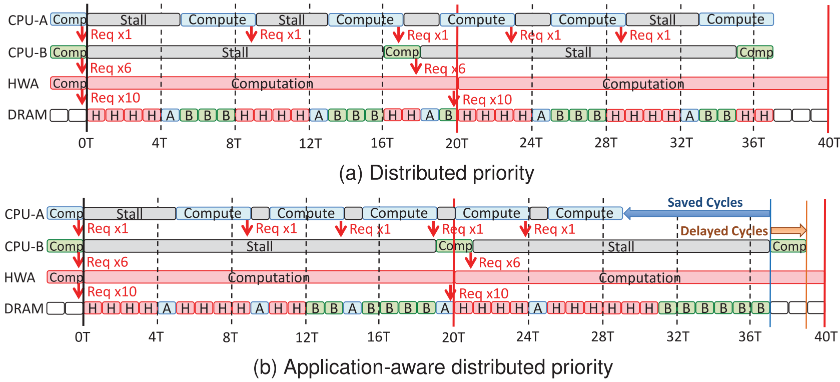

Fig. 4. Memory service timeline example when all agents execute together.

urgent again and four requests of HWA are served. In the next 4T cycles, HWA is deemed as not urgent. After 16T cycles, HWA is deemed as urgent (both *CurrentProgress* and *ExpectedProgress* are 0.8) and its remaining two requests are completed. Hence, Dist-Prio enables the HWA to meet its deadlines while also achieving good CPU performance by distributing the times when HWA is prioritized over its entire deadline period.

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 4(a), during cycles 8T to 12T, HWA stalls both CPU-A and CPU-B. Furthermore, the higher the memory intensity of the HWAs, the greater 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 CPU applications do not experience significant performance degradation when HWAs are prioritized over them.

Often, applications with low memory intensity are more sensitive to memory latency, since they generate few memory requests, and quick service of these 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.

[1]This was also observed by previous works in the context of multicore memory scheduling [Kim et al. 2010a, 2010b; Lee et al. 2009a; Mutlu and Moscibroda 2008].

This is because they generate a large number of memory requests, followed by a small amount of computation at the cores. Hence, even if some requests of such applications are served quickly, subsequent requests stall the pipeline again. Therefore, applications with high memory intensity are less sensitive to memory latency, as observed by previous works [Kim et al. 2010a, 2010b; Lee et al. 2009a; Mutlu and Moscibroda 2008] and as we also show in Section 7.3. Based on this observation, we propose to prioritize HWAs' 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-nonintensive* CPU applications that are latency sensitive, thereby improving the performance of latency-sensitive CPU applications.

Figure 4(b) illustrates the benefits of such an application-aware distributed priority scheme for the same set of requests shown in Figure 4(a). The request schedule remains the same during the first 4T cycles. At time 4T, `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 0.7, which is greater than *ExpectedProgress*. As such, the HWA is *still* deemed *not urgent*, unlike in the Dist-Prio scheme (in Figure 4(a)). Hence, the latency-sensitive `CPU-A`'s requests are served earlier. Thus, prioritizing `HWA`'s requests over the memory-intensive `CPU-B`'s requests enables faster progress for the memory-nonintensive `CPU-A`, as can be seen from Figure 4(b), and `CPU-B` is delayed by only 2T since it already has a large number of requests queued at memory. Therefore, our second key idea results in higher overall CPU performance, while meeting `HWA`'s deadlines.

3.3. Key Idea 3: Application-Aware Scheduling for HWAs

Prioritizing HWAs' requests based on their progress is an effective mechanism to ensure consistent bandwidth to HWAs that have fairly long periods. However, such a scheme is not effective for HWAs with short periods since it is difficult to ensure that these HWAs receive enough priority for sufficient amounts of time within a short deadline period. Specifically, a short deadline provides little time for *all* of an HWA's requests to be served and causes the HWAs to be more susceptible to interference from other agents. We evaluated our previous two key ideas on a heterogeneous system with two HWAs (HWA-A and HWA-B) that have vastly different period lengths (i.e., 63,041 and 5,447 cycles) and bandwidth requirements (i.e., 8.32GB/s and 475MB/s). Our results show that HWA-A meets all its deadlines, whereas HWA-B, on average, misses a deadline every 2,000 periods.

To enable better deadline-met ratios for HWAs with short deadlines, we make the following two observations. First, short-deadline-period HWAs can be enabled to meet their deadlines by giving them a short burst of highest priority close to the deadline. Second, prioritizing short-deadline-period HWAs does not cause much interference to other requestors because these HWAs generally 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 the highest priority for $WorstCaseLatency * NumberOfRequests$ cycles close to its deadline.

4. MECHANISM

In this section, we describe the details of DASH, 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 DASH 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 DASH mechanism.

Overview. DASH 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 a long-deadline-period HWA (*LDP-HWA* for short), DASH monitors its progress periodically and appropriately prioritizes it, enabling it to get sufficient and consistent bandwidth over each of its deadline periods to meet its deadlines (Section 3.1). For a short-deadline-period HWA (*SDP-HWA* for short), DASH gives it a short burst of highest priority close to each deadline, based on worst-case access latency calculations (Section 3.3). DASH also treats memory-intensive and memory-nonintensive CPU applications differently with respect to their priority over HWAs (Section 3.2).

4.1. LDP-HWAs Versus CPU Applications

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

Each CPU application's memory intensity is monitored and applications are classified as memory nonintensive or memory intensive periodically based on their calculated memory intensity using the classification mechanism borrowed from Kim et al. [2010b]. Note that other mechanisms can also be employed to perform this classification.[2]

Based on this classification, the *Dist-Prio* and application-aware components of DASH schedule requests at the memory controller in the following priority order (lower number indicates higher priority):

(1) Urgent HWAs
(2) Memory-nonintensive CPU applications
(3) Nonurgent HWAs
(4) Memory-intensive CPU applications

Based on our first key idea in Section 3.1, HWAs become urgent when they are not on track to meet a deadline, and such urgent HWAs' requests are given highest priority. Based on our second key idea 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 SDP-HWAs

While using Dist-Prio can provide consistent bandwidth to LDP-HWAs to meet their deadlines, SDP-HWAs do not get enough bandwidth to meet their deadlines (as we described in Section 3.3). 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*

[2]Also note that even though we borrow the classification mechanism of Kim et al. [2010b] to categorize memory-intensive and memory-nonintensive applications, the problem we solve and the scheduling policy we devise are very different from those of Kim et al. [2010b], which do not consider HWAs.

memory latency calculations. We classify an HWA as SDP-HWA if its deadline is less than a threshold value (we use a threshold of 10μs in our evaluations).

Estimating the worst-case access latency. In the worst case, all requests from an 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 the *Urgent Period Length (UPL)*.

For example, when an HWA outputs 16 requests every 2,000ns period and tRC is 50ns, the HWA is deemed urgent and given the highest priority for 800 (16×50) ns $+\alpha$ during each period, where α is the waiting time for the in-flight requests to finish. Furthermore, finishing an HWA's requests much earlier than the deadline does *not* improve the HWA's performance any further. Hence, this highest priority can be given to the HWA 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.[4]

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

(1) DASH calculates the UPL of each SDP-HWA x as $UPL(x) = tRC * NumberOfRequests(x)$
(2) Among the urgent SDP-HWAs, the HWAs with shorter deadline periods are given higher priority.
(3) DASH 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 by how long we should extend an SDP-HWA x's UPL, we calculate how many deadline periods (N_i) of each higher priority SDP-HWA (i) can interfere with the UPL of x: $N_i = \lceil (UPL(x)/Period(i)) \rceil$. We then calculate the total length of high-priority UPL, $HP\text{-}UPL(i)$, 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 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. Combined Scheduling Policy for All HWAs

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

(1) **Urgent short-deadline-period HWAs** (Higher priority to shorter-deadline HWAs)
(2) **Urgent long-deadline-period HWAs** (Higher priority to earlier-deadline HWAs)
(3) **Memory-nonintensive CPU applications** (Higher priority to lower-memory-intensity applications)

[3]Accesses to different rows in the same bank have to be spaced apart by a fixed time of tRC based on the DRAM specification [JEDEC 2010; Kim et al. 2012; Lee et al. 2013].

[4]Note that a recent work discusses how to estimate the worst-case delay in a DRAM system [Kim et al. 2014]. Our approach is similar.

(4) **Nonurgent long-deadline-period HWAs** (Higher priority to earlier-deadline HWAs)
(5) **Memory-intensive CPU applications** (Application priorities are shuffled as in Kim et al. [2010b])
(6) **Nonurgent short-/long-deadline-period HWAs** (Higher priority to earlier-deadline HWAs)

In this scheduling order, memory-intensive CPU applications (*Group*5) are always ranked lower than memory-nonintensive CPU applications (*Group*3) and LDP-HWAs (*Group*2, *Group*4). This can potentially always deprioritize memory-intensive applications when the memory bandwidth is only enough to serve memory-nonintensive applications and HWAs. To ensure memory-intensive applications receive sufficient memory bandwidth to make progress, we employ a clustering mechanism that allocates only a fraction of total memory bandwidth (called *ClusterFactor*) to the memory-nonintensive group [Kim et al. 2010b].

As we explain in Section 3.1, the initial state of LDP-HWAs is urgent. When HWAs meet their expected progress, they enter the nonurgent state. Nonurgent LDP-HWAs can be in *Group*4 or *Group*6. They are assigned to *Group*6 only when they first transition to the nonurgent state, but they are assigned to *Group*4 when they re-enter the nonurgent state later on. The rationale is that LDP-HWAs do not need to be prioritized over memory-intensive 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.

Despite these aspects of the scheduling mechanism that attempt to provide enough bandwidth to memory-intensive CPU applications, we observe that memory-intensive applications experience unfair slowdowns due to interference from LDP-HWAs in some workloads. For example, after an LDP-HWA is assigned to *Group*4, the LDP-HWA cannot switch back to *Group*6 until the next deadline even if memory bandwidth demand from CPU cores is decreased. Consequently, the LDP-HWA interferes with memory-intensive applications excessively. We tackle this challenge in the next subsection.

4.4. Probabilistic Switching of Priorities

To ensure memory-intensive CPU applications do not get unfairly slowed down due to interference from LDP-HWAs, we probabilistically prioritize memory-intensive applications over nonurgent 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 increase an LDP-HWA's bandwidth allocation when it is not making sufficient progress. This probabilistic switching helps ensure that the memory-intensive applications are treated more fairly.

ALGORITHM 1: Scheduling using $Pb(x)$

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

ALGORITHM 2: Controlling $Pb(x)$ for an LDP-HWA

```
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
```

4.5. Estimating Memory Requirements of HWAs

DASH is based on the assumption that the number of memory requests served during a deadline period for an HWA can be known ahead of time. As explained in Section 3, we can precisely determine this number for many kinds of HWAs that use scratchpad memory [Gour et al. 2014; Huang et al. 2012; Acasandrei and Barriga 2013; Schmadecke and Blume 2013; Lee et al. 2009b], especially in the media processing space, which is a major segment of today's SoC market. However, if the execution of an HWA changes in a data-dependent manner, and the HWA can access the memory directly, the access pattern of the HWA can vary dynamically. In such cases, we can use (1) worst-case or (2) average-case access pattern estimates by observing memory access behavior during past deadline periods. We leave the exploration of such estimation mechanisms to future work.

5. IMPLEMENTATION AND HARDWARE COST

DASH requires hardware support to monitor HWAs' 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 Kim et al. [2010a, 2010b]. If HWAs access shared caches, the number of completed requests at the shared caches for each HWA is sent to the meta-controller.

Table I lists the major counters required for the meta-controller over a baseline TCM scheduler [Kim et al. 2010b], the state-of-the-art application-aware memory scheduler for multicore systems, which we later provide comparisons 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 LDP-HWAs and memory-intensive applications (as explained in Section 4.4). A 4-byte counter is sufficient to store each of these quantities. Hence, the total counter overhead is 20 bytes for each LDP-HWA and 12 bytes for each SDP-HWA.

Total-Req and *Total-Cyc* are set by the (system) software based on the specification of each HWA. 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 to determine how urgent each LDP-HWA is. For the SDP-HWAs, their state of urgency is determined based on *Priority-Cyc* and *Curr-Cyc*. *Priority-Cyc* is set by the (system) software based on each HWA's specification. This information is used along with *Pb* to determine the scheduling order across all HWAs and CPU applications. The counters are reset

Table I. Storage Required for DASH

For long-deadline-period HWAs	
Name	Function
Curr-Req	Number of requests completed in a deadline period
Total-Req	Total number of requests to be 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 that 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

to zero at the beginning of each period. 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 [Mutlu and Moscibroda 2007, 2008; Kim et al. 2010a, 2010b].

6. METHODOLOGY

6.1. System Configuration

We use an in-house cycle-level simulator to perform our evaluations. We plan to make the simulator publicly available at CMU SAFARI Research Group [2015b]. We model a system with eight x86 CPU cores and four HWAs for our main evaluations. To avoid starving CPU cores or HWAs, we allocate half of the memory request buffer entries to CPU cores and the other half to HWAs. Unless stated otherwise, our system configuration is as shown in Table II. The DRAM model we use is similar to Ramulator's DDR3 model [Kim et al. 2015], which is publicly available [CMU SAFARI Research Group 2015a].

In a typical SoC, applications are run on the CPU cores and they offload specific functionality onto the HWAs. This is the ideal scenario we would like to model in our evaluations. However, it is difficult to get access to such workloads that capture this execution model and communication between the CPU cores and the HWAs. We attempt to, instead, model this scenario by using a combination of CPU workloads and HWA workloads that execute independently. We believe this is a reasonable approximation, since CPU cores and HWAs communicate on a coarse-grained basis in most reasonable current implementations.

Our infrastructure does not model feedback from missed deadlines of the HWAs. We seek to capture the impact of missed deadlines in terms of the *deadline-met ratio metric*. Such missed deadlines could stall the subsequent operations on the CPU cores and HWAs. As a result, contention at the memory controller could be lighter after a missed deadline than what we model in our simulation infrastructure. We evaluate a wide range of workload memory intensities to capture the impact of variations in memory controller load due to different HWA behavior and deadline hit/miss behavior. The results we present are likely pessimistic against our DASH scheduler, since in a more realistic system, deadlines would be *easier* to meet with lighter memory controller load after deadline misses. Therefore, we expect our mechanism to be more effective in reality than the results we report using simulation.

Table II. Configuration of the Simulated System

CPU	8 cores, 2.66GHz, 3-wide issue, 128-entry instruction window, 16 MSHRs/core
L1Cache	Private, 2 way, 32KB, 64-byte line
L2Cache	Shared, 16 way, 4MB, 64-byte line
HWA	4 HWAs
DRAM	DDR3-1333 (9-9-9) [Micron 2014], 300 request buffer entries 2 channels, 1 rank per channel, 8 banks per rank

6.2. Workloads for CPUs

We construct 80 multiprogrammed workloads from the SPEC CPU2006 suite [Standard Performance Evaluation Corporation 2014], TPC [TPC 2015], and the NAS parallel benchmark suite [NASA 2012]. We use Pin [Luk et al. 2005] with PinPoints [Patil et al. 2004] to extract representative phases. We classify CPU benchmarks into two categories, memory intensive and memory nonintensive, 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 memory intensive. Otherwise, it is classified as memory nonintensive. 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 III. 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 1920×1080. 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 [Lee et al. 2009b]. Their implementations are based on Lee et al. [2009b]. HES-HWA accelerates the fast Hessian detector that is executed in SURF (Speed-Up Robust Features) [Bay et al. 2008], which is used to detect interesting points in images and generate descriptors. MAT-HWA accelerates the operation of matching descriptors generated by SURF against those in a database. Their configuration parameters are as shown in Table III. We evaluate HES-HWA and MAT-HWA for three different configurations. The periods and memory bandwidth requirements of the HWAs are different depending on the configuration. HES-HWA fetches some pixels in each line into its line memory and scans the target image vertically. Therefore, each line access is discontinuous. MAT-HWA accesses interesting points of each image in a streaming manner. 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 [Viola and Jones 2001], included in Open-CV [Itseez 2015]. RSZ-HWA shrinks the target frame recursively 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. These HWAs fetch image data into their scratchpad memories in a streaming manner. RSZ-HWA writes two images, the shrunk image and the integral image, to DRAM. 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 III. We simulate for 200 million CPU cycles. Memory traffic generators model HWAs and generate memory requests based on the memory

Table III. Configuration of the HWAs

	Period	Memory Bandwidth	Scratchpad	Deadline Group
IMG-HWA	33ms	360MB/s	double buffer (1 frame × 4)	long
HES-HWA(32)	2us	478MB/s	line buffer (32 lines) 30 lines for computation 2 lines for prefetch	short
HES-HWA(64)	4us	329MB/s		
HES-HWA(128)	8us	224MB/s		
MAT-HWA(30)	23.6us	8.32GB/s	double buffer (4KB × 4) 4KB × 2 for query 4KB × 2 for database	long
MAT-HWA(20)	35.4us	5.55GB/s		
MAT-HWA(10)	47.2us	2.77GB/s		
RSZ-HWA	46.5us–5,183us	2.07GB/s–3.33GB/s	double buffer (1 frame × 4)	long
DET-HWA	0.8us–9.6us	1.60GB/s–1.86GB/s	line buffer (26 lines) 24 lines for computation	short

	Parameters
HES-HWA(N) [Lee et al. 2009b]	image size: 1920 × 1080, max filter size: 30, N entries operated at the same time
MAT-HWA(M) [Lee et al. 2009b]	3,000 interesting points (64 dimension) per image, matching M images
RSZ-HWA, DET-HWA [Itseez 2015]	image size: 1920 × 1080, scale factor : 1.1, 24 × 24 window

	Configuration
Config-A	IMG-HWA × 2, HES-HWA(32), MAT-HWA(30)
Config-B	HES-HWA(32), MAT-HWA(20), RSZ-HWA, DET-HWA

access traces. The size of memory requests from HWAs is 64 bytes, and the number of outstanding memory requests from each HWA is at most 16.

6.4. System with a GPU

We also evaluate CPU-GPU and CPU-GPU-HWA systems with our in-house cycle level simulator. The specification of the GPU we model is 800MHz, 20 cores, and 1,600 operations/cycle, which is similar to the AMD Radeon 5870 specification [Advanced Micro Devices 2009]. 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 six GPU traces from a GPU benchmark (3Dmark) and five games, with a proprietary GPU simulator [Ausavarungnirun et al. 2012]. The target frame rate of all GPU benchmarks is 30 fps. We set the GPU benchmarks' deadline to 33.3msec (=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.

6.5. Performance and Fairness Metrics

We measure CPU performance with the commonly used *Weighted Speedup* metric [Eyerman and Eeckhout 2008; Snavely and Tullsen 2000]. We measure fairness using the *Maximum Slowdown* metric [Vandierendonck and Seznec 2011; Kim et al. 2010a, 2010b; Das et al. 2009, 2010; Subramanian et al. 2013, 2015b, 2014; Wang et al. 2015; Zhao et al. 2014; Seshadri et al. 2014]. 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 the frame rate accordingly).

6.6. Parameters of the Evaluated Schedulers

Unless otherwise stated, for DASH, we use a *SchedulingUnit* of 1,000 CPU cycles and a *SwitchingUnit* of 500 CPU cycles. For TCM [Kim et al. 2010b], we use a *ClusterFactor* of 0.15, a shuffling interval of 800 cycles, and *QuantumLength* of 1M cycles.

7. EVALUATION

We compare DASH with previously proposed schedulers: (1) FRFCFS [Rixner et al. 2000; Zuravleff and Robinson 1997] and (2) TCM, both with static priority (FRFCFS-St and TCM-St), where the HWA always has higher priority than *all* CPU cores, and (3) FRFCFS with *Dyn-Prio* (FRFCFS-Dyn), which employs the dynamic priority mechanism of Jeong et al. [2012a]. 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 IV 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 the target frame rate) are achieved. For DASH, we use an *EmergentThreshold* value of 0.8 for all HWAs.

Figure 5 shows the average CPU performance (in terms of weighted speedup) across all 80 workloads, using both Config-A and B. Table V shows the deadline-met ratio and frame rate of four types of HWAs.

We make 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 achieves either 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 this comes at the cost of reduced CPU performance. Third, DASH achieves comparable performance to FRFCFS-Dyn-0.9 and 9.5% better system performance than FRFCFS-DynOpt, while achieving a deadline-met ratio of 100%. We conclude that DASH achieves both high CPU performance and 100% deadline-met ratio for HWAs. In the next section, we present a performance breakdown of DASH.

7.1. Performance Breakdown of DASH

We break down the performance benefits from the different components of DASH. Figure 6 shows the system performance and maximum slowdown normalized to FRFCFS-DynOpt. The x-axis shows workload memory intensity. The numbers above the bars of FRFCFS-DynOpt show the absolute values for FRFCFS-DynOpt. We compare four different configurations of DASH: (1) DA-D (*distributed priority* on top of TCM for CPU applications; Section 4.1), (2) DA-D+L (DA-D along with *application-aware prioritization* between LDP-HWAs and memory-intensive CPU applications; Section 4.1), (3) DA-D+L+S (DA-D+L along with *worst-case latency-based prioritization* for SDP-HWAs; Sections 4.2 and 4.3), and (4) DA-D+L+S+P (the complete DASH mechanism: DA-D+L+S along with *probabilistic prioritization* between LDP-HWAs and memory-intensive CPU applications; Section 4.4). Table VI shows the deadline-met ratio.

We draw five major conclusions. First, DA-D improves performance over FRFCFS-DynOpt by 9.2%. Because FRFCFS-DynOpt uses a single static value of *EmergentThreshold* to prioritize HWAs over CPU cores, it sometimes overprioritizes an HWA even when bandwidth requirements from CPU cores are low and the HWA is achieving

Table IV. EmergentThreshold for FRFCFS-Dyn

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

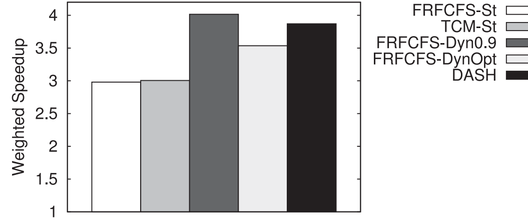

Fig. 5. CPU performance.

Table V. Deadline-Met Ratio and Frame Rate of HWAs

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

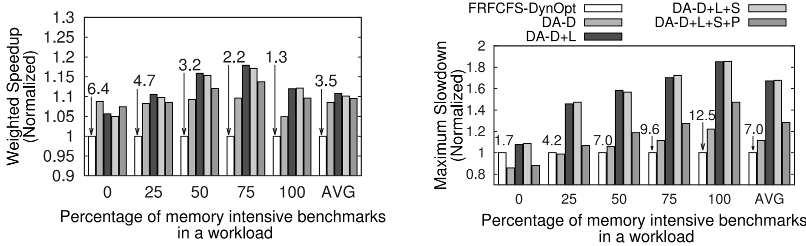

Fig. 6. DASH CPU performance and fairness breakdown for different workload memory intensities.

Table VI. Deadline-Met Ratio of HWAs for DASH Components

Scheduling Algorithms	Deadline-Met Ratio (%) / Frame Rate (fps)				
	IMG	HES	MAT	RSZ	DET
FRFCFS-DynOpt	100 / 30	100 / 30	99.997 / 29.72	100 / 30	99.99 / 25.5
DA-D	100 / 30	99.999 / 29.938	100 / 30	100 / 30	99.88 / 21.06
DA-D+L	100 / 30	99.999 / 29.969	100 / 30	100 / 30	99.87 / 20.44
DA-D+L+S	100 / 30	100 / 30	100 / 30	100 / 30	100 / 30
DA-D+L+S+P	100 / 30	100 / 30	100 / 30	100 / 30	100 / 30

its target progress. In contrast, since DA-D dynamically allocates priority to an HWA over CPU cores based on the HWA's progress, DA-D *avoids overprioritizing the HWA* and thus improves performance over FRFCFS-DynOpt. However, this improvement comes at the cost of missed deadlines for HES and DET-HWA, as shown in Table VI.

Second, application-aware prioritization between LDP-HWAs and memory-intensive CPU applications (DA-D+L) improves performance, especially as the memory intensity increases (7.6% maximum over DA-D). This is because prioritizing HWAs over memory-intensive applications reduces the amount of time HWAs become urgent and interfere with memory-nonintensive CPU applications. Since memory-nonintensive CPU applications are latency sensitive and memory-intensive CPU applications are not as memory latency sensitive, application-aware prioritization mechanisms of DASH (which distinguish between the needs of such different applications) improve total system performance over FRFCFS-DynOpt, which is *not aware of* the memory access characteristics of different applications. However, the SDP-HWAs (HES and DET) still miss some deadlines.

Third, DA-D+L+S enables such HWAs to meet their deadlines, while still achieving high CPU performance by employing worst-case access latency-based prioritization for SDP-HWAs, However, memory-intensive applications still experience high slowdowns with DA-D+L+S.

Fourth, DA-D+L+S+P tackles this problem by probabilistically changing the prioritization order between memory-intensive applications and LDP-HWAs. This increases the bandwidth allocation for memory-intensive applications. The result is a 24% reduction in maximum slowdown, while degrading performance by only 0.6% compared to DA-D+L+S and at the same time meeting *all* HWAs' deadlines. We analyze the differences between DA-D+L+S and DA-D+L+S+P in detail. When we compare DA-D+L+S and DA-D+L+S+P at each memory intensity, DA-D+L+S+P loses some performance (by up to 3%) against DA-D+L+S at high memory intensity workloads (50%–100%) while reducing maximum slowdown significantly (by up to 26%). Probabilistic prioritization of DA-D+L+S+P allocates lower priority to HWAs than memory-intensive CPU applications, when HWAs are making sufficient progress. This probabilistic priority allocation can sometimes not adapt to sudden bandwidth increases from CPU cores, since it adjusts the probability step by step at the end of each epoch. Hence, the controller allocates HWAs lower priority more frequently than ideal. During such times when the HWAs have low priority, their memory requests are almost completely prevented from being serviced due to contention from workloads with high memory intensity. As a result, HWAs become urgent more often, thereby degrading the performance of memory-nonintensive CPU applications. This is why DA-D+L+S+P *sometimes* degrades performance at high memory intensity. Therefore, if the focus is *only* on high performance, DA-D+L+S is the best scheduler, especially for memory-intensive workloads. On the other hand, if the goal is to achieve a *good balance* between performance and fairness, DA-D+L+S+P is a better choice, since it achieves *both* high performance *and* high fairness. Note that this is not a major concern at low memory intensities, since an HWA with low priority can still get requests served and achieve its target progress due to low contention. Therefore, the performance penalty for CPU applications is small (for instance, in workloads with 25% memory-intensive applications) and DA-D+L+S+P achieves even better performance than DA-D+L+S at the lowest memory intensity.

Finally, although DASH improves performance especially as the memory intensity increases, its performance improvement peaks at the 75% mark (i.e., when 75% of the CPU applications in a workload are memory intensive). This is because DASH prioritizes HWAs over memory-intensive CPU applications and thereby reduces interference caused by HWAs to memory-nonintensive CPU applications, which are especially vulnerable to interference. DASH enables better performance for the relatively memory-nonintensive CPU applications, even when the fraction of memory-intensive benchmarks is 100%, by virtue of prioritizing HWAs over memory-intensive CPU applications. However, such relatively memory-nonintensive CPU applications are less vulnerable/sensitive to memory latency. Hence, the performance improvement

of such applications (from prioritizing HWAs over memory-intensive CPU applications) is small. As a result, performance improvement of DASH peaks when the fraction of memory-intensive CPU applications is 75%.

We conclude that DASH is effective at achieving high CPU performance, achieving good fairness, *and* meeting HWAs' deadlines.

7.2. Impact of *EmergentThreshold*

We study the impact of *EmergentThreshold* on performance and deadline-met ratio and the tradeoffs it enables. Figure 7 shows the average system performance with FRFCFS-Dyn and DASH when sweeping *EmergentThreshold* across all 80 workloads using Config-A and B. We employ the same *EmergentThreshold* value for all HWAs. Tables VII and VIII show HWAs' deadline-met ratios.

We draw two major conclusions. First, for both FRFCFS-Dyn and DASH, there is a tradeoff between system performance and HWA deadline-met ratio, as the *EmergentThreshold* is varied. As the *EmergentThreshold* increases, CPU performance improves at the cost of decreases in deadline-met ratio. Second, for a given value of *EmergentThreshold*, DASH achieves a significantly higher deadline-met ratio than FRFCFS-Dyn, while achieving similar CPU performance, because of *distributed priority* and *application-aware scheduling* mechanisms. Specifically, DASH is able to meet *all* deadlines with an *EmergentThreshold* of 0.8 for Config-A, whereas FRFCFS-Dyn needs an *EmergentThreshold* of 0.1 to meet *all* deadlines. DASH-0.8 achieves 22.7% higher performance than FRFCFS-Dyn-0.1. Because FRFCFS-Dyn prioritizes an HWA over CPU cores only when the HWA is close to a deadline, FRFCFS-Dyn needs a low *EmergentThreshold* to meet all deadlines. On the other hand, because DASH dynamically allocates higher priority to an LDP-HWA based on its progress, DASH gives the LDP-HWA consistent bandwidth, despite fluctuations in other agents' bandwidth usage. Application-aware scheduling for HWAs, which is worst-case latency-based prioritization for SDP-HWAs, also enables short latencies for the SDP-HWA's requests, enabling them to meet their deadlines. Therefore, DASH is able to meet *all* deadlines with a high *EmergentThreshold* value. Based on these observations, we conclude that DASH is effective at 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-nonintensive CPU applications. Figure 8 shows average CPU performance and fairness with FRFCFS-DynOpt and DASH across 80 workloads using Config-A. For DASH, we sweep the *ClusterFactor* from 0 to 1.0. All HWAs' deadlines are met for all values of the *ClusterFactor* for DASH.

We draw three major conclusions. First, there is a tradeoff 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 nonintensive 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, if the focus were *only* on performance, we would pick a *ClusterFactor* of 0.3. On the other hand, if the focus were *only* on fairness, we would pick a *ClusterFactor* of 0. In our main evaluations, we focus on identifying/evaluating a high-performance and well-balanced design. Hence, we pick a *ClusterFactor* of 0.15. Alternatively, a *ClusterFactor* of 0.1 would result in a fairer, yet also well-balanced system, which improves performance by 14%, compared to FRFCFS-DynOpt, while degrading fairness by only 3.8%. This result also shows that delaying memory-intensive applications does not significantly

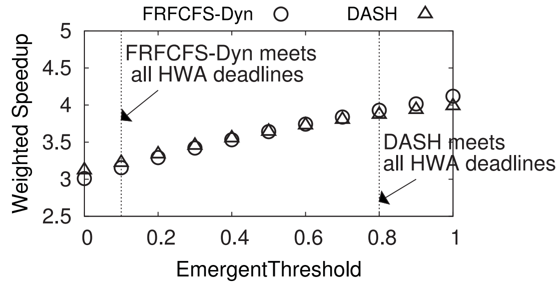

Fig. 7. CPU performance sensitivity to *EmergentThreshold*.

Table VII. Deadline-Met Ratio of FRFCFS-Dyn

Emergent Threshold	Deadline-Met Ratio (%)					
	Config-A		Config-B			
	HES	MAT	HES	MAT	RSZ	DET
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	100	99.997
0.6	99.905	44.691	99.989	94.697	100	99.960
0.7	99.875	38.097	99.957	86.366	100	99.733
0.8	99.831	34.098	99.906	74.690	99.886	99.004
0.9	99.487	31.385	99.319	60.641	97.977	97.149
1	96.653	27.320	95.798	33.449	55.773	88.425

Table VIII. Deadline-Met Ratio of DASH

Emergent Threshold	Deadline-Met Ratio (%)					
	Config-A		Config-B			
	HES	MAT	HES	MAT	RSZ	DET
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	95.93	100

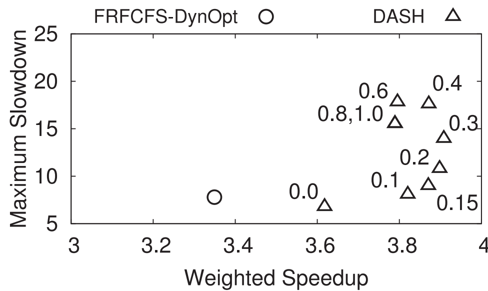

Fig. 8. Performance sensitivity to *ClusterFactor*.

affect their performance. Third, regardless of the *ClusterFactor*, DASH is able to meet all HWAs' deadlines, since it 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 two MAT-HWAs and two HES-HWAs. We vary the memory intensity of the HWAs by varying their parameters in Table III (we do not show these plots due to space constraints). As the HWAs' memory intensity increases, DASH provides higher performance improvement for the CPU (by 28.3% maximum) while meeting almost all HWA 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. DASH is effective in mitigating this interference while respecting HWAs' deadlines.

7.5. Evaluation on Systems with GPUs

Figure 9 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 IV for the other HWAs. For DASH, we use a *ClusterFactor* of 0.2 and an *EmergentThreshold* of 0.9 for the GPU and other HWAs.

DASH achieves 10.1% higher CPU performance than FRFCFS-Dyn, while also achieving a higher frame rate for MAT-HWA than FRFCFS-Dyn. DASH'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. DASH improves CPU performance by 2% over FRFCFS-Dyn, while meeting all deadlines, whereas FRFCFS-Dyn misses a deadline. We conclude that DASH 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 Memory Channels. Figure 10 (left) shows the CPU performance with a different number of channels across 25 workloads (executing 90M cycles) using HWA Config-A (other parameters are the same as baseline). All HWAs meet all deadlines with all schedulers as there is enough bandwidth. The key conclusion is that as the number of channels decreases, memory contention increases, resulting in higher performance benefits from DASH. Even at eight memory channels, where memory bandwidth is ample, DASH significantly improves CPU performance.

7.6.2. Number of Agents (Cores and HWAs). Figures 10 (right) and 11 show the same performance metrics for the same schedulers as in Section 7.6.1 when using a different number of CPU cores (from eight to 24) and HWAs[5] (four or eight). We draw three conclusions. First, DASH *always* improves CPU performance over FRFCFS-DynOpt. Second, as the number of agents increases, there is more memory contention, providing more opportunity for DASH, which achieves greater performance improvement (24.0% maximum). Finally, DASH meets all deadlines for all HWAs. FRFCFS-DynOpt is particularly ineffective at meeting deadlines at high core and accelerator counts. We conclude that DASH is scalable to high core and HWA counts.

7.6.3. Scheduling Unit *and* Switching Unit*.* Figure 12 shows average system performance and maximum slowdown when we sweep the *SchedulingUnit* (Section 4.1) from 1,000 to 5,000 cycles and *SwitchingUnit* (Section 4.4) from 500 to 2,000 cycles (*SwitchingUnit* < *SchedulingUnit*). We observe two trends. First, as the *SchedulingUnit* increases, system performance decreases because once an HWA is deemed as urgent, it interferes with CPU cores for a longer time (and it stays as urgent until at least the end of the current *SchedulingUnit*). Second, a smaller *SwitchingUnit* provides better fairness, since fine-grained switching of the probability *Pb* enables memory-intensive applications to not be deprioritized for long periods of time at a stretch. Based on these observations, we empirically pick a *SchedulingUnit* of 1,000 cycles and *SwitchingUnit* of 500 cycles.

8. RELATED WORK

Memory scheduling. We have already compared DASH qualitatively and quantitatively to the state-of-the-art QoS-aware scheduler for CPU-GPU systems, proposed by Jeong et al. [2012a]. When this scheduler is adapted to a CPU-HWA context, DASH outperforms it in terms of both system performance and deadline-met ratio.

[5]The 4-HWA configuration is the same as Config-A. The 8-HWA configuration contains 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.

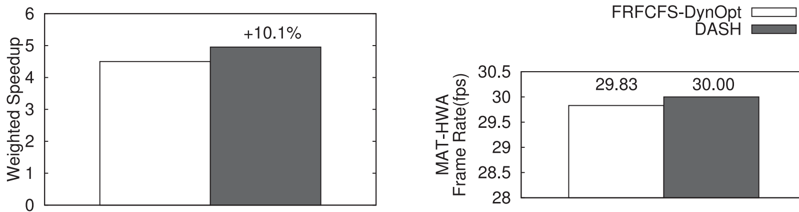

Fig. 9. CPU performance and HWA frame rate on a CPU-GPU-HWA system.

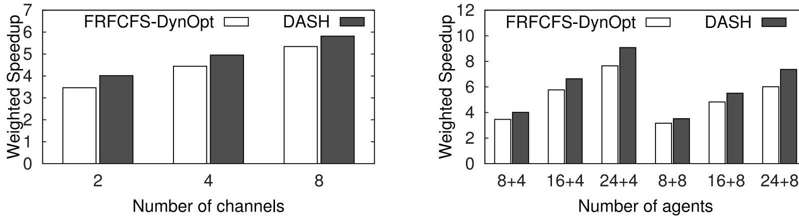

Fig. 10. Performance sensitivity to system parameters.

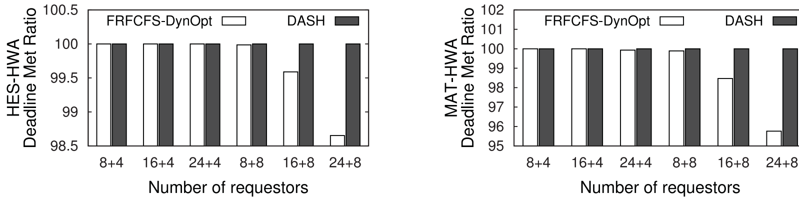

Fig. 11. Deadline-met ratio sensitivity to core count.

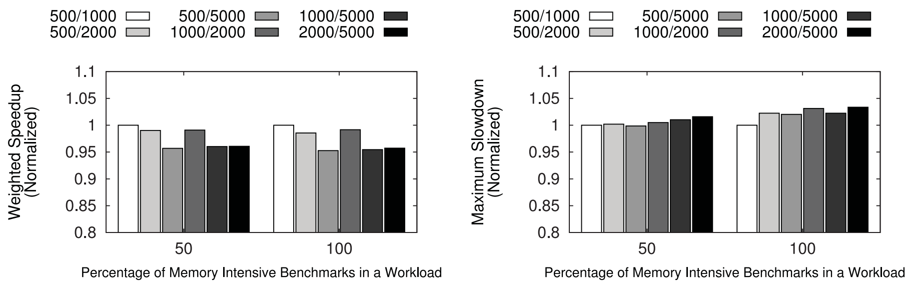

Fig. 12. CPU performance sensitivity to *Scheduling Unit* and *Switching Unit*. Legend indicates A/B where A is *Scheduling Unit* and B is *Switching Unit*.

Ausavarungnirun et al. [2012] propose Staged Memory Scheduling (SMS) to improve system performance and fairness in a CPU-GPU system. Unlike DASH, SMS does not explicitly attempt to provide QoS to the GPU and aims to optimize overall performance and fairness.

Most other previously proposed memory schedulers (e.g., Rixner et al. [2000], Mutlu and Moscibroda [2007, 2008], Nesbit et al. [2006], Kim et al. [2010a, 2010b], Subramanian et al. [2013], Hur and Lin [2004], Subramanian et al. [2015b, 2014], Moscibroda and Mutlu [2008], Muralidhara et al. [2011], Zhao et al. [2014], Das et al. [2013], Ebrahimi et al. [2011], Lee et al. [2008], Subramanian et al. [2015a], Lee et al. [2010], and Ipek et al. [2008]) have been designed to improve 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, DASH is specifically designed to provide high system performance and QoS in heterogeneous systems with CPU cores and HWAs.

Lee et al. [2005] propose a 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, which might prevent bandwidth-sensitive requestors from meeting potential deadline requirements. Other previous works [Akesson et al. 2007; Reineke et al. 2011; Paolieri et al. 2009; Wu and Zhang 2013; Macian et al. 2003; Kim et al. 2014] have proposed to build memory controllers that provide support to guarantee real-time access latency constraints for each master. The PRET DRAM Controller [Reineke et al. 2011] partitions DRAM into multiple resources that are accessed in a periodic pipelined fashion. Wu and Zhang [2013] propose to strictly prioritize real-time threads over non-real-time threads. Macian et al. [2003] bound the maximum latency by scheduling in a round-robin manner. Other works [Akesson et al. 2007; Paolieri et al. 2009] group a set of accesses to all banks and schedule requests at the group granularity. All these works aim to bound the worst-case latency by scheduling requests in a fixed, predictable order. They also do not consider CPU performance. As a result, they waste a significant amount of memory bandwidth and do not achieve high system performance.

Source throttling. Memguard [Heechul et al. 2013] guarantees worst-case bandwidth to each core by regulating the number of injected requests from each core. Other works [Ebrahimi et al. 2010; Nychis et al. 2012; Chang et al. 2012; Nychis et al. 2010; Cheng et al. 2010] propose to throttle the request injection rate at the cores to improve fairness and performance in CPU-only systems. Still other previous works [Stevens 2010; Jeong et al. 2012a; Kayiran et al. 2014] propose to throttle the number of GPU requests in CPU-GPU systems to mitigate interference to CPU applications. These schemes are complementary to our approach and can be employed in conjunction with DASH to achieve better interference mitigation.

Data interleaving. The Heterogeneous Memory Controller [Nachiappan et al. 2014] divides the address space into two regions with different address interleaving policies. Each region is associated with different kinds of agents (e.g., CPUs, HWAs) that would benefit from the specific interleaving policy. DASH and such a heterogeneous interleaving approach can be employed in conjunction with each other to achieve better performance and deadline-met ratios.

Memory controller/channel/bank partitioning. To mitigate interference, previous works [Muralidhara et al. 2011; Jeong et al. 2012b; Liu et al. 2012b; Das et al. 2013] propose to map data of interfering applications to different channels or banks. Our memory scheduling approach can be combined with such partitioning approaches to achieve higher system performance and QoS for HWAs.

Virtual time scheduling. The rate-based approach of DASH, that is, prioritization of HWAs based on comparison between the target progress and the expected progress, is similar to a virtual time scheduler for CPU scheduling [Nieh and Lam 1997; Chandra et al. 2000; Duda and Cheriton 1999; Nieh et al. 2001; Goyal et al. 1996]. These schedulers assign a weight to each application and allocate resources proportional to this weight, based on the concept of lottery scheduling [Waldspurger and Weihl 1994]. They

schedule a CPU application using its virtual time, which is updated by the amount of processing time divided by the weight when the application is scheduled. Among such virtual time schedulers, SMART [Nieh and Lam 1997] and the BVT scheduler [Duda and Cheriton 1999] are applicable to real-time systems. Goyal et al. [1996] propose a hierarchical CPU scheduler, which partitions bandwidth among CPU application classes and employs a different CPU scheduler for each class. A virtual time scheduler is used to partition bandwidth between different classes. For multimedia systems, each application is classified as hard real time, soft real time, or best effort. The idea of classifying applications into groups and using different scheduling policies has similarities to DASH. However, the key distinction between these virtual time schedulers and DASH is that (1) the constraints DASH has as a memory scheduler are very different from the constraints that these software schedulers have (e.g., DASH requires awareness of memory-level locality and bank-level parallelism, is subject to hardware cost and complexity constraints, and operates at finer time scales), and (2) these past schedulers are *not* aware of the memory access characteristics of the different applications, which is a key characteristic of DASH that enables high performance.

9. CONCLUSION

We introduce a deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators, DASH, with the goal of enabling hardware accelerators (HWAs) to meet their deadlines while achieving high CPU performance. Our evaluations across a wide variety of workloads/systems show that DASH meets HWAs' deadlines while also greatly improving CPU performance, compared to state-of-the-art techniques. We conclude that DASH is 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.

ACKNOWLEDGMENTS

We thank anonymous reviewers of ACM TACO as well as anonymous reviewers of MICRO 2015, ISCA 2015, HPCA 2015, and MICRO 2014, who reviewed previous versions of this work. We acknowledge members of the SAFARI research group for their feedback. An earlier, nonrefereed version of this work appears in arXiv.org [Usui et al. 2015].

REFERENCES

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

Advanced Micro Devices. 2009. AMD Radeon HD 5870 Graphics. Retrieved July 30, 2015, from http://www.amd.com/en-us/products/graphics/desktop/5000/5870#.

B. Akesson, K. Goossens, and M. Ringhofer. 2007. Predator: A predictable SDRAM memory controller. In *CODES+ISSS*.

R. Ausavarungnirun, K. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. 2012. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *ISCA*.

H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool. 2008. SURF: Speeded up robust features. In *CVIU*.

A. Chandra, M. Adler, P. Goyal, and P. Shenoy. 2000. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *OSDI*.

N. Chandramoorthy, G. Tagliavini, K. Irick, A. Pullini, S. Advani, S. Al Habsi, M. Cotter, J. Sampson, V. Narayanan, and L. Benini. 2015. Exploring architectural heterogeneity in intelligent vision systems. In *HPCA*.

K. Chang, R. Ausavarungnirun, C. Fallin, and O. Mutlu. 2012. HAT: Heterogeneous adaptive throttling for on-chip networks. In *SBAC-PAD*.

K. Chang, D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. 2014. Improving DRAM performance by parallelizing refreshes with accesses. In *HPCA*.

H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang. 2010. Memory latency reduction via thread throttling. In *MICRO*.

CMU SAFARI Research Group. 2015a. Ramulator. (2015). Retrieved October 29, 2015, from https://github.com/CMU-SAFARI/ramulator.

CMU SAFARI Research Group. 2015b. SAFARI GitHub. (2015). Retrieved November 9, 2015, from https://github.com/CMU-SAFARI.

R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *HPCA*.

R. Das, O. Mutlu, T. Moscibroda, and C. Das. 2009. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*. 280–291.

R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. 2010. AéRgia: Exploiting packet latency slack in on-chip networks. In *ISCA*.

K. J. Duda and D. R. Cheriton. 1999. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *SOSP*.

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

E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. 2011. Parallel application memory scheduling. In *MICRO*.

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

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

P. Goyal, X. Guo, and H. M. Vin. 1996. A hierarchical CPU scheduler for multimedia operating systems. In *OSDI*.

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

F.-C. Huang, S.-Y. Huang, J.-W. Ker, and Y.-C. Chen. 2012. High-performance SIFT hardware accelerator for real-time image feature extraction. *IEEE Transactions on Circuits and Systems for Video Technology,* 22, 3 (March 2012).

I. Hur and C. Lin. 2004. Adaptive history-based memory schedulers. In *MICRO*.

E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*.

Itseez. 2015. Open Source Computer Vision. (2015). Retrieved July 30, 2015, from http://opencv.org.

JEDEC. 2010. Standard No. 79-3. DDR3 SDRAM STANDARD. (2010).

M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. 2012a. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *DAC-49*.

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

O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. 2014. Managing GPU concurrency in heterogeneous architectures. In *MICRO*.

S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu. 2014. The efficacy of error mitigation techniques for DRAM retention failures: A comparative experimental study. In *SIGMETRICS*.

H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *RTAS*.

W. Kim, H. Chung, H.-D. Cho, and Y. Kim. 2012. Enjoy the ultimate WQXGA solution with Exynos 5 Dual. *Samsung Electronics White Paper* (2012).

Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. 2010a. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*.

Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. 2010b. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*.

Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. 2012. A case for exploiting subarray-level parallelism (SALP) in DRAM. In *ISCA*.

Y. Kim, W. Yang, and O. Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE CAL* PP, 99 (2015).

C. J. Lee, O. Mutlu, V. Narasiman, and Y. Patt. 2008. Prefetch-aware DRAM controllers. In *MICRO*.

C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt. 2010. DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems. HPS Technical Report, TR-HPS-2010-002. (2010).

C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. 2009a. Improving memory bank-level parallelism in the presence of prefetching. In *MICRO*.

D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. 2015. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. In *HPCA*.

D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. 2013. Tiered-latency DRAM: A low latency and low cost DRAM architecture. In *HPCA*.

K.-B. Lee, T.-C. Lin, and C.-W. Jen. 2005. An efficient quality-aware memory controller for multimedia platform SoC. *IEEE Transactions on Circuits and Systems for Video Technology* 15, 5 (May 2005).

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

J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. 2012a. RAIDR: Retention-aware intelligent DRAM refresh. In *ISCA*.

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

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

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

Micron. 2014. 1Gb: x4, x8, x16 DDR3 SDRAM Features. Retrieved July 30, 2015, from http://www.micron.com/~/media/Documents/products/data-sheet/dram/ddr3/1gb_ddr3_sdram.pdf.

T. Moscibroda and O. Mutlu. 2008. Distributed order scheduling and its application to multi-core dram controllers. In *PODC*.

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

O. Mutlu and T. Moscibroda. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*.

O. Mutlu and T. Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*.

N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. 2014. GemDroid: A framework to evaluate mobile platforms. In *SIGMETRICS*.

NASA. 2012. NAS Parallel Benchmark Suite. Retrieved July 30, 2015, from http://www.nas.nasa.gov/publications/npb.html.

K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. 2006. Fair queuing memory systems. In *MICRO*.

J. Nieh and M. S. Lam. 1997. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *SOSP*.

J. Nieh, C. Vaill, and H. Zhong. 2001. Virtual-time round-robin: An O(1) proportional share scheduler. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*.

G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu. 2010. Next generation on-chip networks: What kind of congestion control do we need? In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HOTNETS'10)*.

G. P. Nychis, C. Fallin, T. Moscibroda, O. Mutlu, and S. Seshan. 2012. On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects. In *SIGCOMM*.

M. Paolieri, E. Quiones, F. Cazorla, and M. Valero. 2009. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded Systems Letters* 1, 4 (December 2009).

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

Qualcomm. 2011. Snapdragon S4 processors: System on chip solutions for a new mobile age. *Qualcomm White Paper* (2011).

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

S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. 2000. Memory access scheduling. In *ISCA*.

I. Schmadecke and H. Blume. 2013. Hardware-accelerator design for energy-efficient acoustic feature extraction. In *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE'13)*.

V. Seshadri, A. Bhowmick, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry. 2014. The dirty-block index. In *ISCA*.

V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2013. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *MICRO*.

V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2015. Gather-scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses. In *MICRO*.

A. Snavely and D. M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*.

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

Standard Performance Evaluation Corporation. 2014. SPEC CPU2006. Retrieved July 30, 2015, from http://www.spec.org/spec2006.

G. P. Stein, I. Gat, and G. Hayon. 2008. Challenges and solutions for bundling multiple DAS applications on a single hardware platform. In *V.I.S.I.O.N.*

A. Stevens. 2010. QoS for high-performance and power-efficient HD multimedia. *ARM White Paper* (2010).

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

L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. 2015a. The blacklisting memory scheduler: Balancing performance, fairness and complexity. *CoRR* abs/1504.00390 (2015).

L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. 2015b. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *MICRO*.

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

J. Tanabe, S. Toru, Y. Yamada, T. Watanabe, M. Okumura, M. Nishiyama, T. Nomura, K. Oma, N. Sato, M. Banno, H. Hayashi, and T. Miyamori. 2015. A 1.9TOPS and 564GOPS/W heterogeneous multicore SoC with color-based object classification accelerator for image-recognition applications. In *ISSCC*.

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

TPC. 2015. TPC Benchmarks. Retrieved July 30, 2015, from http://www.tpc.org/.

H. Usui, L. Subramanian, K. Chang, and O. Mutlu. 2015. SQUASH: Simple QoS-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *CoRR* abs/1505.07502 (2015).

H. Vandierendonck and A. Seznec. 2011. Fairness metrics for multi-threaded processors. *IEEE CAL* 10, 1 (February 2011).

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

C. A. Waldspurger and W. E. Weihl. 1994. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*.

H. Wang, C. Isci, L. Subramanian, J. Choi, D. Qian, and O. Mutlu. 2015. A-DRM: Architecture-aware distributed resource management of virtualized clusters. In *VEE*.

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

P. Yedlapalli, N. Nachiappan, N. Soundararajan, A. Sivasubramaniam, M. Kandemir, and C. Das. 2014. Short-Circuiting Memory Traffic in Handheld Platforms. In *MICRO*.

J. Zhao, O. Mutlu, and Y. Xie. 2014. FIRM: Fair and high-performance memory control for persistent memory systems. In *MICRO*.

W. K. Zuravleff and T. Robinson. 1997. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. U.S. Patent Number 5,630,096. (1997).

Received August 2015; revised October 2015; accepted October 2015