A Self-aware Resource Management Framework for Heterogeneous Multicore SoCs with Diverse QoS Targets

YANG SONG, University of California San Diego
OLIVIER ALAVOINE, Qualcomm Inc.
BILL LIN, University of California San Diego

In modern heterogeneous MPSoCs, the management of shared memory resources is crucial in delivering end-to-end QoS. Previous frameworks have either focused on singular QoS targets or the allocation of partitionable resources among CPU applications at relatively slow timescales. However, heterogeneous MPSoCs typically require instant response from the memory system where most resources cannot be partitioned. Moreover, the health of different cores in a heterogeneous MPSoC is often measured by diverse performance objectives. In this work, we propose the Self-Aware Resource Allocation framework for heterogeneous MPSoCs. Priority-based adaptation allows cores to use different target performance and self-monitor their own intrinsic health. In response, the system allocates non-partitionable resources based on priorities. The proposed framework meets a diverse range of QoS demands from heterogeneous cores. Moreover, we present a runtime scheme to configure priority-based adaptation so that distinct sensitivities of heterogeneous QoS targets with respect to memory allocation can be accommodated. In addition, the priority of best-effort cores can also be regulated.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**; *Multicore architectures*;

Additional Key Words and Phrases: Memory system, resource allocation, quality-of-service, memory efficiency

ACM Reference format:
Yang Song, Olivier Alavoine, and Bill Lin. 2019. A Self-aware Resource Management Framework for Heterogeneous Multicore SoCs with Diverse QoS Targets. *ACM Trans. Archit. Code Optim.* 16, 2, Article 16 (April 2019), 23 pages.
https://doi.org/10.1145/3319804

This article is an extension of a conference paper: "SARA: Self-Aware Resource Allocation for Heterogeneous MPSoCs," published in DAC'18 (Song et al.). In particular, we extend our work in the following ways:
—We add a new section to discuss the challenges of priority-based adaptation in the SARA framework. To deal with the challenges, we introduce a two-stage runtime configuration solution, including distributed self-configuration and global regulation, to accommodate best-effort cores and real-time cores that are particularly sensitive to memory allocation updates.
—We expand the evaluation section to include more details on our simulation platform. We also present evaluation results on the runtime configuration scheme for the SARA framework.
—In addition, a more comprehensive review of related work is provided in this paper to summarize prior efforts on memory scheduling and management.

Authors' addresses: Y. Song and B. Lin, University of California San Diego, 9500 Gilman Dr, La Jolla, CA 92093, USA; emails: {y6song, billlin}@eng.ucsd.edu; O. Alavoine, Qualcomm Inc., San Diego, CA 92121, USA; email: oliviera@qti.qualcomm.com.

1544-3566/2019/04-ART16
https://doi.org/10.1145/3319804

1 INTRODUCTION

Modern heterogeneous MPSoCs [15, 17] have been widely deployed in mobile devices thanks to their energy efficiency. These MPSoCs typically integrate a diverse collection of cores. Figure 1 depicts an example of a heterogeneous MPSoC. Besides general-purpose cores like the CPU for running applications, most heterogeneous cores are dedicated to certain functions, such as the GPU, the DSP, and the display. These cores have diverse notions of Quality-of-Service (QoS). For example, the GPU measures target real-time performance in terms of frame rate, the DSP demands the memory latency to remain below a certain limit, and the display requires sufficient bandwidth to refresh frames at a constant rate.

To save cost and energy, heterogeneous cores commonly share resources, among which the sharing of the memory system including the network-on-chip (NoC) and the memory controller (MC) are the most challenging, because memory performance often has a direct and substantial impact on system performance. As data are being shared through memory, memory requests from different cores interfere with each other, and these memory interferences can cause the memory system to fail in meeting the target performance of some cores. Figure 2 depicts a camcorder application, which represents a typical use case in that it involves many cores at the same time. With ineffective memory scheduling, a real-time core (e.g., the display) may not achieve the target real-time performance due to inadequate memory bandwidth. Moreover, as latency-sensitive cores such as the DSP share memory with other cores, they can be easily overwhelmed by real-time cores consuming high bandwidth.

QoS-aware management for specific types of memory resources has been well studied by previous work [1, 4, 9, 22, 25]. In Reference [9], a QoS-aware scheduling policy was proposed for CPU-GPU systems. The concept of frame progress was introduced to monitor GPU performance. Although the policy can be extended to include more media cores, it cannot be applied to real-time cores whose target QoS cannot be assessed in terms of frame rate. Moreover, holistic memory management frameworks for CPU-centric homogeneous systems have also been explored recently [2, 5, 21]. This series of work typically constructs a management model based on the control theory to partition computing and memory resources. These frameworks accept flexible QoS targets as clients are allowed to define their own target performance. Nonetheless, such types of approaches are performed at a relatively slow timescale (e.g., on the order of milleseconds) due to the computational complexity. In comparison, real-time cores in heterogeneous MPSoCs often demand much more instant response from the memory system. Besides, communication between heterogeneous cores is mainly conducted through shared memory as shown in Figure 2, because multimedia data are generally too large to fit into caches. Therefore, DRAM plays a more crucial role in heterogeneous systems. However, previous frameworks cannot handle DRAM effectively, because its bandwidth is not partitionable. The memory access time at DRAM relies on many factors, such as row-buffer locality, bank-level parallelism, write-to-read turnaround times, and so on. It takes less time to serve a traffic stream with higher locality and parallelism and fewer write-to-read turnaround times. Without knowing memory access patterns of all traffic streams, any attempts to partition DRAM bandwidth by specific ratios or rations will be futile. What is worse, the memory access pattern of all traffic streams cannot be easily predicted by the memory subsystem.

So far, there has not been a QoS-aware resource management model for heterogeneous MPSoCs that is capable of allocating non-partitionable resources to fleeting QoS demands. In this work, we propose the Self-Aware Resource Allocation (SARA) framework as a solution. The contributions of our work can be summarized as follows.

- We propose a QoS-aware holistic resource management framework for heterogeneous systems. The SARA model accepts diverse notions of QoS and monitors performance distributively with lightweight meters to guarantee end-to-end QoS.

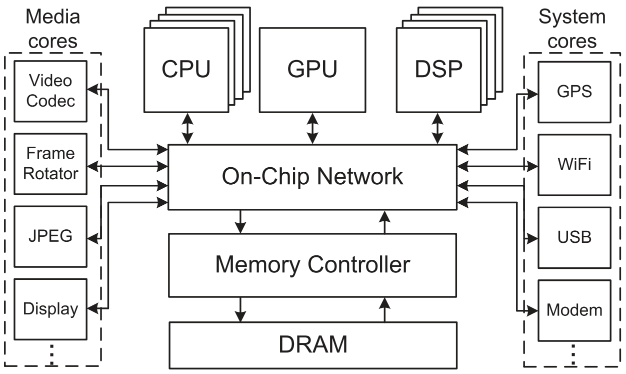


Fig. 1. Heterogeneous system architecture example.

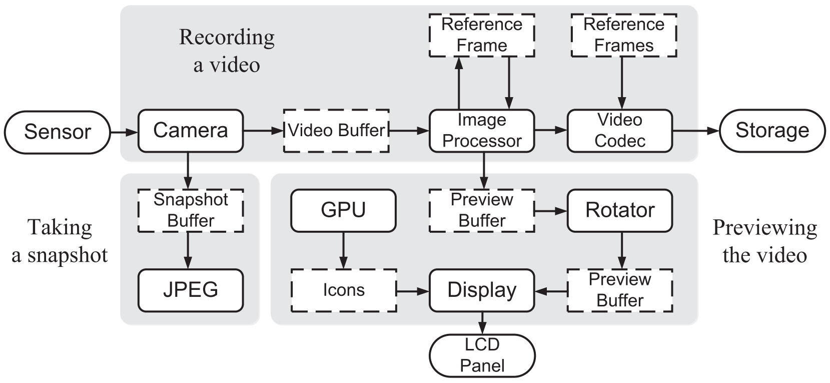


Fig. 2. A simplified dataflow of a camcorder application. Shared memory represented by boxes in dashed lines and cores by boxes in solid lines.

- We introduce priority-based self-adaptations for the management of non-partitionable resources. With properly tuned translation functions, the monitored performance is translated into priority levels indicating the healthiness of the real-time core. The memory subsystem further applies priority-based allocations in response.
- We present a runtime configuration scheme for priority-based adaptation. With runtime configuration, per-core translation from performance to priority level can be dynamically adjusted to accommodate failure-prone and best effort cores. This avoids the necessity to pre-tune translation functions and allows the proposed management framework to adapts to varying memory parameters and workloads.
- We evaluate the proposed framework using memory traffic of state-of-the-art MPSoCs and show that the proposed SARA model delivers target performance to all cores. In contrast, the performance of critical cores can fall below 10% of their targets without the SARA framework. Furthermore, the SARA framework is evaluated with runtime configuration of priority-based adaptation. Evaluation results show that runtime configuration successfully helps SARA framework to accommodate failure-prone and best-effort cores under declining memory frequency.

The rest of this article is organized as follows: Section 2 provides a comprehensive review on related work. Section 3 describes the proposed SARA framework. Section 4 presents a runtime configuration scheme for priority-based adaptation in the SARA framework. Experimental results and conclusions follow in Sections 5 and 6.

2 RELATED WORK

Memory Scheduling: In early studies on memory scheduling, improving DRAM throughput was the major focus. The First-Ready First Come First Serve (FR-FCFS) policy [19] was introduced to deliver high memory throughput by prioritizing requests that are going to open rows. Application-aware scheduling for CPU-only systems began to draw more attention later [7, 11, 12]. ATLAS [11] prioritizes applications with low memory intensity to improve system throughput. Thread cluster memory scheduling [12] dynamically clusters applications into low and high memory-intensity clusters and improves system performance as well as fairness. Dual-criticality memory controller [7] handles memory contention between real-time and high-performance applications by bank separation. Minimalist Open-page policy [10] adopts a fair DRAM address mapping scheme to increase bank level parallelism while avoiding row-buffer locality starvation. Parallelism-aware batch scheduling [14] batches requests in time order before prioritizing row-hit requests within each batch to circumvent starvation.

QoS-aware Memory Fabrics: Along with the emergence of heterogeneous SoCs, QoS-aware memory fabrics have been well studied in the recent decade. Staged memory scheduler [1] was the first application-aware scheduler proposed for CPU-GPU systems. The proposed scheduler architecture consists of separate queuing stages for memory transactions and DRAM commands. In between two stages a QoS-aware scheduler is deployed. The single-tier virtual queuing memory controller [22] was later proposed to overcome the inefficacy of two-tier controllers in QoS-aware scheduling. A single queuing stage is used to sort per-source traffic flows into per-bank virtual queues once they are received by transaction buffers. The QoS-aware scheduling is involved as the last step in the memory controller to avoid unwanted latency between the scheduler and DRAM. In Reference [9], a QoS-aware scheduling policy was proposed to dynamically balance CPU and GPU bandwidth. The proposed policy tracks GPU workload progress and allocates sufficient bandwidth to the GPU to achieve the target frame rate. Meanwhile best-effort service is provided to the CPU. Furthermore, a deadline-aware scheduler [23] extends the idea of frame progress tracking and takes into account more multimedia cores with target frame rates while improving CPU performance.

Besides memory scheduling, QoS-aware on-chip networks have been extensively investigated in recent years as well. Globally Synchronized Frames (GSF) [13] is one of the earliest approaches that adopts a frame-based scheduling scheme to provide throughput guarantees to network traffic. In GSF, time is coarsely quantized into frames, and each flow can reserve a fraction in the frame to inject data flits. Frames are prioritized according to their ages, and the flits belonging to older frames are chosen over those in younger frames. The limitation of GSF is twofold: The hardware complexity due to the recycling of frames and source buffering and the coarse granularity of global scheduling. To overcome the limitations of GSF, LOFT scheme [16] localizes the frame-based scheduling. In LOFT, each output link of a router manages frame recycles its own frames independently, which enables more flexibility and less hardware cost. Preemptive Virtual Clock (PVC) [4] takes a different approach to circumvent the issues in GSF. In PVC, frames are defined as a fixed number of cycles so that no frame recycling is needed. Moreover, each router maintains a table of bandwidth counters for network traffic flows. Rate-based scheduling is performed at each router in reference to the bandwidth counters. The OSCAR scheme [26] was recently proposed for CPU-GPU systems. To accommodate the vastly different traffic from the CPU and the GPU, the authors deployed asynchronous batch scheduling. Every router bundles packets into mini batches, inside which specific provisions are allocated to the CPU and the GPU. Packets from older batches are scheduled ahead of younger batches. Inside a batch, packets are reordered so that higher priority is given to CPU requests and read requests.

Memory Management Frameworks: Nonetheless, these works cannot guarantee end-to-end QoS, because they only deal with certain parts of the memory system. For example, the QoS provided in the memory controller could be deteriorated by the interconnect if it is not applying the same QoS policy. Moreover, implementing a centralized QoS monitor in the memory system can be prohibitive, since it needs to collect runtime information from all cores. More limiting, these works assume specific notions of QoS, which is not applicable to modern heterogeneous MPSoCs where the health of different cores is often evaluated by diverse performance objectives.

METE [21] is a multi-level framework for end-to-end resource management based on the control theory. It utilizes runtime information to predict application behaviors. Application controllers calculate the amounts of resources required to achieve target application performance. A global resource broker determines the final resource partitions for applications. SEEC [5] is a self-aware computing framework designed for a many-core processor. It follows the control loop of observe-decide-act (ODA) for resource allocations. Performance of CPU applications are observed by the decision engine that decides resource partitions using available actions defined by system designers. ARCC [2] is a self-computing framework implemented in the Tessellation many-core OS. It performs a two-level scheduling scheme: First, the resource allocation broker distributes global resources, and then at the user level scheduling policies are customized separately. CPSoC [20] is a class of sensor-actuator-rich MPSoCs following the observe-decide-act paradigm to enable self-awareness. With augmented on-chip and cross-layer sensing and actuation capabilities, it supports adaptation per layer and multiple cooperative ODA loops to achieve multi-objective goals. SPECTR [18] is a new management approach for many-core processors that leverages formal supervisory control theory to decompose complex control problems into sub-problems that are solved by local controllers. A survey has been made covering recent advancements in self-aware platforms [8].

Aforementioned frameworks are aimed at allocating partitionable resources, such as CPU cores and network bandwidth, to applications at software/OS level. They are not suitable for heterogeneous MPSoCs for the following reasons.

First, complicated control models may not be fast enough for heterogeneous cores, because real-time cores often have tight deadlines. For example, the DSP sets limit on memory latency at nanosecond level, but prior frameworks adapt through control theory computations at milleseconds timescales. Only a management framework implemented on the hardware level is able to deliver service that is sufficiently responsive.

More importantly, prior works assume that all memory resources can be partitioned using weights. However, the assumption does not apply to DRAM. In DRAM, data storage of a memory bank is organized into rows and columns. Only one row can be active in a bank. Columns in an open row can be accessed directly without extra operations. Whereas, to access a column in a closed row, the row where this column is located have to be loaded into the row-buffer (i.e., row activation operation) after all other rows have been closed (i.e., precharge operation) [6]. These row activation and precharge operations cause time penalty without contributing to actual data transfers, which makes DRAM access time unpredictable. For example, a traffic stream with higher row-buffer locality suffers from less time penalty and thus achieves lower access latency; a stream with higher bank-level parallelism avoids more precharge operations but accesses unengaged banks. Moreover, there is also time penalty for switching from a write request to a read request. Due to above physical limitations of DRAM, the deliverable bandwidth from DRAM cannot be partitioned without considering the erratic memory access patterns of all streams. To avoid the difficulty for weight-based partitioning approaches, we adopt a priority-based approach for resource allocation. The proposed framework will be detailed in the next section.

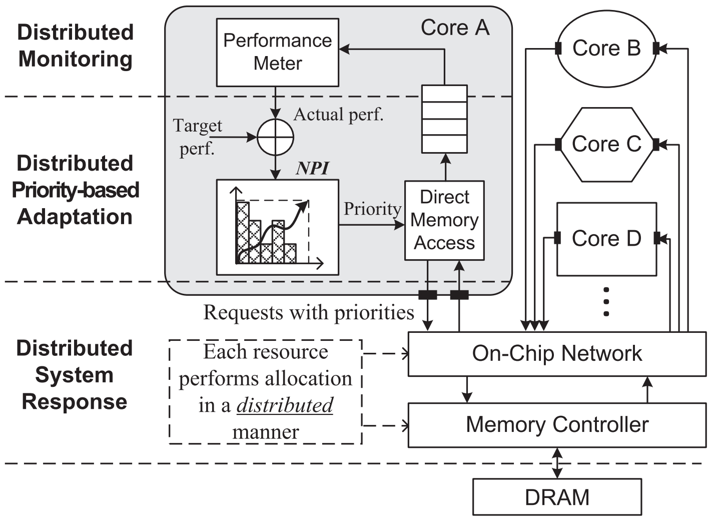


Fig. 3. The proposed SARA framework for heterogeneous MPSoCs. Each core self-monitors its performance and self-adapts its priority, and each resource performs priority-based allocation in a distributed manner.

3 SELF-AWARE RESOURCE ALLOCATION FRAMEWORK

The proposed architecture of SARA framework is shown in Figure 3. The resource management model consists of three stages, including distributed monitoring, priority-based runtime adaptation, and system response. Next, we will go through SARA framework stage by stage.

3.1 Distributed Self-Monitoring

In the first stage, each core self-monitors its own performance. Distributed monitoring relieves the memory system from the burden of monitoring cores with various notions of QoS. It is also good for scalability, since a new core can be added or modified without updating the rest of the system. In addition, the self-monitoring provides more accurate feedback on the end-to-end QoS compared with centralized monitoring in the memory system.

Every core customizes its own internal performance meter to measure its own performance or progression against a given target, and the measurement gets normalized into a fractional number called the Normalized Performance Indicator (NPI), which is used as an indicator of the core's intrinsic health. In the DSP, the performance meter monitors the average latency of its transactions, while in the display the meter measures the occupancy level of the read buffer. The deviation from the target performance (e.g., latency, occupancy level, etc.) produces the NPI metric. In our framework, each real-time Direct Memory Access (DMA) unit is equipped with a performance meter. Note that there are usually multiple DMAs in a single core. For simplicity, we only show one DMA per core in Figure 3.

3.2 Distributed Priority-based Adaptation

In the second stage, the NPI value delivered by the performance meter is translated into a relative priority level that is attached to memory transactions from the same DMA. As transactions travel to DRAM, their priority levels will be accessed and evaluated by arbiters in the on-chip network and the memory controller. Priority-based arbitrations allow the memory system to provide QoS without specifying the heterogeneous QoS for all cores and DMAs. Same with performance meters, the formulation of the NPI metric and the adaptations of priority can be implemented differently

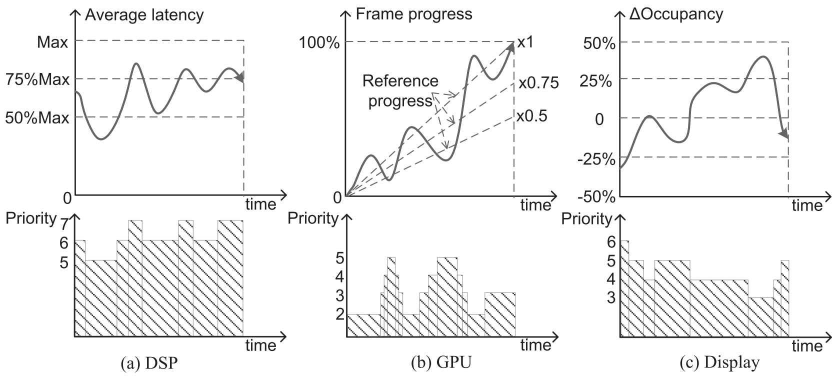


Fig. 4. Examples of priority-based adaptation in heterogeneous cores, including the DSP, the GPU, and the display.

from core to core, depending on the local target performance. Figure 4 shows three examples of priority-based adaptation in different cores.

As for the DSP, the target performance is to have the average memory latency lower than the maximum latency limit. The average latency is measured and compared with a pre-set limit to produce the NPI value (see Equation (1)), which remains above or equal to 1 when the target performance is achieved. This NPI value is then translated to a relative priority level (Figure 4(a)). The priority level increases along with average latency.

$$NPI_{DSP} = \frac{maximum\ latency\ limit}{average\ latency} \tag{1}$$

Similarly, cores requesting for bandwidth produce NPI metrics by computing the ratio between the average and the target bandwidth. However, frame rate differs from bandwidth, because frame size can be variable and thus a constant frame rate can lead to variable bandwidth. Hence frame progress [9, 22] is used instead to produce NPI metrics for frame rate–based cores. Take the GPU as an example, the target is to let the frame progress reach 100% as the current frame period comes to an end. The GPU's NPI value is produced at any time by comparing the frame progress with reference progresses that grow proportionally with frame time. The NPI value is then translated to a relative priority level of GPU transactions. Figure 4(b) shows the reference progresses achieving 1, 0.75, and 0.5 times the average data rate of target performance.

$$NPI_{GPU} = \frac{frame\ progress}{reference\ progress} \tag{2}$$

In the display, LCD panel reads data from a read buffer at a constant frame rate, while the display controller DMA tries to refill this buffer from DRAM so it never gets empty. Its *health* (see Equation (3)) relies on maintaining the refill rate (R_{refill}) no lower than the read data rate (R_{read}) and can be indicated by the variation of buffer occupancy level ($\Delta occupancy$). Compared with an initial level (e.g., 50%), the lower the occupancy level of this buffer gets, the worse the NPI value becomes, which is in turn translated to a higher priority level (Figure 4(c)).

$$NPI_{display} = \frac{R_{refill}}{R_{read}} = 1 + \frac{\Delta occupancy}{R_{read} \cdot time} \tag{3}$$

Intuitively, one might be concerned that every core would intentionally raise its priority to the maximum level to obtain as much resource as possible. Since a priority level is only maximized when the actual performance is far below the target, the pitfall can be avoided by properly setting performance targets. This should be done during design time with exhaustive profiling tests on realistic workloads while priority-based adaptation is enabled. In runtime, pre-determined performance targets are loaded into real-time cores depending on the workload. Note that performance targets of real-time cores should not be programmable, because they are limited by hardware capability.

In our evaluations, the priority levels are quantized into 2^k levels, which can be encoded with k bits. We found that $k = 3$ bits provides sufficient granularity in priority levels to produce satisfying results (i.e., the priority levels range from 0 to 7). The translation of NPI value into a priority level can be defined separately for each core, as long as lower NPI value is mapped to a higher priority level. An example of such an NPI translation function is shown in Equation (4), in which the priority is adjusted between 0 and 7 in an inverse proportion to NPI value. When NPI value equals or drops below 1, the priority is raised to the maximum level (7). However, if NPI value grows above 1, then the priority is gradually lowered to the minimal level (0).

$$\text{priority}(NPI) = \begin{cases} 7 & NPI \in (0,1) \\ Round\left(\frac{7}{NPI}\right) & NPI \in [1,14] \\ 0 & NPI \in (14,\infty) \end{cases} \tag{4}$$

Note that, instead of [0,7], the upper and lower bounds of priority range can be different from core to core to reflect their heterogeneity. Same with performance targets, priority ranges of real-time cores can be pre-tuned per workload during design time. Alternatively, an adaptive configuration scheme is needed to adjust priority ranges according to real-time workloads. This discussion will be continued in Section 4.

3.3 Distributed System Response

As transactions travel through the memory system, the system responds to QoS demands by providing resource management based on their priority levels. The priority-based management is performed correspondingly in different parts of the memory system. In on-chip network routers, transactions with higher priorities are preferentially selected during switch allocation. In the memory controller, when a priority-based scheduler arbitrates among transactions going to available memory banks, the ones with higher priorities have more chances to be served. An example of such memory scheduling policies is the priority-based round-robin shown in Policy 1. To avoid starvation of transactions with low priorities, the scheduler also needs to consider the aging factor during arbitration. In our evaluations, the scheduler periodically clears the backlog of transactions that have waited for at least T cycles (e.g., $T = 10{,}000$ cycles).

- **Policy 1:** Suppose P_A and P_B are priorities for transactions A and B, if $P_A > P_B$, then choose A; if $P_A < P_B$, then choose B; otherwise, choose between A and B in round-robin manner.

Priorities notify the system whether the cores are in urgent QoS demands. That gives the memory system an opportunity to optimize memory performance without undermining the QoS. Specifically, when transactions are in low urgency, the system can improve memory performance such as row-buffer hit rate, instead of focusing on serving QoS demands.

Row-buffer hits refer to multiple memory accesses to the same active row before it is precharged. Having more row-buffer hits means less time and power are wasted on row activation and precharge operations. Therefore, increasing row-buffer hits helps lower memory latency and improve DRAM total bandwidth.

To increase row-buffer hits, the memory controller re-orders transactions to favor the ones hitting open rows. It may cause degradations to the QoS when the transactions in high urgency are postponed due to row-buffer hits optimization. Yet, with priorities, the memory controller is aware of the urgency levels of transactions and able to avoid delaying urgent transactions during the optimization. Policy 2 shows an extension of Policy 1 to increase row-buffer hits without QoS degradations. The parameter δ is an adjustable threshold to balance row-buffer hits optimization and QoS-aware scheduling. When the priority level is lower than δ, the scheduler focuses on row-buffer hits; otherwise, the QoS comes first. A higher δ value gives more favor to DRAM bandwidth but also potentially causes more disturbance to the QoS. We found $\delta = 6$ a good setting to achieve high DRAM bandwidth without causing QoS degradations.

- **Policy 2:** Suppose transaction A is going to an active row-buffer and B is not. If $P_A, P_B < \delta$ or $P_A = P_B$, then choose A. Otherwise, perform priority-based round-robin.

Priority-based resource allocation handles non-partitionable with little computation in comparison with previous management models [2, 5, 21]. This facilitates instant response from the memory system to QoS demands.

3.4 Hardware Implementation

The implementation of the proposed SARA framework includes three parts: the computation of NPI value, the translation of NPI value to a priority level, and the priority-based arbitration in the memory system.

To calculate the NPI, a divider is needed at the performance meter for each DMA. For the translation of the NPI, a mapping function can be stored in a look-up table at each core. Each priority level is assigned with a table entry, and this entry stores the lowest NPI value allowed at that priority level. For example, if priority = p when $NPI \in [u, v)$, then the value u will be stored at the entry for p on the look-up table. Note that v will be the lower bound of the NPI for the priority level $p - 1$. Comparators are used to access table entries in parallel. If the given NPI value is not lower than the stored lower bound of NPI value, then the corresponding priority level will be asserted. When multiple priority levels are asserted, the lowest level will be chosen.

Supposed each priority level is encoded into three bits, a look-up table requires $2^3 = 8$ entries and each entry is a register for NPI value. A comparator is paired with each table entry. In total, the implementation only costs the storage of eight registers and eight comparators per core.

In the memory system, 3-bit comparators are required to perform priority-based arbitrations. Since most existing QoS-aware schedulers already provide hardware support for priorities, our framework can be integrated into the memory system without raising complexity.

4 RUNTIME CONFIGURATION OF PRIORITY-BASED ADAPTATION

The SARA framework provides a platform for heterogeneous cores to define and monitor their performance independently. In addition, the proposed framework also enables fast and consistent responses from memory fabrics through priority-based arbitrations. As the common currency within the framework, the priority attached to memory requests bridges the gaps between distinct notions of QoS and different types of memory resources.

Since the priority plays a central role in the SARA framework, priority-based adaptation is critical in the effectiveness of the framework. An exemplary function that translates given NPI value to a priority level has been shown in Equation (4). However, a single translation function may not be universally efficient with all cores due to their heterogeneous behaviors. An inefficient translation function may fail to raise the priority high enough to request for sufficient resources. However, an improperly configured function may also set priority level excessively high, hindering other cores from obtaining enough resources. The challenges of configuring effective adaptations are specified as below.

Challenge I: Due to different QoS targets, some cores are inheritantly more sensitive to the updates in memory allocation than the others. For example, the video codec and the DSP are vastly different real-time cores. Their QoS targets are defined respectively in terms of frame rate and memory latency. To achieve 30fps, the codec needs to finish a single frame in 33ms, which is an abundant amount of time for priority-based adaptation. By comparison, the latency limit of the DSP can be even shorter than $1\mu s$, resulting in little tolerance of allocation failure. If the same NPI translation function is applied to both cores, then it could be too conservative for the DSP (i.e., the priority is not high enough) or too aggressive for the codec (i.e., the priority is overly high). Excessive resources achieved by the codec due to the inappropriately high priority level could further lead to more QoS failures at other cores.

Challenge II: For the cores with no QoS targets, its priority level depends on the health status of those with QoS targets. In other words, we provide best-effort service to cores with no QoS targets. The CPU can be such a type of core in that, unlike real-time cores, the CPU may not have a pre-fixed performance target. This does not mean that the CPU is not as critical as others. On the contrary, we want to allocate as much resource as possible to the CPU as long as real-time cores are able to achieve their performance targets. This presents another challenge to our priority-based adaptation, because the priority of the CPU cannot be simply derived through an independent NPI translation function.

Aforementioned challenges can be tackled in two ways. The first approach is to pre-tune NPI translation functions per core during design time through off-line full-system simulations. Pre-tuning has to be done in regard to numerous possible real-time workloads and memory subsystem parameters such as DRAM frequency, because memory competition varies under different scenarios. The other approach is to configure NPI translation functions in runtime according to the healthiness of real-time cores and available memory resources. In this section, we propose a runtime configuration scheme for the priority-based adaptation. The proposed scheme includes two stages: distributed self-configuration (DSC) and global regulation (GR). In the DSC stage, NPI translation functions are configured separately at each core to reflect its sensitivity to memory allocation. In the GR stage, priority levels of best-effort cores are adapted according to the healthiness of the cores with QoS targets. In the rest of this section, the proposed configuration scheme will be discussed stage by stage.

4.1 Distributed Self-Configuration

The DSC stage takes place at each core in parallel. The self-configuration consists of two components: a configurable translation function and a configuration method.

4.1.1 Configurable Translation. To begin with, we convert Equation (4) into a configurable format shown in Equation (5), where B_l and B_h are two integer variables used as lower and upper bounds of the priority range. These two variables are intended to be updated in runtime as long as they satisfy $0 \le B_l < B_h \le 7$. This way, we can configure a translation function by adjusting its priority range. Note that when B_h is raised to the maximum level and B_l is reduced to the minimum, the translation function regresses to Equation (4).

$$\text{priority}(NPI)\begin{cases} B_h & NPI \in [0,1) \\ Round\left(\frac{B_h - B_l}{NPI}\right) + B_l & NPI \in [1,14] \\ B_l & NPI \in (14,\infty) \end{cases} \tag{5}$$

According to the discussion in **Challenge I**, we want to raise the priority ranges of cores that are prone to QoS failure and vice versa for robust cores. However, adjusting both bounds at the same time could lead to unexpected results. For example, it is not obvious which one of $[3,5]$ and

[2, 6] would be a better choice to fix QoS failure. To simplify the adjustment, we increment or decrement one bound at a time. The pseudo-functions for priority range adjustment are shown below. During priority range increment, the upper bound is first raised to the maximum level before the lower bound is increased. As for priority range decrement, we reduce the lower bound to the minimum level before lowering the upper bound. As can be seen, these two priority range adjustment routines are complementary to each other.

ALGORITHM 1: priority range increment

```
function inc_range($B_l$, $B_h$)
if $B_h < 7$ then
    $B_h$ + +;
else if $B_l < B_h - 1$ then
    $B_l$ + +;
end if
endfunction
```

ALGORITHM 2: priority range decrement

```
function dec_range($B_l$, $B_h$)
if $B_l > 0$ then
    $B_l$ − − ;
else if $B_h > B_l + 1$ then
    $B_h$ − −;
end if
endfunction
```

4.1.2 *Configuration Method.* The configurable translation function and the configuration routines lay the foundation for runtime configuration of translation functions. The objective of self-configuration is to adjust the priority range of any given core to meet its target QoS without causing excessive interference to others. This includes two sub-tasks: First, the priority range requires increments in the case of QoS failure. Second, the priority range needs decrements to avoid requesting for excessive resources. Before the presentation of our scheme for runtime self-configuration, a few metrics are introduced as below.

- **Failure Rate (FR):** the percentage of time when NPI is below 1. To compute FR, the NPI of a real-time core is sampled periodically. FR is the percentage of samples with NPI < 1.
- **Healthiness:** a binary state indicating the healthiness of a given core. Different from NPI that indicates a core's health at a specific time instant, healthiness indicates the overall performance within a period of time. A core is considered *healthy* when its FR is below a certain threshold. In our evaluation, we set 3% as the FR threshold, which means a core is *healthy* if its NPI is above or equal to 1 for at least 97% of the time during the measured period.
- **Stasis:** a flag indicating whether self-configuration has reached a stasis. This metric is introduced to evaluate the progress of self-configuration. A stasis is achieved by self-configuration under two circumstances: First, the core is considered in stasis if it is healthy and its priority range is not too high to cause unnecessary competition against others; second, a core is at a stasis if it is unhealthy but its priority range is already at the highest level, i.e., References [6, 7]. In both circumstances, self-configuration cannot make further progress without violating QoS target or priority limit.

The complete configuration method is shown in Figure 5, where P_{ave} and NPI_{ave} are the average priority level and the average NPI, respectively. They are measured along with the failure rate and used to determine whether self-configuration has reached a stasis. As indicated by the colors, self-configuration operations are performed in different timescales from priority-based adaptation. First, periodical sampling is performed to monitor NPI and priority level. Frequent sampling may fail to reflect a core's healthiness, because system response to priority-based adaptation takes time, whereas long intervals between samples will slow down the following operations. In our evaluation, sampling is triggered every 100 cycles. P_{ave}, NPI_{ave}, and FR can only be evaluated after sufficient samples have been collected, which means evaluation operation is executed in a

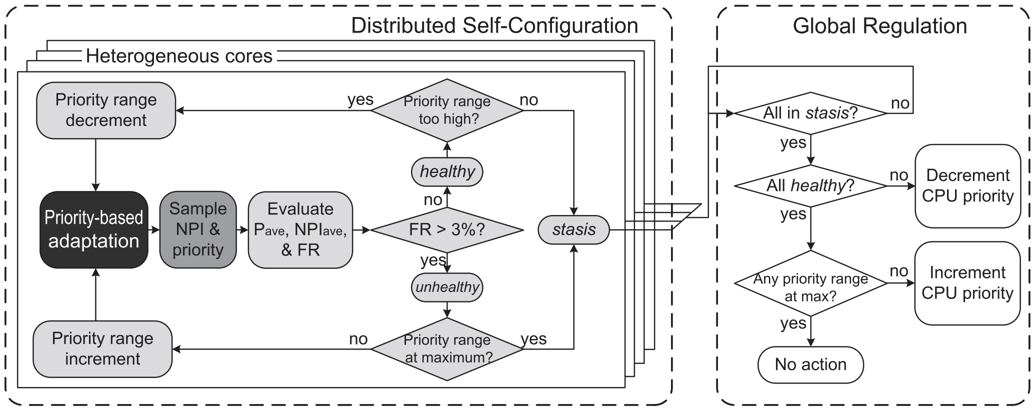


Fig. 5. Runtime configuration of priority-based adaptation including the distributed self-configuration stage and the global regulation stage where best-effort service is provided to the CPU. Operations that are executed in different timescales are labeled in different colors as a darker color indicates a smaller timescale.

further larger timescale than sampling. Evaluations are performed on every 100 samples in our evaluation.

As the last operation, if a stasis has not been reached, the priority range will be adjusted according to the runtime information. The priority range should be incremented if the core is unhealthy and its priority range is not set at the maximum level. This condition can be easily verified but checking whether $FR > 3\%$ and $[B_l, B_h] = [6, 7]$. As has been discussed, the priority range can also be decremented if a healthy core is causing unnecessary competition against others. P_{ave} is used to quantify the competition pressure that a core is causing to others. If a core is robustly healthy, then we want to keep P_{ave} around the center of the full priority range $[0, 7]$, i.e., 3, to avoid pressing too much pressure on others. Since *healthiness* is a binary metric, we still need another metric, NPI_{ave} to quantify the robustness of healthiness. We consider $NPI_{ave} > 2$ as the indication of robust healthiness. Therefore, this condition can be verified by checking whether $FR \leq 3\%$, $NPI_{ave} > 2$, and $P_{ave} > 3$.

4.2 Global Regulation

Distributed self-configuration adjusts the priority range of the translation function at each core according to their sensitivities with respect to memory service. It also prevents the core from raising its priority range unnecessarily high to interfere with others. However, **Challenge II** remains to be solved, since best-effort cores do not have NPI translation functions for their adaptations.

As has been discussed, the priority of a best-effort core should be set according to the healthiness of the cores with QoS targets. Specifically, the priority of a best-effort core can only be raised when there is no QoS failure at other cores, and it should be lowered if it causes QoS failure to others due to memory interference. A global regulator is introduced to monitor the healthiness of real-time cores and adjust the priority of best-effort cores accordingly. Different from the SARA framework and the self-configuration of translation functions that are implemented in hardware and distributed all over the SoC, this regulator can be deployed in the CPU at OS level, which enables programmable regulation policies for best-effort cores. Note that the application of global regulation is not limited to the priority of best-effort cores. It can also be applied to best-effort system features such as DVFS. As an example, frequency or supply voltage is gradually reduced to save energy if all real-time cores are stabilized in healthy states.

Figure 5 shows an example of global regulation with the CPU as a best-effort core. The regulation stage is invoked after all real-time cores with QoS targets have settled in stases. If any core is not healthy, then the regulator will decrement the priority level of the CPU to release more memory resources. If all cores are healthy and no one has set its priority range to the maximum level (indicating the brink of QoS failure), then CPU priority will be incremented. Since the regulation is performed after real-time cores have reached stases, the GR stage is operated at an even larger timescale than priority range adjustment in the DSC stage. This allows algorithms with much higher time complexity than that in Figure 5 to be deployed at the global regulator. Since prior work have extensively studied the management policies for best-effort cores and services [2, 5], we will not further the discussion on global regulation algorithms.

It is worth noting that an important feature of the SARA framework is decoupling memory allocation into different control layers across hardware and OS levels operating in different timescales, including priority-based adaptation, distributed self-configuration, and global regulation. First, priority-based adaptation enables transaction-level responsive memory allocation in hardware. Second, distributed self-configuration adjusts adaptation to fit heterogeneous QoS targets by real-time cores. At last, best-effort cores are handled by global regulation.

5 EVALUATION

In this section, the proposed SARA framework will first be tested with pre-tuned NPI translation functions to demonstrate its effectiveness in providing target performance to heterogeneous cores. Runtime configuration of priority-based adaptation is not enabled. As for memory traffic, two test cases based on the camcorder dataflow (Figure 2) will be used for demonstration. We will also show that row-buffer hits optimization can be performed efficiently within SARA framework without performance degradations. Furthermore, the SARA framework will be evaluated with runtime configuration of NPI translation functions without any preliminary tuning.

The proposed framework is modeled as in Figure 3, where memory traffic from every DMA in heterogeneous cores is generated based on memory specifications of a state-of-the-art MPSoC [17]. Our memory traffic generator models distinct memory access patterns of heterogeneous cores, including memory access rate, address access pattern, outstanding request count and request size, and so on. The emulated memory traffic streams are injected into a cycle-accurate on-chip network simulator. The simulated network topology is shown in Figure 6. Lookahead routing is applied in NoC routers with four-stage internal pipelines [3]. As for VC and switch allocations, separable allocators are implemented to perform priority-based arbitration. As the sink for all traffic streams, the memory controller and DRAM are simulated by DRAMSim2 [24], which is integrated into our network simulator. LPDDR4 timing model is applied in DRAMSim2. Table 1 shows detailed simulation settings. Table 2 lists the emulated cores and their target performance. The target performance for each core is set according to the camcorder dataflow (Figure 2) running at 30fps. For instance, the frame rotator writes and reads 1080p YUV420 images at 30fps, consuming 89MB/s for write and read requests respectively and 178MB/s in total. For the CPU, we have four CPU cores, each of which is running an application from SPEC2000/2006 benchmark, including mcf, art, sjeng, and sphinx3.

5.1 SARA Framework with Pre-tuned Configuration

5.1.1 Delivering Target Performance. To begin with, we test the SARA framework in delivering target performance to heterogeneous cores with pre-tuned NPI translation functions. Pre-tuning avoids the necessity of runtime configuration so that the efficacy of SARA framework can be demonstrated without the intricacy of heterogeneous NPI translations. Table 2 lists the pre-tuned priority ranges of heterogeneous cores. Their priority ranges are tuned to reflect their sensitivities

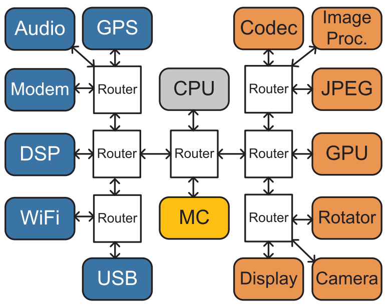


Fig. 6. The simulated topology of on-chip network.

Table 1. Memory System Simulation Settings

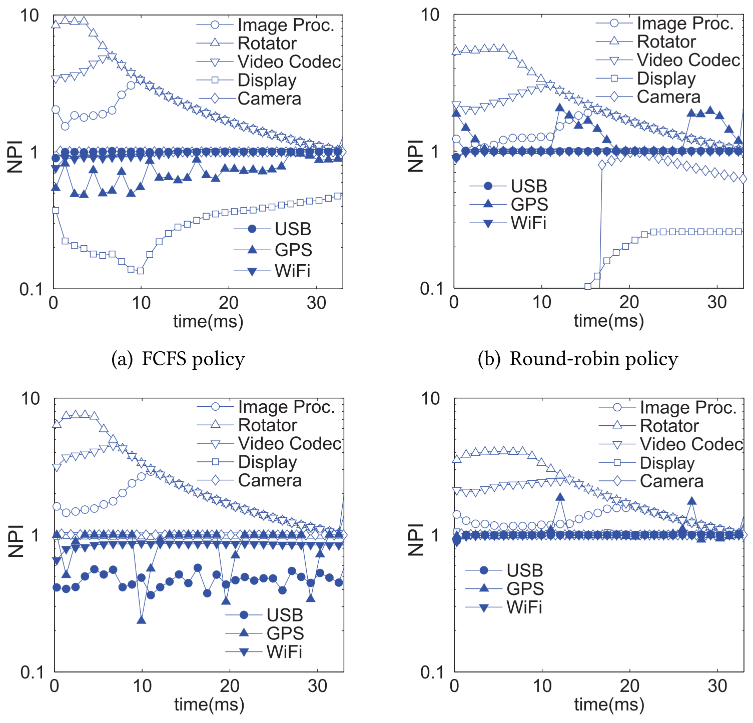
Memory Traffic	
Case A	All cores are active with DRAM @1866MHz
Case B	Inactive cores include GPS, camera, rotator and JPEG with DRAM @1700MHz
Memory System	
NoC	1GHz clock rate, dimension-order routing, 128-bit link width, 4 VCs per port, 5 buffers per VC, separable VC/switch allocation
MC	42 request buffers, 5 request queues, address mapping scheme (from physical address to DRAM address): channel-rank-row-bank-column
DRAM	2GB memory, 2KB page size, Channels-Ranks-Banks: 2-2-8, CL-tRCD-tRP(cycles): 36-34-34, tWTR-tRTP-tWR(cycles): 19-14-34, tRRD-tFAW(cycles): 19-75

Table 2. Summary of Heterogeneous Cores and Their Pre-tuned Priority Ranges

Name	Target performance	Priority range	Name	Target performance	Priority range
GPU	frame rate $\geq$ 30fps	[0, 7]	Display	buffer occupancy $\geq$ 10%	[3, 7]
DSP	average latency $\leq$ 450ns	[5, 7]	GPS	16KB/55KB in 500μs/900μs	[4, 7]
Image Processor	frame rate $\geq$ 30fps	[0, 7]	WiFi	read B/W $\geq$ 33MB/s	[3, 7]
Video Codec	frame rate $\geq$ 30fps	[0, 7]	USB	total B/W $\geq$ 370MB/s	[2, 7]
Rotator	frame rate $\geq$ 30fps	[0, 7]	Modem	total B/W $\geq$ 700MB/s	[3, 7]
JPEG	frame rate $\geq$ 10fps	[0, 7]	Audio	average latency $\leq$ 1μs	[0, 7]
Camera	buffer occupancy $\leq$ 90%	[3, 7]	CPU	N/A	2

with respect to memory allocation. Robust real-time cores are initialized with the full priority range from 0 to 7, while failure-prone cores are assigned with higher priority ranges according to their sensitivities. As a best-effort core, the CPU is assigned with a constant priority level of 2 for this test.

For comparison, four arbitration policies are used in the memory controller and on-chip network arbiters, including first-come-first-serve (FCFS), round-robin (RR), a frame-rate-based QoS policy [9, 23], and the priority-based QoS policy (Policy 1). FCFS policy serves all the transactions according to the arrival order. Round-robin policy separates transactions into different queues and



(a) FCFS policy (b) Round-robin policy

(c) Frame-rate-based QoS policy (d) Priority-based QoS policy

Fig. 7. NPI value of critical cores during one frame period (33ms) for test case A with different arbitration policies.

serves them in a round-robin fashion. In the memory controller, we have five transaction queues respectively designated to the CPU, the GPU, the DSP, media cores, and system cores. Round-robin policy also applies to on-chip network arbiters, as input queues are served in turn. The frame-rate-based QoS policy prioritizes media cores when they are missing real-time deadlines, but otherwise, the policy provides best-effort service to latency-sensitive cores. Furthermore, the priority-based QoS policy compares priority levels for arbitration and uses round-robin as the tiebreaker. The NPI of critical cores during a frame period are shown in Figure 7 when test case A is applied. As explained in Section 3.2, the NPI metric reflects performance as higher value indicates better performance. When NPI value drops below 1, it means the the target performance is not achieved.

Without reordering memory requests, FCFS policy ends up spending most of the time serving cores consuming high bandwidth. That easily leads to the starvation of latency-sensitive cores. As shown in Figure 7(a), the NPI of the GPS drops below 1, because the GPS is overwhelmed by other system cores sharing the same interconnect, such as the USB. For media cores, the video codec, the rotator and the image processor have all the frame data available at the beginning of a frame period and thus create bursty traffic, meanwhile the camera and the display generate and consume data at constant rates that are determined by image sensor and LCD panel. In Figure 7(a), media cores with bursty traffic obtain most of the bandwidth in the beginning, resulting in high NPI value. However, the display fails to achieve the target performance. The display's NPI drops as low as 0.13, which means only 13% of the target performance is achieved.

When round-robin policy is applied, the competition among media cores becomes more intense, since they share the same transaction queue in the memory controller. In Figure 7(b), the display

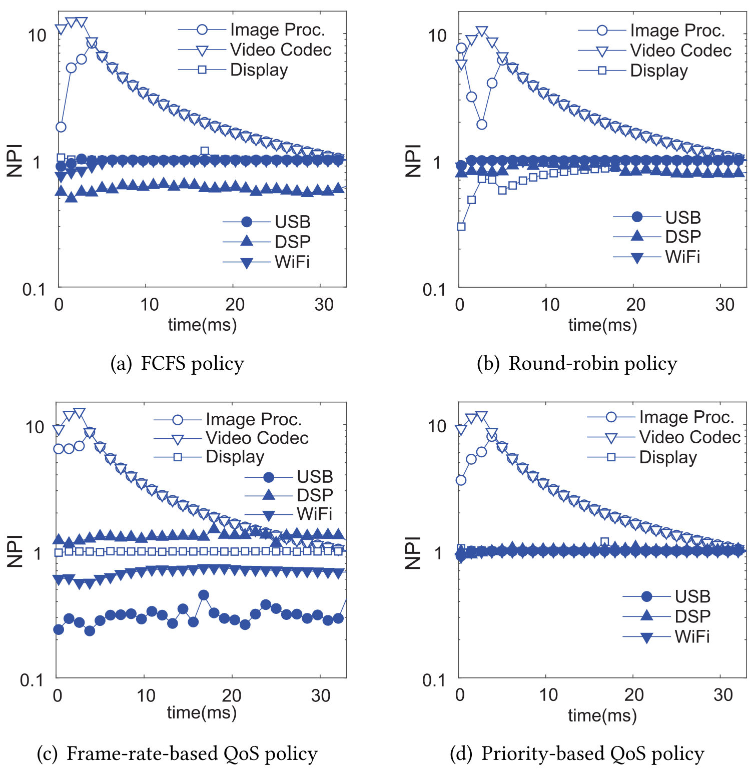


Fig. 8. NPI value of critical cores during one frame period (33ms) for test case B with different arbitration policies.

and the camera both fail due to the interference from other media cores. Less than 10% of their target performance is achieved in the worst case. In the meantime, all the system cores meet their target performance, because they avoid the interference from media cores by using a separate transaction queue.

The frame-rate-based QoS policy helps all media cores achieve NPI value above 1 in Figure 7(c). However, all system cores fail due to the absence of adaptations for the cores with different QoS targets other than frame rates.

In Figure 7(d), all the cores reach their target performance when QoS-aware scheduling is performed, because priority-based adaptations help arbiters serve the cores in urgent needs. Note that the NPI of the other cores such as the GPU are not shown, because no failure is observed from these cores.

The results by test case B are shown in Figure 8. Similarly to Figure 7, the latency-sensitive DSP suffers when FCFS policy is adopted (Figure 8(a)). When round-robin policy is applied (Figure 8(b)), the DSP suffers less, since it has its own transaction queue, while the display fails due to the increased interference from other media cores sharing the same transaction queue. Again, the frame-rate-based QoS policy fails to serve non-media cores. At last, the dynamic priorities help the memory system deliver target performance to all cores (Figure 8(d)).

Next, we take the image processor from test case A as an example to examine the priority-based adaptation in a single core. Figure 9 shows the distributions of the image processor's priority levels during one frame period, while DRAM frequency decreases from 1700MHz to 1300MHz. Each

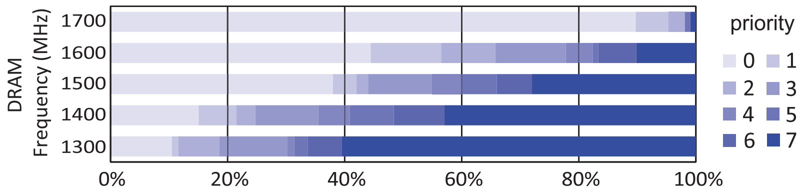


Fig. 9. Distributions of the image processor's priority levels during one frame period (33ms) with respect to different DRAM frequencies.

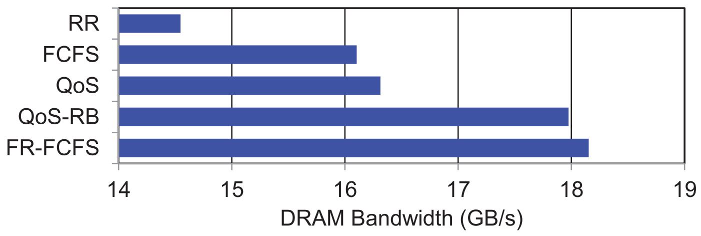


Fig. 10. Summary of average bandwidth when different scheduling policies applied.

horizontal bar is designated to a certain DRAM frequency. In a single bar, each block represents the percentage of time during which a certain priority level is adopted. Different shades of blue represent different priority levels, as higher priority levels in darker shades. As shown in Figure 9, when DRAM frequency is set to 1700MHz, for 90% of the time the image processor is adapted to the priority of 0. As frequency decreases, less memory requests can be processed by DRAM. More memory interferences and competitions happen as the result. To maintain target bandwidth, the self-adaptation leads to a gradual increase in priority levels, which can be observed through the increasing area of blocks in dark shades. When DRAM frequency is lowered to 1300MHz, the image processor has the priority of 7 for 60% of the time. In addition, as frequency decreases, the average bandwidth of the image processor remains above target bandwidth thanks to the priority-based adaptation.

5.1.2 Row-buffer Hits Optimization. As explained in Section 3.3, row-buffer hits optimization helps improve available DRAM bandwidth. With the knowledge of heterogeneous cores' urgency levels, the memory controller in the SARA framework is capable of optimizing row-buffer hits without degrading system performance.

For comparison, we compare with FR-FCFS, which prioritizes transactions going to open rows whenever it is possible and otherwise schedules transactions based on FCFS. FR-FCFS policy is expected to achieve the most row-buffer hits and the highest DRAM bandwidth. Figure 10 shows the average DRAM bandwidth during one frame period when test case A is applied. Four memory scheduling policies are tested, including RR, FCFS, QoS (Policy 1), QoS-RB (Policy 2), and FR-FCFS. Figure 11 shows the NPI of critical cores as QoS-RB and FR-FCFS are adopted.

As expected, FR-FCFS policy achieves the highest bandwidth, whereas performance degradations happen to the GPS and the display as the expense. The bandwidth by QoS-RB is slightly lower (by 1%) than FR-FCFS but much higher than other policies. Specifically, the average DRAM bandwidth obtained by QoS-RB policy is 24%, 12%, and 10% higher than RR, FCFS, and QoS policies, respectively. In the meantime, no performance degradations are caused to heterogeneous cores.

5.2 SARA Framework with Runtime Configuration

We have shown that SARA framework helps all real-time cores to achieve their target performance given pre-tuned NPI translation functions for each core. However, pre-tuning requires tremendous

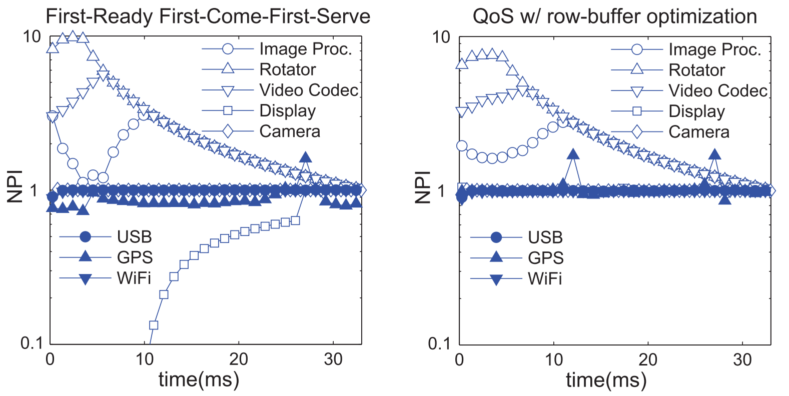


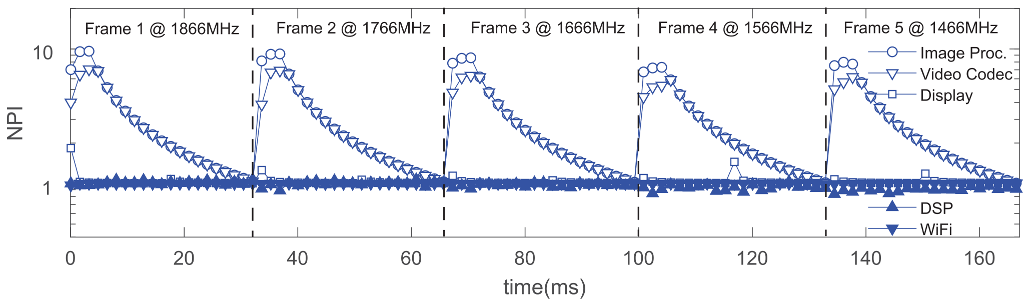
Fig. 11. NPI value for test case A with respect to FR-FCFS and QoS-RB scheduling policies.

effort during design time to consider all potential application scenarios. To spare the effort, we have proposed an alternative solution that is a two-stage runtime configuration scheme. In this section, we will demonstrate its effectiveness in comparison to SARA framework without runtime configuration.

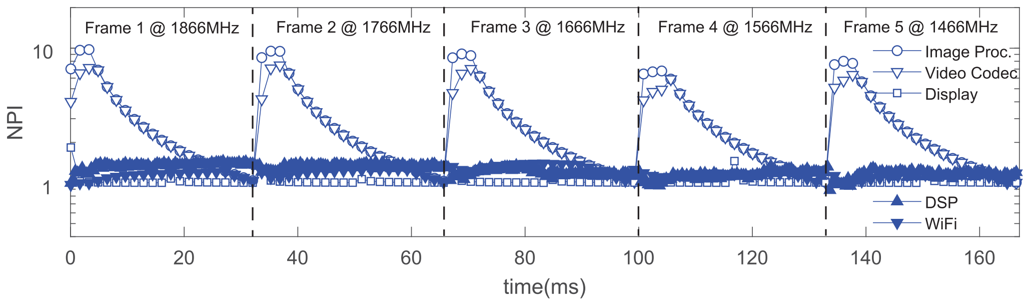
As shown in Figure 5, best-effort service is provided to the CPU while guaranteed services are allocated to other real-time cores. Test case A is applied in this evaluation. All real-time cores with QoS targets are initialized with the priority range $[0, 7]$, while CPU priority level starts with 0. The priority-based QoS policy as in Policy 1 is used in the memory system. For the demonstration of runtime self-configuration, five frames are simulated with different DRAM frequencies. DRAM frequency starts with 1866MHz in the first frame and lowers by 100MHz in each following frame. Although not all of the emulated frequencies are supported by LPDDR4, reducing memory frequency is a straightforward way to intensify memory competitions in runtime for the demonstration of self-configuration.

NPI values of major cores over five frame periods are shown in Figure 12. For comparison, four configuration schemes are implemented, including fully functional runtime configuration with the DSC stage and the GR stage (Figure 12(d)), semi-functional configuration schemes with the DSC stage and static priority levels ($P_{CPU} = 0$ or 4) for the CPU (Figure 12(c) and (b)), and no runtime configuration at all with CPU priority at 4 (Figure 12(a)). As can be seen, most real-time cores maintain their NPIs above 1 regardless of DRAM frequency and configuration scheme, which means the full priority range $[0, 7]$ enables effective priority-based adaptation at most cores. However, NPI curves by the DSP show much more variations as different frequencies and configuration schemes are applied. This has to do with the fact that the DSP is highly subject to QoS failure given the same priority range as multimedia cores. Without runtime configuration, as shown in Figure 12(a), the DSP often suffers from $NPI < 1$, which is particularly obvious during the last two frames. With runtime configuration, as shown in Figure 12(b)–(d), failure rate of the DSP is much reduced while its NPI is improved by different extents depending on CPU priority. Note that the figures are drawn at the millisecond timescale to include the total simulation time span of five frames. Distributed self-configuration actually occurs on the order of hundreds of nanoseconds.

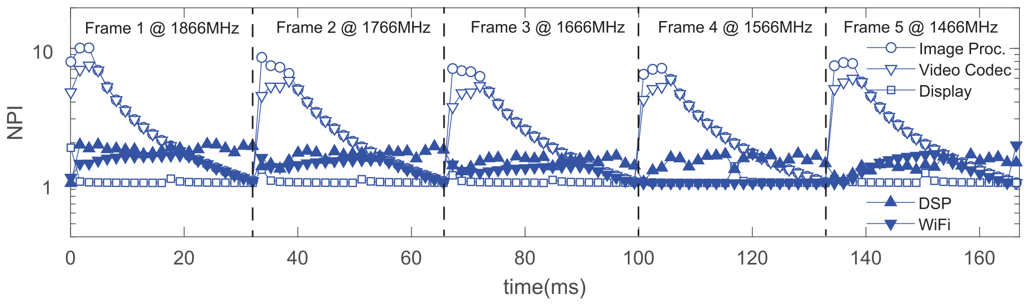
Table 3 lists the priority range, the average priority level, and the failure rate of the DSP at the end of each frame. When DSC is disabled, frame rate of the DSP is far beyond 3% most of the time. When CPU priority is set at 4, DSC helps to improve the priority level of the DSP by raising the lower bound of priority range, resulting in higher DSP priority on average and much lower failure rate. However, without global regulation, setting CPU priority constantly at 4 still leads



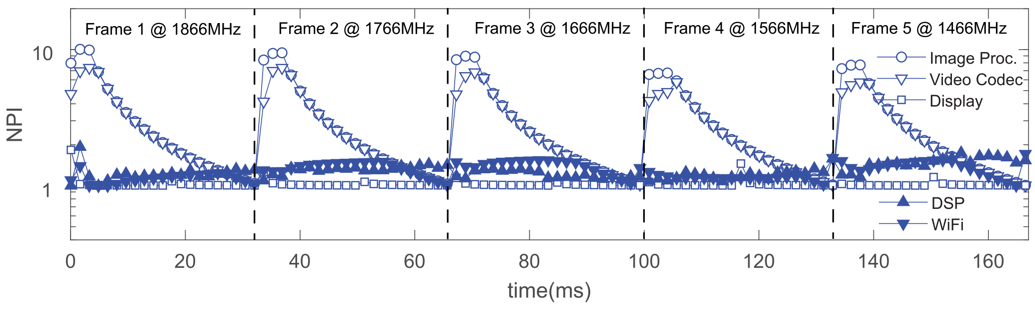
(a) No DSC and $P_{CPU} = 4$



(b) DSC and $P_{CPU} = 4$



(c) DSC and $P_{CPU} = 0$

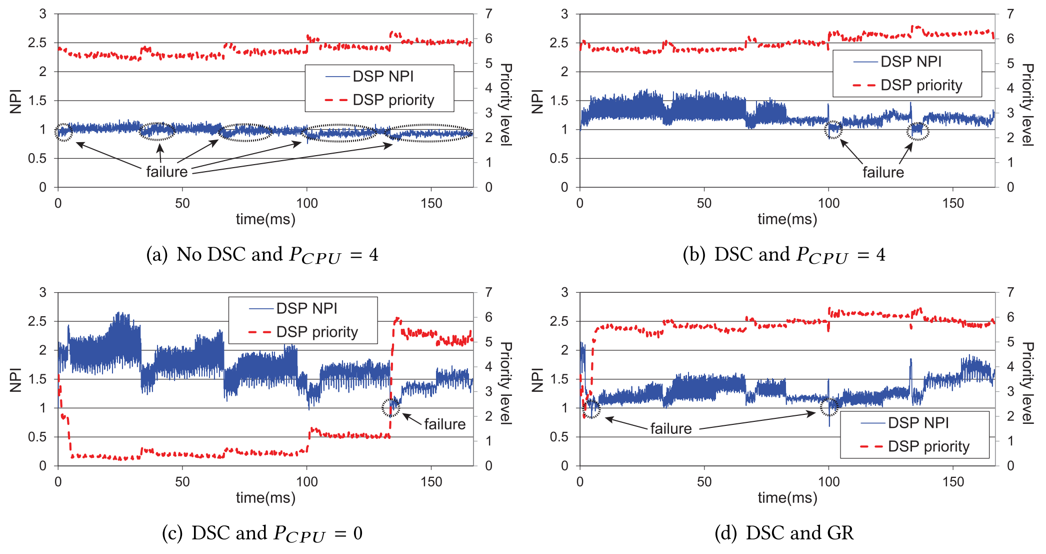


(d) DSC and GR

Fig. 12. NPI value of critical cores during five frame periods for test case A. A different DRAM frequency is set during each frame, ranging from 1866MHz to 1466MHz.

Table 3. Results of Self-configuration by the DSP per Frame, Including the Final Priority Range, the Average Priority Level over Current Frame Period, and the Overall Failure Rate during Each Frame

Frame	No DSC and P_{CPU} = 4			DSC and P_{CPU} = 4			DSC and P_{CPU} = 0			DSC and GR		
number	Range	P_{ave}	FR	Range	P_{ave}	FR	Range	P_{ave}	FR	Range	P_{ave}	FR
1	[0,7]	5.33	37.11%	[3,7]	5.58	0.66%	[0,1]	0.60	0.00%	[2,7]	5.09	1.32%
2	[0,7]	5.35	46.56%	[3,7]	5.59	0.00%	[0,1]	0.45	0.00%	[3,7]	5.60	0.02%
3	[0,7]	5.50	76.81%	[3,7]	5.82	0.00%	[0,1]	0.53	0.00%	[3,7]	5.76	0.00%
4	[0,7]	5.70	95.05%	[4,7]	6.11	2.95%	[0,2]	1.22	0.16%	[4,7]	6.10	1.48%
5	[0,7]	5.90	99.76%	[4,7]	6.24	4.37%	[2,7]	5.15	1.47%	[4,7]	5.86	0.00%



(a) No DSC and P_{CPU} = 4
(b) DSC and P_{CPU} = 4
(c) DSC and P_{CPU} = 0
(d) DSC and GR

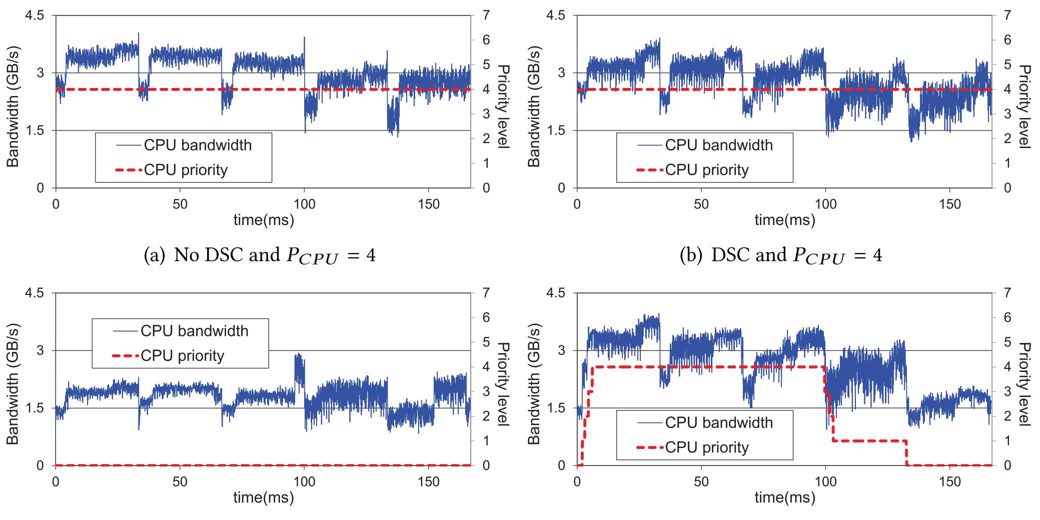
Fig. 13. Runtime NPI and priority level of the DSP with respect to different configuration schemes for priority-based adaptation.

to increasing failure rate at the DSP, which is up to 4.37% during the last frame. This is because as DRAM frequency decreases, memory bandwidth becomes more limited, and the priority of the CPU is no longer appropriate. When CPU priority is set at 0, the DSP experiences fairly low failure rate. Due to the light competition pressure from the CPU, the DSP lowers its priority range in most frames to release more resources for other cores. When DSC and GR are both applied, failure rate of the DSP remains below 3% in all frames.

To provide more insights into how runtime configuration impacts the DSP and the CPU simultaneously, Figure 13 shows NPI and priority level of the DSP under four different configuration schemes, and Figure 14 shows bandwidth and priority level of the CPU during the same time.

As mentioned, without DSC (Figure 13(a)), DSP NPI falls below 1 most of the time. When DRAM frequency is reduced to 1466MHz, DSP NPI remains below 1 during the whole frame period. Meanwhile, as shown in Figure 14(a), the CPU achieves the highest bandwidth among four schemes at the expense of DSP NPI, averaging 3.14GB/s over five frames.

When DSC is enabled and CPU priority is still set at 4 (Figures 13(b) and 14(b)), DSP NPI settles above 1 most of the time except at the beginning of the last two frames. This is because multimedia cores tend to generate bursty traffic during this time, causing major memory interference to latency



(a) No DSC and $P_{CPU} = 4$

(b) DSC and $P_{CPU} = 4$

(c) DSC and $P_{CPU} = 0$

(d) DSC and GR

Fig. 14. Runtime bandwidth and priority level of the CPU with respect to different configuration schemes for priority-based adaptation.

sensitive cores. During the same time, CPU bandwidth sees a subtle decrease due to the increased priority level of the DSP. The average CPU bandwidth is lowered to 2.89GB/s.

When DSC is enabled and CPU priority is lowered to 0 (Figures 13(c) and 14(c)), the DSP achieves the highest NPI among all configuration schemes, whereas the CPU suffers from the lowest bandwidth, which is 1.86GB/s on average.

Finally, when DSC and GR are both applied (Figures 13(d) and 14(d)), CPU priority is dynamically adjusted in response to the decreasing DRAM frequency. As the result, DSP failure rate stays below 3% in all frames and the average CPU bandwidth is 2.75GB/s, merely 5% lower than the case where CPU priority is statically set at 4.

In summary, DSC helps the DSP to settle at the healthy state by adjusting its priority range during runtime. Meanwhile, GR increases the bandwidth allocated to the CPU without causing starvation at the DSP. When the proposed runtime configuration scheme is fully enabled, the DSP and the CPU can both be properly handled by the SARA framework.

6 CONCLUSIONS

In this work, we proposed the SARA framework for memory management in heterogeneous systems. Lightweight performance meters are distributed in each core to monitor end-to-end QoS with low cost. Priority-based adaptation allows cores to adjust their priority levels according to the observed performance. The memory system with non-partitionable resources responds to QoS requests by performing priority-based management without tremendous overhead. Experimental results show that with properly tuned priority-based adaptation, the SARA framework helps all the heterogeneous cores to achieve their target performance. By comparison, without using priorities, performance of critical cores can drop lower than 10% of the target. Furthermore, memory performance such as row-buffer hits can be optimized within the SARA framework. Up to 24% improvement of total bandwidth is obtained without degrading QoS. Finally, we introduced a runtime configuration scheme for priority-based adaptation in the SARA framework to accommodate best-effort cores and real-time cores that are particularly prone to QoS failure. In our evaluation, the runtime configuration scheme successfully adjusts priority-based adaptation to reduce failure rate and allocates resource to the best-effort core without causing QoS failure.

REFERENCES

[1] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Los Alamitos, CA, 416–427. http://dl.acm.org/citation.cfm?id=2337159.2337207

[2] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. 2013. Tessellation: Refactoring the OS around explicit resource containers with continuous adaptation. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*. ACM, New York, NY, Article 76, 10 pages. DOI: https://doi.org/10.1145/2463209.2488827

[3] William Dally and Brian Towles. 2003. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[4] Boris Grot, Stephen W. Keckler, and Onur Mutlu. 2009. Preemptive virtual clock: A flexible, efficient, and cost-effective QOS scheme for networks-on-chip. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, New York, NY, 268–279. DOI: https://doi.org/10.1145/1669112.1669149

[5] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. 2012. Self-aware computing in the angstrom processor. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. ACM, New York, NY, 259–264. DOI: https://doi.org/10.1145/2228360.2228409

[6] Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[7] Javier Jalle, Eduardo Quiñones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. 2014. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *Proceedings of the Real-Time Systems Symposium (RTSS'14)*. 207–217. DOI: https://doi.org/10.1109/RTSS.2014.23

[8] Axel Jantsch, Nikil Dutt, and Amir M. Rahmani. 2017. Self-awareness in systems on chip—A survey. *IEEE Design Test* 34, 6 (Dec. 2017), 8–26. DOI: https://doi.org/10.1109/MDAT.2017.2757143

[9] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. ACM, New York, NY, 850–855. DOI: https://doi.org/10.1145/2228360.2228513

[10] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'44)*. ACM, New York, NY, 24–35.

[11] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. 1–12. DOI: https://doi.org/10.1109/HPCA.2010.5416658

[12] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. IEEE Computer Society, Los Alamitos, CA, 65–76. DOI: https://doi.org/10.1109/MICRO.2010.51

[13] Jae W. Lee, Man Cheuk Ng, and Krste Asanovic. 2008. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Los Alamitos, CA, 89–100. DOI: https://doi.org/10.1109/ISCA.2008.31

[14] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Los Alamitos, CA, 63–74.

[15] NVIDIA. 2015. Tegra X1. Retrieved from http://www.nvidia.com/object/tegra-x1-processor.html.

[16] Jin Ouyang and Yuan Xie. 2010. LOFT: A high performance network-on-chip providing quality-of-service support. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. 409–420. DOI: https://doi.org/10.1109/MICRO.2010.21

[17] Qualcomm. 2017. Snapdragon 845. Retrieved from https://www.qualcomm.com/products/snapdragon/processors/845.

[18] Amir M. Rahmani, Bryan Donyanavard, Tiago Mück, Kasra Moazzemi, Axel Jantsch, Onur Mutlu, and Nikil Dutt. 2018. SPECTR: Formal supervisory control and coordination for many-core systems resource management. In

Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18). ACM, New York, NY, 169–183. DOI: https://doi.org/10.1145/3173162.3173199

[19] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00).* ACM, New York, NY, 128–138. DOI: https://doi.org/10.1145/339647.339668

[20] S. Sarma, N. Dutt, P. Gupta, N. Venkatasubramanian, and A. Nicolau. 2015. CyberPhysical-system-on-chip (CPSoC): A self-aware MPSoC paradigm with cross-layer virtual sensing and actuation. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'15).* ACM, New York, NY, 625–628. DOI: https://doi.org/10.7873/DATE.2015.0349

[21] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. 2011. METE: Meeting end-to-end QoS in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev.* 39, 1 (Jun. 2011), 13–24. DOI: https://doi.org/10.1145/2007116.2007119

[22] Yang Song, Kambiz Samadi, and Bill Lin. 2016. Single-tier virtual queuing: An efficacious memory controller architecture for MPSoCs with multiple realtime cores. In *Proceedings of the 53rd Annual Design Automation Conference (DAC'16).* ACM, New York, NY, Article 6, 6 pages. DOI: https://doi.org/10.1145/2897937.2898093

[23] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. 2016. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Trans. Archit. Code Optim.* 12, 4, Article 65 (Jan. 2016), 65:1–65:28.

[24] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMsim: A memory system simulator. *SIGARCH Comput. Archit. News* 33, 4, 100–107. DOI: https://doi.org/10.1145/1105734.1105748

[25] Po-Han Wang, Cheng-Hsuan Li, and Chia-Lin Yang. 2016. Latency sensitivity-based cache partitioning for heterogeneous multi-core architecture. In *Proceedings of the 53rd Annual Design Automation Conference (DAC'16).* ACM, New York, NY, Article 5, 6 pages. DOI: https://doi.org/10.1145/2897937.2898036

[26] Jia Zhan, Onur Kayiran, Gabriel H. Loh, C. R. Das, and Yuan Xie. 2016. OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16).* 1–13. DOI: https://doi.org/10.1109/MICRO.2016.7783731

Received October 2018; revised January 2019; accepted February 2019