Interference from GPU System Service Requests

Arkaprava Basu[*]
Indian Institute of Science
arkapravab@iisc.ac.in

Joseph L. Greathouse
AMD Research
joseph.greathouse@amd.com

Guru Venkataramani[*]
George Washington University
guruv@gwu.edu

Ján Veselý[*]
Rutgers University
jan.vesely@cs.rutgers.edu

***Abstract*—Heterogeneous systems combine general-purpose CPUs with domain-specific accelerators like GPUs. Recent heterogeneous system designs have enabled GPUs to request OS services, but the domain-specific nature of accelerators means that they must rely on the CPUs to handle these requests.**

Such system service requests can unintentionally harm the performance of unrelated CPU applications. Tests on a real heterogeneous processor demonstrate that GPU system service requests can degrade contemporaneous CPU application performance by up to 44% and can reduce energy efficiency by limiting CPU sleep time. The reliance on busy CPU cores to perform the system services can also slow down GPU work by up to 18%. This new form of interference is found only in accelerator-rich heterogeneous designs and may be exacerbated in future systems with more accelerators.

We explore mitigation strategies from other fields that, in the face of such interference, can increase CPU and GPU performance by over 20% and 2×, respectively, and CPU sleep time by 4.8×. However, these strategies do not always help and offer no performance guarantees. We therefore describe a technique to guarantee quality of service to CPU workloads by dynamically adding backpressure to GPU requests.

I. Introduction

Heterogeneous systems combine general-purpose CPUs and domain-specific accelerators to increase performance and energy efficiency. GPUs are perhaps the most popular and capable accelerators today [23], [28], [40], but SoCs may dedicate more than half of their area to dozens of others [13]. These can include media processors [2], cryptography engines [33], and accelerators for web search [52], computer vision [18], machine learning [17], [37], regular expressions [25], and databases [63].

It is safe to say that future systems will have numerous highly capable accelerators. As part of this transition to heterogeneity, accelerators like GPUs are now first-class citizens that share a global, coherent view of virtual memory and can launch tasks both to CPUs and to each other [31], [38], [57]. GPUs have outgrown the coprocessor model, and software can now use them for more advanced tasks [58].

Recent designs have allowed GPUs to request *system services* from the OS, such as file system accesses, page faults, and system calls [60], [61]. For instance, GPUs and other accelerators that support shared virtual memory may take page faults for tasks such as memory protection, swapping, copy-on-write, or memory compression [8], [35], [61].

Unfortunately, accelerators cannot directly execute these system services because they are not fully programmable; their application-specific nature is their raison d'être. Even modern GPUs lack full support for the types of privileged accesses needed by OS routines. In addition, it can be dangerous to run privileged code on unverifiable third-party hardware designs [46]–[48], [60].

General-purpose CPUs must therefore handle system service requests (SSRs) from these GPUs, potentially disrupting unrelated CPU applications through stolen cycles and polluted microarchitectural state. We demonstrate that this *host interference from GPU system services* (HISS), coming from even *a single* GPU, can degrade the performance of contemporaneous CPU workloads by up to 44%. Similarly, unrelated CPU-resident work can delay the handling of GPU SSRs, reducing accelerator throughput by 18%.

Such SSRs can also prevent idle processors from saving power by sleeping. A single GPU's SSRs can reduce CPU sleep time from 86% to 12%, implying that CPUs may rarely sleep in future accelerator-rich SoCs.

Unlike the interference that may occur between CPU applications, the source of this new type of interference is the system's heterogeneity. GPUs run semi-independently and are not subject to the same OS quality of service controls as CPU tasks. It is thus imperative to take this new interference into account in accelerator-rich SoCs.

We analyze the efficacy of mitigation techniques inspired by existing fields: interrupt steering [30], interrupt coalescing [32], and driver optimizations. These can improve CPU and accelerator performance by 20% and 2×, respectively, and they can increase CPU sleep time by 4.8×. Nevertheless, they do not universally mitigate this interference and provide no performance guarantees.

We therefore design a mechanism that can control CPU quality of service (QoS) in the face of HISS. We track the time spent handling GPU SSRs and dynamically apply backpressure to these requests. This will eventually stall the GPU, reducing CPU interference. This allows control of CPU overheads at the cost of lower GPU performance.

This work explores trade-offs between CPU and GPU performance and energy in heterogeneous systems making system calls and makes the following contributions:

- We demonstrate host interference from GPU system services, a new form of interference between CPUs and GPUs. Similar interference can happen between CPUs and any other OS-service requesting accelerators.
- We quantify the performance and power effects of such interference on a real heterogeneous SoC and study miti-

[*]Authors contributed while at AMD Research.

gation strategies that expose complex trade-offs between power, CPU performance, and GPU performance in the face of this interference.

- We demonstrate the need to ensure CPU QoS in the face of GPU SSRs and evaluate such a technique in the Linux® kernel.

II. Motivation and Background

We begin by exploring how modern accelerators like GPUs have enabled system service requests (SSRs). We then give SSR examples, show how they are performed, and describe the type of overheads they cause.

A. GPU System Service Requests

While GPUs were once standalone devices [45], they now interact much more closely with other components. AMD [11] and Intel [38] support coherent shared virtual memory between CPUs and GPUs, and Oracle supports the same for their database accelerators [7], [54]. CCIX [15], Gen-Z [24], HSA [31], and OpenCAPI [49], [57] allow accelerators to share coherent virtual memory with the rest of the system. This allows fine-grained data and pointer sharing and allows accelerator work launching without going through the OS [12], [14].

Rather than simply receiving a parcel of work and returning an answer, modern GPUs can operate in a tightly coupled manner with CPUs [26], [58] and can launch work to themselves [34], [36] or other devices [41]. This paper focuses on a new feature of advanced accelerators: the ability to request services from the operating system.

System Service Examples. Table I describes a sample of the system services that a modern GPU can request and a qualitative assessment of how complex they are to perform. Beyond these examples, Veselý et al. provide a more thorough study of accelerator system calls [60].

SSR	Description	Complexity
Signals	Allows GPUs to communicate with other processes.	Low
Page faults	Enables GPUs to use un-pinned memory [61].	Moderate to High
Memory allocation	Allocate and free memory from the GPU [56].	Moderate
File system	Directly access/modify files from GPU [53].	High
Page migration	GPU initiated memory migration. [5].	High

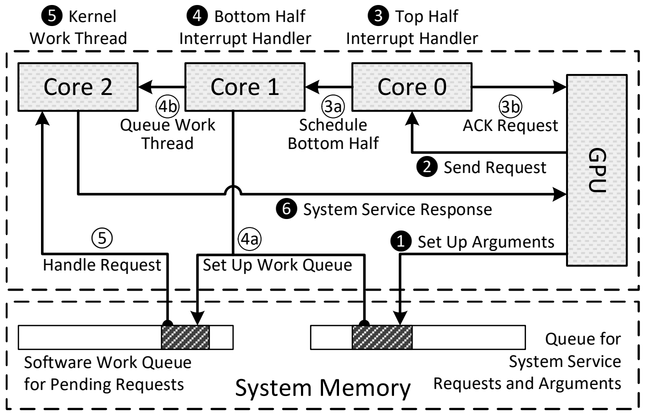
TABLE I: Descriptions of various GPU SSRs and a qualitative estimate of their complexity.

Page faults imply accelerators have the same virtual memory protection and swapping capabilities as traditional CPUs [8], [27], [35], and Veselý et al. quantitatively showed how GPUs benefited from demand paging [61]. Spliet et al. demonstrated that memory management directly from an accelerator could ease programmability [56], and others showed benefits from accessing file systems and networks from accelerators [22], [39], [53]. Finally, Agarwal et al. showed benefits from migrating accelerator data in NUMA systems [5].

The complexity of handling these SSRs varies. Signals require little more than informing the receiving process of the request. Page faults, however, could involve disk and file system accesses. Such I/O requests, as well as network accesses, can be extremely complex. In all cases, these SSRs require access to kernel-level data and logic that can vary wildly between OSs.

B. Performing GPU SSRs

Despite the benefits of SSRs, GPUs or other accelerators cannot themselves perform these services. Their hardware specialization prevents them from running the OS; adding logic to run general-purpose OS software would negate many of their performance and efficiency benefits.



❺ Kernel Work Thread
❹ Bottom Half Interrupt Handler
❸ Top Half Interrupt Handler
Core 2
Core 1
Core 0
(4b) Queue Work Thread
(3a) Schedule Bottom Half
(3b) ACK Request
GPU
❷ Send Request
❻ System Service Response
⑤ Handle Request
(4a) Set Up Work Queue
❶ Set Up Arguments
Software Work Queue for Pending Requests
System Memory
Queue for System Service Requests and Arguments

Fig. 1: Handling GPU system service requests

In addition, there are potential security concerns with allowing third-party accelerators to directly run OS routines [46]–[48]. The OS provides security and memory isolation from buggy and/or malicious devices, so third-party devices cannot be trusted to run system software. As such, any accelerator SSRs are thus typically serviced by the host CPUs.

Figure 1 illustrates the basic process of a GPU requesting an OS service (SSR) in the Linux® kernel. While the specific steps may differ slightly for other accelerators, this generic description remains the same.

① The GPU stores the data needed for the SSR into a memory buffer; this can include information such as the system call number and the function arguments.

② The GPU interrupts a CPU to tell it that there is a new SSR. The GPU can optionally wait to accumulate multiple requests before sending this interrupt.

③ The CPU that received the interrupt enters the *top half interrupt handler*, which takes place in the hardware interrupt context with further interrupts disabled. **(3a)** The top half handler then schedules a *bottom half interrupt handler*, and an inter-processor interrupt (IPI) is performed if the bottom half's kernel thread is located on a different core. Such split designs are common in device drivers because they quickly re-enable interrupts and allow the scheduler to defer the real work to opportune times [19]. **(3b)** The top half handler then acknowledges the request and re-enables interrupts.

④ The kernel thread for the bottom half handler is later scheduled, potentially onto a different core from the top half. (4a) It reads information about the SSR from memory and performs some amount of pre-processing. Kernel threads can harm the performance of other CPU tasks since they run at high priority. As such, many drivers further split the bottom half and (4b) queue the bulk of the work to a kernel worker thread [62].

⑤ The worker thread is later scheduled by the OS, where it handles the SSR. Depending on the system service complexity, a great deal of time may be spent here.

⑥ Afterwards, the worker thread informs the GPU (or devices like the IOMMU) that its request has been completed.

C. Example GPU SSRs

This section details two specific examples of GPU SSRs supported by today's heterogeneous processors and OSs.

Page faults: The Linux `amd_iommu_v2` driver implements a page fault handler for AMD GPUs [4]. The GPU requests address translations from the IO Memory Management Unit (IOMMU), which walks the page table and can thus take a page fault. In this case, ① the IOMMU writes the information about the request (e.g., the virtual address) to memory and ② interrupts a CPU.

③ The top half interrupt handler in the IOMMU driver (3a) schedules a bottom half handler kernel thread and (3b) quickly sends an acknowledgement to the IOMMU. ④ Later, the bottom half kernel thread (4a) reads the IOMMU request buffer and ascertains that this is a page fault. It then (4b) queues the page fault handler.

⑤ The OS later schedules a kernel thread to read the request from the work queue and handle the page fault. This may involve a great deal of work, such as retrieving data from a swap file. ⑥ It then notifies the IOMMU of the completion, and the IOMMU, in turn, tells the GPU.

Signals: AMD GPUs can also signal other processes using a similar sequence of CPU handlers.

Signals do not make use of the IOMMU. Instead, they execute the GPU's `S_SENDMSG` instruction to accomplish step ② of Figure 1 [1]. The rest of the steps are similar to page faults, except that different OS routines are invoked in step ⑤ (and different memory queues are used for ① and ④). In the end, the OS signal handler wakes the target process that is to receive the signal.

D. System Service Request Overheads

SSRs can cause slowdowns both on the GPU and on potentially unrelated CPU applications, as illustrated in Figure 2. The solid light blue bars represent useful user-level work on the GPU and CPUs.

The dark gray sections are time spent in the kernel handling the SSR; the labels match those from Figure 1. This time is *direct CPU overhead* caused by the extra OS instructions that must be executed to handle the SSR.

Sections labeled 'a' (red hatching) and 'b' (blue cross-hatching) are *indirect CPU overheads*. The former ('a') is time spent transitioning between user and kernel mode. This takes place before and after all of the handlers. The latter ('b') is user-mode execution running slower due to the kernel's pollution of microarchitectural state, which may increase with larger kernel handlers [55].

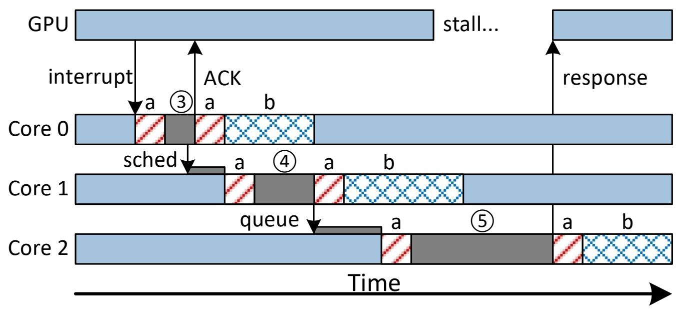


Fig. 2: GPU service request overheads. See Section II-D for details of the bar colors and labels.

The GPU may also eventually stall due to the latency of handling the SSR. We observe that SSRs can directly and indirectly affect CPU performance, and CPU's time to handle the request can affect GPU's performance.

III. Experimental Setup

The following sections demonstrate SSRs interference on a real heterogeneous processor with a CPU and a GPU.

System Configuration: Table II describes the system used in our experiments. Our heterogeneous processor, an AMD A10-7850K, has four CPU cores and an integrated GPU, which we use as our accelerator.

Methodology: To measure the impact of interference from the SSRs, we concurrently run independent CPU-only and GPU-accelerated applications. We focus on independent workloads to demonstrate that accelerator SSRs breach performance isolation between unrelated processes running on separate processors. Benchmarks that *simultaneously* utilize CPUs and GPUs have only recently appeared [26], [58], and they have yet to utilize SSRs. Nonetheless, SSR interference would also harm such applications.

Our CPU applications are PARSEC v2.1 running four threads and *native* inputs [10], and we use hardware performance counters to gather performance measurements.

SSR generating GPU workloads: Commercial processors only recently added support for SSRs [31]. Thus, applications directly exploiting SSRs are not yet commonplace. We therefore focus on an SSR that has already shown benefits, GPU page faults [61]. Traditionally, memory that the GPU *may* access is pinned before starting the GPU work. Veselý et al. demonstrated two applications, *BPT* [20] and *XSBench* [59], that could significantly consolidate their memory footprints by using GPU page faults [61]. We use these workloads in our tests, and, like that work, we use soft page faults (meaning that the handler – ⑤ in Figure 1 – does not access the disk).

In addition, we modified three other benchmarks to also take advantage of GPU page faults. We modified *BFS* and *SpMV* from SHOC [21] and *SSSP* from Pannotia [16] to allocate their

SoC	AMD A10-7850K
CPU	4× 3.7GHz AMD Family 15h Model 30h
Accelerator	720 MHz AMD GCN 1.1 GPU
Memory	32 GB Dual-Channel DDR3-1866
Software	Ubuntu 14.04.3 LTS 64-bit Linux® 4.0.0 with AMD HSA Driver v1.6.1

TABLE II: Test System Configuration

inputs on demand. When their GPU kernels access this data, the GPU creates a soft page fault that the host handles, as described in Section II-C.

Future accelerator-rich SoCs may also cause more SSRs than we see in our single-accelerator testbed. As such, we designed a microbenchmark to test what would happen if our accelerator continually creates SSRs at a high rate. This microbenchmark (*ubench*) streams through a data array on the GPU, and each of its memory accesses generates a page fault.

We ran each combination of CPU and GPU benchmark 3 times to increase confidence in our results.

IV. Impact of GPU SSRs

We here quantify the performance and power effects of interference due to GPU SSRs.

A. Implication on Performance

CPU Overhead: Figure 3a shows how GPU SSRs affect the performance of concurrent, but independent, CPU applications. Each bar represents the performance (1/runtime) of a CPU application while a GPU application creates SSRs. Each bar's height is normalized to the same pair of applications, but without the GPU application generating any SSRs. Thus, any bar below 1 shows a performance loss *only due to the SSRs*.

The performance of unrelated CPU applications can drop by up to **31%** due to SSRs from a single accelerator (*fluidanimate* with *SSSP*). The average performance loss on the CPU in this case is **12%**. Projecting to future SoCs with our microbenchmark, the performance of CPU applications could degrade by up to **44%** (x264) and by **28%** on average.

These overheads depend both on the CPU application and how the GPU application requests system services. For instance, the *raytrace* CPU benchmark sees less interference in general because much of its execution is single-threaded; the SSRs can be handled by other idle cores. In contrast, *fluidanimate* is affected more by the same SSR patterns since SSRs harm this benchmark's L1 cache hit rate, leading to high indirect overheads (blue cross-hatched segments in Figure 2).

The SSR patterns created by the GPU application have a strong effect on CPU overhead. *BFS*, for instance, has a low SSR rate and its SSRs are clustered near the beginning of its execution. This limits the interrupts that disturb the CPUs, yielding less slowdown. Our GPU microbenchmark, on the other hand, continually creates a large number of SSRs, so most CPU applications experience significant slowdown.

Accelerator Overhead: Accelerators that request SSRs are also affected by independent CPU applications. Figure 3b shows the performance of our SSR-causing GPU applications running in conjunction with our CPU workloads. This is normalized to the performance of each application when the CPUs are idle. Bars lower than 1 indicate performance degradation due to the CPU application interfering with parts of the SSR handling chain. For example, the work thread (⑤ in Figure 1) can be delayed as other user-level work runs. This host interference leads to slowdowns up to **18%** (*SSSP* and *streamcluster*) and an average of **4%**.

We also see the interplay between the needs of concurrently running CPU and GPU applications. *BPT*, *SSSP*, and our microbenchmark are more heavily affected by interference from the CPUs; their kernels stall whenever CPU workloads delay SSR handling. The CPU benchmark *streamcluster* delays SSR responses so much that accelerator benchmarks suffer; the average GPU performance drops by 8%.

We note that GPU application performance is slightly higher than 1 in some cases. In part, this is because these active CPUs can respond to SSRs slightly faster compared to idle CPUs that may be asleep when the interrupts arrive.

Summary: GPU SSRs can degrade performance of concurrent CPU applications. Furthermore, GPUs requesting system services can lose performance due to contemporaneous CPU work. This problem may be exacerbated as future chips include many such accelerators that request system services at a higher rate.

B. Implication on Energy Efficiency

Sleep states play a crucial role in energy efficiency. When a CPU is idle, it can shut down circuits in order to reduce power. Short periods of sleep can be detrimental, since some sleep states have overheads like cache flushes that must be amortized. However, long periods of sleep can significantly decrease energy usage [9].

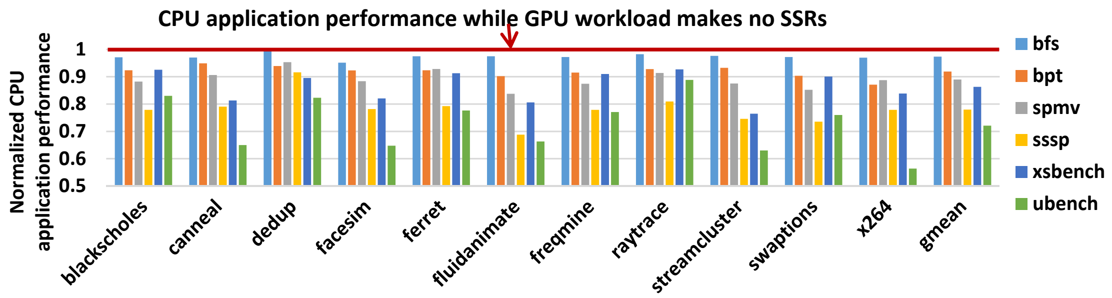
Unfortunately, SSRs can leave CPUs with little opportunity to sleep. We measured the fraction of time our CPUs were in their lowest-power sleep state, "Core C6" (CC6) [3] while there is *no* CPU-only work, but while running GPU applications both with and without SSRs. The differences between these show the CC6 residency lost due to SSRs.

Figure 4 shows the percentage of time the CPUs reside in CC6 (y-axis) while executing our GPU applications. SSRs always decrease CPU sleep time, but the amount is affected by the number of SSRs and their pattern. *BFS* sleep time decreases 14 percentage points because its SSRs are clustered early in its execution; the CPUs can sleep afterwards. The other four applications see a reduction of 23-30 percentage points. Effectively, these applications have an average of one more core awake throughout their runs.

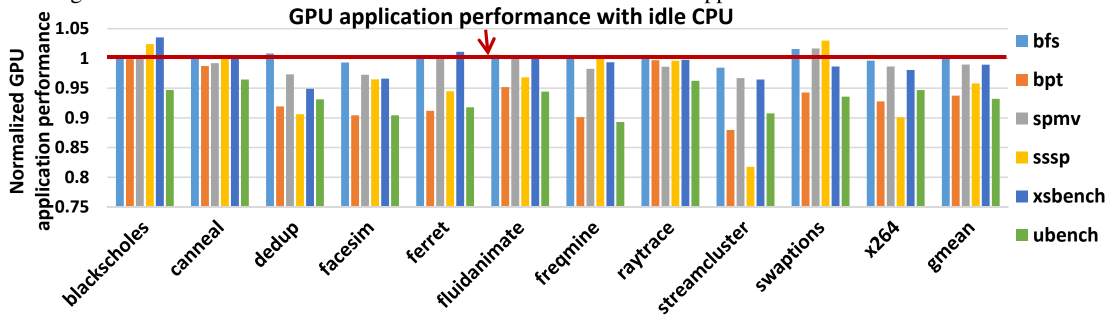
Our microbenchmark that constantly requests SSRs reduces sleep time from 86% to 12%. Nearly all opportunities to sleep are eliminated, which bodes poorly for future accelerator-rich SoCs. As their SSR request rate and regularity increases, a vital mechanism for saving energy may be virtually eliminated.

C. Analysis of SSR Overheads

This section describes some sources of accelerator SSR overheads. First, SSR interrupts (② in Figure 1)



(a) Normalized performance of CPU-only applications (PARSEC) due to SSRs (page faults) from concurrently running GPU workloads. Performance normalized to the same GPU applications without SSRs.



(b) Normalized GPU performance when making SSRs and running concurrently with CPU applications. Performance is normalized to GPU applications running with an idle CPU.

Fig. 3: Performance implications of system service requests (page faults) from an accelerator (GPU)

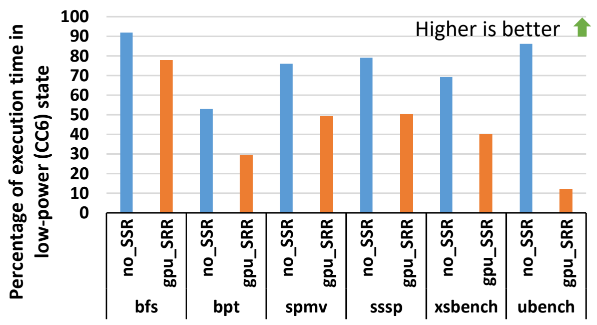
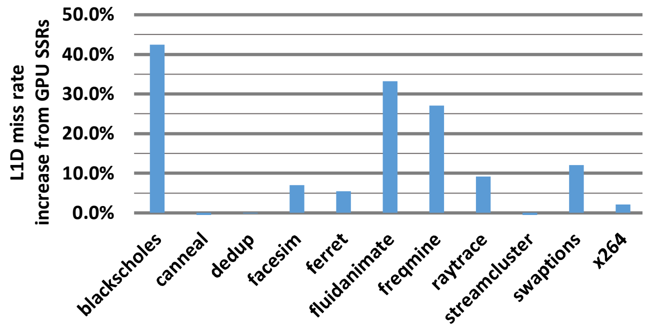


Fig. 4: CPU low-power sleep state residency with and without GPU system service requests.

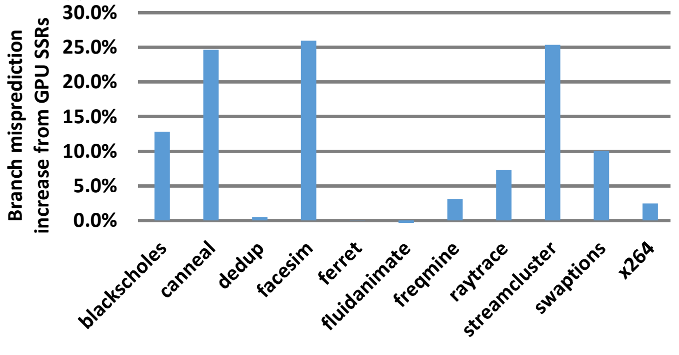
are evenly distributed across all CPUs (measured using `/proc/interrupts`). This is to balance the work required of each core, but it causes every core to suffer direct overheads from the interrupts (as in Figure 2).

Next, there is a 477× increase in inter-processor interrupts (IPIs) when our microbenchmark creates SSRs due to the top half of the interrupt handler waking the bottom half (3a in Figure 1). IPIs cause direct overheads in multiple cores, further reducing performance.

Servicing these requests also indirectly affects the user-level CPU applications by polluting the microarchitectural states of the CPUs. The SSR handlers evict useful user-space data from structures like caches and branch predictors. Then, when user-level execution resumes, the application will experience lower performance [55]. Figure 5 demonstrates how SSRs from our microbenchmark increase the L1 data cache misses and branch misprediction rate of unrelated CPU-only applications.



(a) Increase in user-level cache misses from GPU SSRs.



(b) Increase in user branch mispredictions due to GPU SSRs.

Fig. 5: Microarchitectural effects of GPU SSRs

V. Mitigation Techniques

This section explores interference mitigation strategies from domains such as high-speed networking. We study each technique first in isolation and then in combinations.

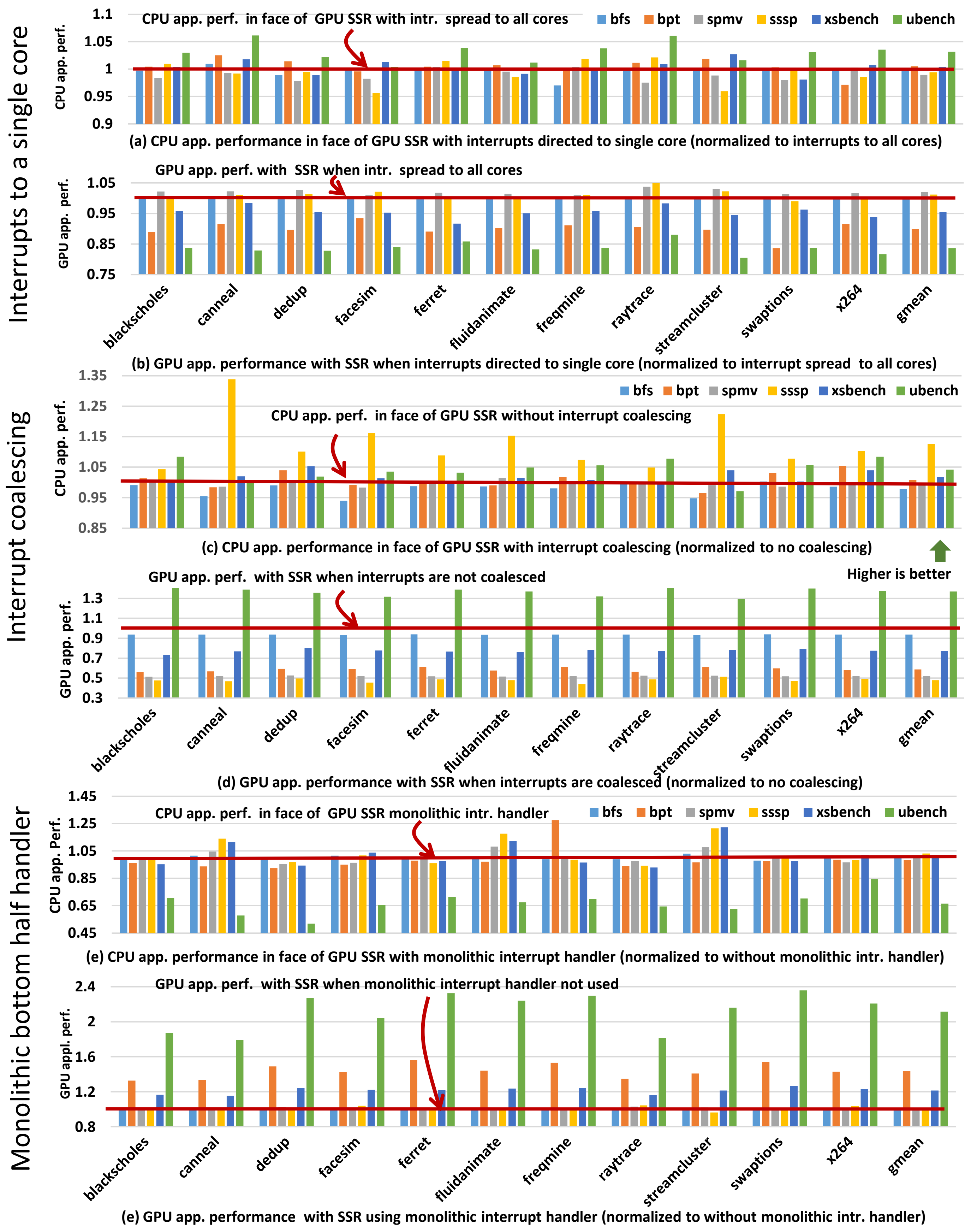


Fig. 6: Performance of GPU SSR overhead mitigations. ubench performance is the SSR rate relative to an idle CPU.

A. Steering Interrupts to a Single Core

Section IV-C showed that SSR interrupts were uniformly distributed across all CPUs. This has also been observed in network stacks with high packet arrival rates. To isolate the detrimental effects of these interrupts, we modified the IOMMU MSI configuration registers to steer the SSR interrupts to a single CPU [30].

Interrupt steering has the potential to reduce overheads around ③ in Figure 2, but it can also reduce fairness. Workloads that require performance balance between the cores (e.g., applications with static input partitioning) may suffer. This can also stall the GPU while it waits to send interrupts to the bottlenecked core.

Figure 6a shows the performance of our CPU benchmarks when steering SSR interrupts to a single core. These results are normalized to the case where the GPU SSRs are spread across all of the CPUs.

Interrupt steering neither universally helps nor hurts the performance of CPU applications. Interrupts from applications such as *SSSP* can harm the performance of some CPU applications (e.g., *facesim*) because one core is overburdened with interrupts. However, if the interrupts coming from the GPU could inundate many cores (e.g., with our microbenchmark), steering them all to one core reduces the harm.

Figure 6b shows a similar story for the performance of the GPU applications. Some GPU applications gain a small amount of performance due to faster handling of the individual interrupts. However, when there are many continuous SSRs, the single core that handles the requests becomes a bottleneck.

B. Interrupt Coalescing

While a CPU must be interrupted for GPU SSRs, it is not necessary to do so immediately. GPUs can request multiple unrelated SSRs within a short time, and these can be accumulated before sending the interrupt.

Similar to techniques used in high-performance networking [32] and storage [6], we leverage this freedom to perform interrupt coalescing. Specifically, we configure the IOMMU (PCIe® register `D0F2xF4_x93`) to allow it to wait 13 μs (its maximum wait time) before raising the interrupt; other SSRs that arrive during this interval are combined into the same interrupt. This can reduce the number of CPU interrupts, but it can also add latency to accelerator SSR requests.

Coalescing reduced the number of SSR interrupts by an average of 16%. Each interrupt handles multiple SSRs, allowing the direct overheads to be shortened. In addition, there are fewer switches into kernel mode, so the indirect overheads are reduced. On the other hand, interrupt coalescing can also reduce load balancing, since each interrupt can come with a variable amount of work.

Figures 6c and 6d illustrate the impact of interrupt coalescing on CPU and GPU applications, respectively. Coalescing can help when there are continuous interrupts; we see a 13% increase in CPU performance when running *SSSP*. However, the load balancing problem causes a 2% drop in performance when *BFS* is running.

For GPU performance, the efficacy of interrupt coalescing depends on the application's need for SSR throughput or latency. Our full applications, such as *SSSP*, see slowdowns as high as 50% because the handling of their SSRs can be significantly delayed while waiting for the coalescing period. This can happen when handling the SSR is on the GPU kernel's critical path.

On the other hand, our microbenchmark sees a performance increase because it has other parallel work that does not rely on the SSR results. As such, coalescing the interrupts allows more of them to be handled before the IOMMU must stall, raising its throughput.

C. Monolithic Bottom Half Handler

Figure 1 illustrates a split driver design where a top half handler receives and acknowledges the SSR interrupt (③) and a bottom half handler (④) pre-processes the data for the SSR. Finally, a kernel thread can perform the system service for the GPU (⑤).

We explore the implications of such a design by modifying our IOMMU driver. We moved the pre-processing of the SSR (④) into the top half (③). This removed the extra IPIs described in Section IV-C, since there is no need to interrupt another core to wake up the bottom half.

This removes the direct and indirect overheads shown on Core 1 of Figure 2, while still handling the majority of the SSR outside hard interrupt context (⑤). This *does not* eliminate the top-half/bottom-half paradigm; it simply moves more work into the top half. On the other hand, it moves this additional work into the hard interrupt context.

Figures 6e and 6f show the impact of this change on CPU and GPU performance, respectively. Figure 6e shows that this modification can help some applications by reducing the amount of overhead caused by each SSR. However, the increased work in the interrupt context means our microbenchmark causes 35% more overhead.

Figure 6f shows that removing the two-stage bottom half handler greatly improves the GPU application performance (up to **2.3**×) by eliminating the OS scheduling delay in waking up the first bottom half handler. This reduces the latency of handling each SSR.

D. Combining Multiple Techniques

The above-mentioned mechanisms are orthogonal to one another and may interact in non-obvious ways. As such, this section studies their effects when combined.

Figure 7 shows a Pareto comparison of the above three techniques for both CPU and GPU application performance. The X axis shows the geometric mean (across all applications) of the CPU overhead caused by our microbenchmark, while the Y axis shows the geometric mean of the microbenchmark's performance across all CPU workloads. Points to the right have better CPU performance; points to the top have better GPU throughput.

Figure 8 shows a similar frontier for the other GPU applications. As demonstrated in Figure 6, the larger GPU

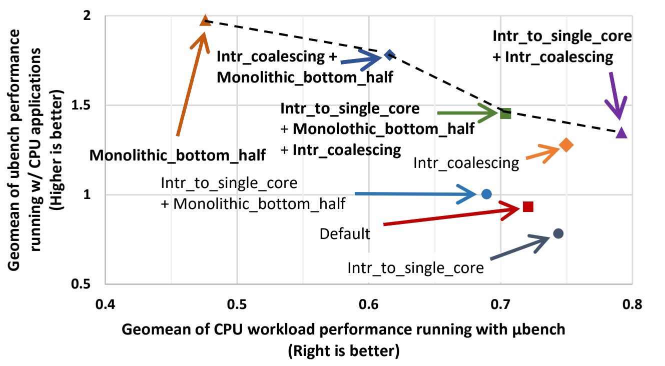


Fig. 7: Pareto chart showing trade-offs between combinations of mitigation techniques for our microbenchmark. Bolded entries are on the Pareto frontier.

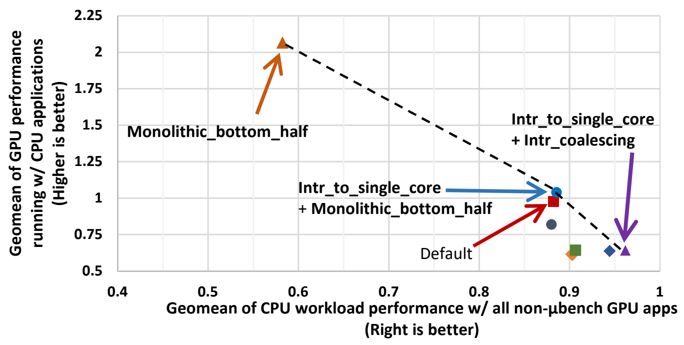


Fig. 8: Pareto trade-off between combinations of mitigation techniques for all non-microbenchmark programs.

benchmarks and our microbenchmark were often affected differently by the mitigation techniques, so we separated their analyses. We believe that Figure 8 is more representative of applications on current systems, while Figure 7 may better represent future systems with many accelerators. We note that the default configuration is not Pareto optimal in either chart.

If GPU SSR throughput is the primary goal, the monolithic SSR handler is a better choice. This design generally increases CPU overhead, but it increases GPU performance by over **2**×.

The combination of interrupt coalescing and steering results in the best CPU performance in both charts. Coalescing can reduce the number of SSRs interrupt to the CPU application, and steering those interrupts to a single core can further reduce the overhead. That said, this combination's effect on GPU performance depends on the GPU workload. Figure 8 shows that this causes the SSR-generating GPU applications to slow down by **35%** on average in order to gain **10%** more CPU performance. In contrast, Figure 7 shows that coalescing and steering speeds up our GPU microbenchmark by **45%** and the CPU workloads by **10%**. In the latter case, this configuration is obviously a better choice than the default, while the former may be better served by other Pareto optimal points.

Various other combinations can also be Pareto optimal, depending on the GPU benchmark. For example, combining all three techniques yields a good mix of CPU and GPU performance when running with the microbenchmark, while steering and a monolithic bottom half yield slightly better CPU

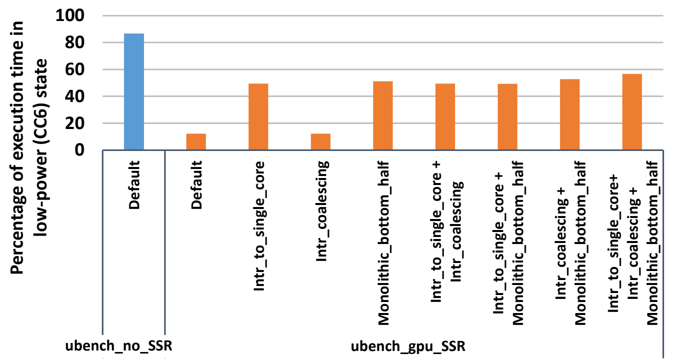


Fig. 9: Mitigation techniques affect CPU sleep states

and GPU performance for our other GPU benchmarks.

Many configurations yield GPU performance *above* that of running with idle CPUs because they reduce the latency of the SSR handler. However, we note that no mitigation strategies fully mitigate the problem of *host* interference from SSRs. Even the best-performing combination still results in CPU slowdowns of 5-20% compared to running the same applications with no SSRs. We return to this in Section VI.

E. Implication on Energy-efficiency

As discussed in Section IV-B, accelerator SSRs can severely degrade CPU sleep state residency; Figure 9 shows how the various mitigation techniques affect this.

The first bar shows the fraction of time the CPUs are asleep when ubench generates no SSRs (86%). The second shows that this drops to mere 12% when ubench generates SSRs. These are from Figure 4.

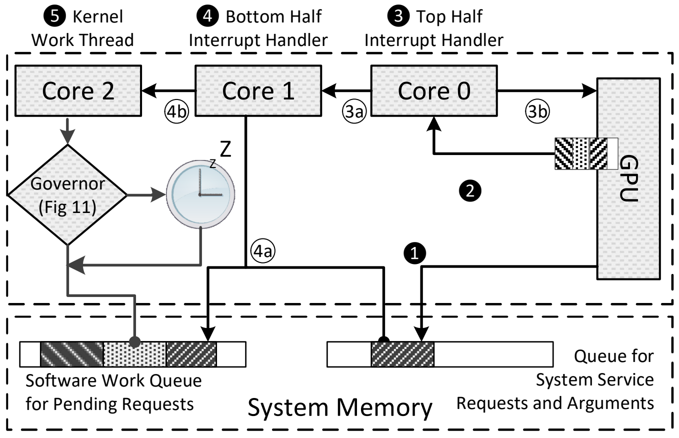
We observe that, except for interrupt coalescing, all other combinations of mitigation techniques significantly increase CPU sleep state residency and thus enhance energy efficiency. For example, deploying all three techniques together increases sleep time to **57%** from **12%**.

Sending interrupts to a single core raises the sleep state residency to 50%, because we send these interrupts to the same core that handles the bottom half. This essentially forces steps ③ and ④ in Figure 1 to run on the same core. The worker thread and bottom half handler run on two cores that stay awake, while the other two cores are able to sleep. Using a monolithic bottom half handler yields similar results due to the reduction in IPIs.

Interrupt coalescing by itself has little effect on sleep state residency, since multiple cores are still interrupted and awoken with IPIs. In addition, the best sleep state residency is still only 57%. Finding mechanisms that yield lower power in the face of GPU SSRs is an interesting area for further study.

VI. CPU QoS in the Presence of SSRs

Section IV showed that GPU SSRs degrade the performance of unrelated CPU-resident applications. Section V-D explored mitigation strategies that could reduce accelerator SSR interference, but none of these techniques were able to eliminate



5 Kernel Work Thread
4 Bottom Half Interrupt Handler
3 Top Half Interrupt Handler
Core 2
Core 1
Core 0
4b
3a
3b
Governor (Fig 11)
Z
2
GPU
4a
1
Queue for System Service Requests and Arguments
Software Work Queue for Pending Requests
System Memory

Fig. 10: Our QoS mechanism in the SSR handler

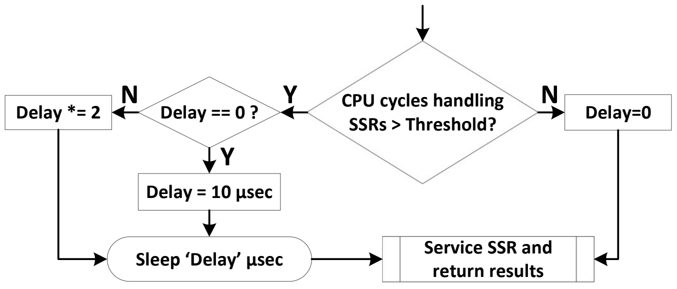
slowdown in the CPU applications. Even the best combination of techniques still resulted in CPU overheads of more than 20% in several cases. Thus, the trend of integrating ever-increasing numbers of accelerators into future heterogeneous systems makes SSR interference a potential bottleneck.

Importantly, in the absence of any quality-of-service (QoS) guarantees, it is possible to incur unbounded overhead due to SSR interference [44]. Malicious or buggy accelerators can potentially even use SSRs to mount denial-of-service attacks on the CPU [47]. It is therefore essential to build a QoS mechanism that can ensure a desired degree of performance isolation for CPU applications on a heterogeneous system.

Towards this end, we designed a software-based QoS mechanism that can arbitrarily limit CPU performance degradation from SSRs. Our technique requires no hardware modifications, and it is also orthogonal to (and can run in conjunction with) the techniques of Section V. To the best of our knowledge, *we are first to show the need for such a QoS mechanism in heterogeneous systems and also to build one in the OS.*

The key observation behind our QoS mechanism is that each accelerator has a hardware limit on the number of outstanding SSRs. This limit stems from the fact that accelerators must store state related to the SSR until it is serviced. This limit varies across accelerators, but its existence implies that it is possible to apply back-pressure to an accelerator by delaying the service of its SSRs. Ultimately this will stall the GPU until its SSRs are serviced. Further, if a program running on the accelerator depends on the results of its SSRs, then it may stall before reaching this hardware limit. Therefore, while GPUs run semi-independently from the CPU, this limit on outstanding SSRs can moderate SSR generation.

We thus extended the OS driver to allow the system administrator to specify the maximum amount of CPU time that may be spent processing GPU SSRs. A low value will ensure lower CPU performance loss at the cost of reduced SSR throughput and vice versa. Figures 10 and 11 illustrate how this is implemented (cf. unmodified Figure 3). The key to moderating the SSR rate is to delay processing of already-arrived SSRs when the amount of CPU time spent processing SSRs is higher than the desired rate. This delay will apply back-pressure to



CPU cycles handling SSRs > Threshold?
N
Delay=0
Y
Delay == 0 ?
N
Delay *= 2
Y
Delay = 10 µsec
Sleep 'Delay' µsec
Service SSR and return results

Fig. 11: Flow chart for the QoS governor

the GPU to stop generation of new SSR requests. Importantly, by adding delay in servicing SSRs instead of rejecting them outright, we can implement this technique without requiring any modification to how accelerators request SSRs. A governor decides whether to delay processing of SSRs according to the limit set by the system administrator.

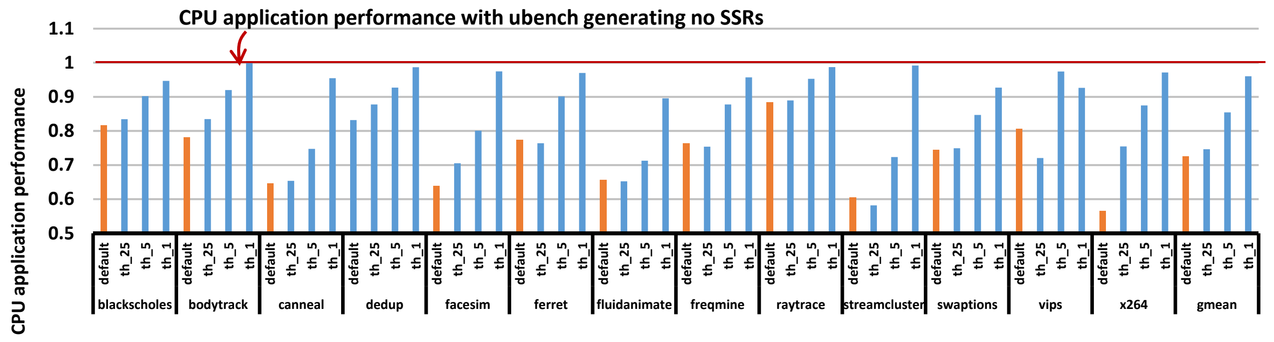
Our QoS mechanism has two parts. First, all OS routines involved in servicing SSRs are updated to account for their CPU cycles. This is then used by a kernel background thread that periodically (e.g., every 10μs) calculates if the fraction of cycles spent on servicing SSRs is over a specified limit.

Next, we modify the worker thread that processes SSRs as shown in the flowchart of Figure 10. The new worker thread first checks if the CPU time spent handling SSRs is above the specified threshold based on the information gathered by the background thread. If not, it sets the desired delay to zero and continues to process the SSR. Otherwise, it delays processing the SSRs following an exponential back-off starting at 10μs.

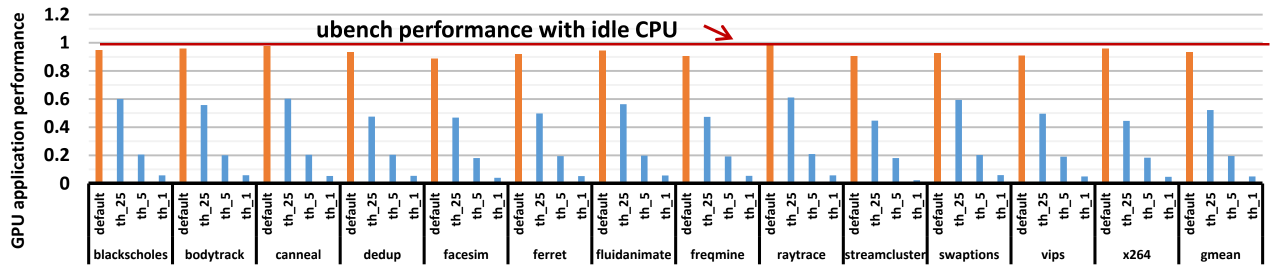
As the delay is increased, GPUs will begin to stall and the SSR rate will eventually drop. When the overhead falls below the set limit, the SSRs will be once again serviced without any artificial delay.

Figures 12a and 12b show the effect of this QoS on the performance of CPU applications and accelerator system services, respectively. Each cluster has a bar for the default configuration and for three different throttling values while running our GPU microbenchmark. The parameter *th_x* means the governor begins to throttle if more than *x*% of CPU time is spent servicing accelerator SSRs. For example, *th_1* attempts to cap the CPU overhead at 1%. The figures show this setting reduces the average CPU performance loss to less than 4% from 28%. However, it comes at a heavy cost to the throughput of the accelerator, which drops to a mere 5% of its unhindered throughput. Lower throttle values limit CPU overheads, but also reduce the throughput of the accelerator's requests. Note that even when threshold is set to x%, the CPU performance loss can be slightly more than x% because our driver enforces the limit periodically, rather than continuously.

Due to space constraints, we do not include data for all combinations of CPU applications, GPU workloads and thresholds. Similarly, we do not utilize any of the orthogonal techniques from Section V, in conjunction with the QoS. Figure 12 shows that our QoS mechanism is effective in limiting CPU overheads even when dealing with the aggressive stream of SSRs from our microbenchmark. This acts as a general proof



(a) CPU application performance in the presence of GPU SSRs while performing backpressure-based QoS. Higher is better.



(b) GPU application performance when concurrently running CPU application and performing backpressure-based QoS. Higher is better.

Fig. 12: Impact of different software throttling values

point of efficacy of our proposed QoS mechanism.

In summary, we implemented a simple but effective throttling technique in the OS that can help ensure QoS for CPU applications in face of GPU SSRs. While we focus on GPUs, as they are the most mature accelerators today, our observations and mitigation techniques are similarly applicable to other accelerators. Our current mechanism needs the system administrator to set the throttling rate. This can possibly be avoided by dynamically setting the throttling rate based on characteristics of the applications running at any given time. A shortcoming of the current framework is that it only guarantees performance for CPU applications. Future work can explore ways to implement QoS for GPU and other accelerators.

VII. Related work

Networking: The field of high-performance networking contains related works, since packet arrival interrupts have long been known to cause performance issues. For example, Mogul and Ramakrishnan recommend falling back to polling in the face many interrupts [44]. Polling for accelerator SSRs, however, could result in much higher relative CPU overheads.

A popular way to reduce interference due to networking is to use offload engines to process packets [43]. This works for very constrained tasks like processing packets but would not allow for generic, complex SSRs. It is infeasible to build an accelerator like TCP offload engines for *all* OS services.

A major area of overlap between our studies and networking is the use of interrupt steering [30] and coalescing [32]. Ahmad et al. determined how much to coalesce interrupts from networking and storage devices [6]; similar studies for accelerators are warranted.

We note that our QoS solution (Section VI) would not be feasible for networking, since incoming network packets are not necessarily susceptible to backpressure.

Jitter in HPC systems: Perhaps the closest analogue to an SSR offload engine is the Blue Gene/Q service processor [29]. This core runs the OS and deals with I/O to reduce timing jitter in scientific workloads. No consumer chip does such over-provisioning, since dedicating a core for SSRs may not be economically feasible.

León et al. recently showed that simultaneous multithreading (SMT) could be utilized in a similar way in HPC applications [42]. By assigning the OS to run on secondary threads, the authors were able to further reduce user-visible noise induced by OS routines.

Heterogeneous systems: Our study has implications on other heterogeneous systems research. Paul et al. showed that high-powered CPUs could cause nearby accelerators to run at lower frequencies due to thermal coupling [50]. Arora et al. [9] studied predicting upcoming CPU idleness in heterogeneous workloads to allow better sleep decisions. SSRs could affect both of these, implying that coordinated energy-management solutions should take SSRs into consideration [51].

VIII. Conclusion

We demonstrate that system service requests (SSRs) from a capable accelerator, like a GPU, can degrade performance of unrelated CPU applications by as much as 44%. Furthermore, SSR throughput can decrease by up to 18% due to concurrently running CPU applications. We then explored interference mitigation techniques that can reduce CPU overheads to 5% in the face of SSRs and yield faster SSRs than the default case with idle CPUs. We also demonstrated QoS techniques that allow configurable control of CPU overheads.

We believe that interference induced from accelerator-generated SSRs will become an increasingly important problem, and that more advanced QoS techniques are warranted.

References

[1] Advanced Micro Devices, Inc. Sea Islands Series Instruction Set Architecture. http://developer.amd.com/wordpress/media/2013/07/AMD_Sea_Islands_Instruction_Set_Architecture.pdf. Accessed: 2017-08-11.

[2] Advanced Micro Devices, Inc. AMD Unified Video Decoder (UVD). White Paper, June 2012.

[3] Advanced Micro Devices, Inc. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors*, February 2015.

[4] Advanced Micro Devices Inc. Linux IOMMU driver. http://elixir.free-electrons.com/linux/latest/source/drivers/iommu, 2017. Accessed: 2017-08-11.

[5] Agarwal, N., Nellans, D., O'Connor, M., Keckler, S. W., and Wenisch, T. F. Unlocking Bandwidth for GPUs in CC-NUMA Systems. In *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)* (2015).

[6] Ahmad, I., Gulati, A., and Mashtizadeh, A. vIC: Interrupt Coalescing for Virtual Machine Storage Device IO. In *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)* (2011).

[7] Aingaran, K., Jairath, S., and Lutz, D. Software in Silicon in the Oracle SPARC M7 Processor. Presented at Hot Chips, 2016.

[8] Appel, A. W., and Li, K. Virtual Memory Primitives for User Programs. In *Proc. of the Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1991).

[9] Arora, M., Manne, S., Paul, I., Jayasena, N., and Tullsen, D. M. Understanding Idle Behavior and Power Gating Mechanisms in the Context of Modern Benchmarks on CPU-GPU Integrated Systems. In *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)* (2015).

[10] Bienia, C., and Li, K. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proc. of the Workshop on Modeling, Benchmarking and Simulation* (June 2009).

[11] Bouvier, D., and Sander, B. Applying AMD's "Kaveri" APU for Heterogeneous Computing. Presented at Hot Chips, 2014.

[12] Bratt, I. HSA Queueing. Tutorial Presented at Hot Chips, 2013.

[13] Brooks, D., Hempstead, M., Lui, M., Mokri, P., Nilakantan, S., Reagen, B., and Shao, Y. S. Research Infrastructures for Accelerator-Centric Architectures. Tutorial Presented at HPCA, 2015.

[14] Brookwood, N. Everything You Always Wanted to Know About HSA But Were Afraid to Ask. White Paper, October 2013.

[15] CCIX Consortium. Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com/. Accessed: 2017-08-11.

[16] Che, S., Beckmann, B. M., Reinhardt, S. K., and Skadron, K. Pannotia: Understanding Irregular GPGPU Graph Applications. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)* (2013).

[17] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014).

[18] Clemons, J., Pellegrini, A., Savarese, S., and Austin, T. EVA: An Efficient Vision Architecture for Mobile Systems. In *Proc. of the Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (2013).

[19] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers*, 3rd ed. O'Reilly Media, Inc., 2005, ch. 10, p. 275.

[20] Daga, M., and Nutter, M. Exploiting Coarse-grained Parallelism in B+ Tree Searches on an APU. In *Proc. of the Workshop on Irregular Applications: Architectures & Algorithms (IA3)* (2012).

[21] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proc. of the Workshop on General-Purpose Computing on Graphics Processing Units (GPGPU)* (2010).

[22] Daoud, F., Watad, A., and Silberstein, M. GPUrdma: GPU-side Library for High Performance Networking from GPU Kernels. In *Proc. of the Int'l Workshop on Runtime and Operating Systems for Supercomputers (ROSS)* (2016).

[23] Ditty, M., Montrym, J., and Wittenbrink, C. Nvidia's Tegra K1 System-on-Chip. Presented at Hot Chips, 2014.

[24] Gen-Z Consortium. Gen-Z. http://genzconsortium.org/. Accessed: 2017-08-11.

[25] Gogte, V., Kolli, A., Cafarella, M. J., D'Antoni, L., and Wenisch, T. F. HARE: Hardware Accelerator for Regular Expressions. In *Proc. of the Int'l. Symp. on Microarchitecture (MICRO)* (2016).

[26] Gómez-Luna, J., Hajj, I. E., Chang, L.-W., García-Flores, V., de Gonzalo, S. G., Jablin, T. B., Peña, A. J., and Hwu, W. Chai: Collaborative Heterogeneous Applications for Integrated-architectures. In *Proc. of the Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)* (2017).

[27] Greathouse, J. L., Xin, H., Luo, Y., and Austin, T. A Case for Unlimited Watchpoints. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[28] Hammarlund, P. 4th Generation Intel® Core™ Processor, codenamed Haswell. Presented at Hot Chips, 2013.

[29] Haring, R. The Blue Gene/Q Compute Chip. Presented at Hot Chips, 2011.

[30] Herbert, T., and de Bruijn, W. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt. Accessed: 2017-08-11.

[31] HSA Foundation. Heterogeneous System Architecture (HSA). http://www.hsafoundation.com/. Accessed: 2017-08-11.

[32] Intel Corporation. Interrupt Moderation Using Intel® GbE Controllers, April 2007.

[33] Intel Corporation. Integrated Cryptographic and Compression Accelerators on Intel® Architecture Platforms. White Paper, 2013.

[34] Ioffe, R., Sharma, S., and Stoner, M. Achieving Performance with OpenCL 2.0 on Intel® Processor Graphics.

[35] Jennings, S. Linux's zswap Overview. https://www.kernel.org/doc/Documentation/vm/zswap.txt. Accessed: 2017-08-11.

[36] Jones, S. Introduction to Dynamic Parallelism. Presented at GPU Technology Conference (GTC), 2012.

[37] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)* (2017).

[38] Junkins, S. The Compute Architecture of Intel® Processor Graphics Gen8. Tech. rep., Intel Corporation, 2015.

[39] Kim, S., Huh, S., Hu, Y., Zhang, X., Witchel, E., Wated, A., and Silberstein, M. GPUnet: Networking Abstractions for GPU Programs. In *Proc. of the Conf. on Operating Systems Design and Implementation (OSDI)* (2014).

[40] Krishnan, G., Bouvier, D., Zhang, L., and Dongara, P. Energy Efficient Graphics and Multimedia in 28nm Carrizo APU. Presented at Hot Chips, 2015.

[41] Kyriazis, G. Heterogeneous System Architecture: A Technical Review. Tech. rep., Advanced Micro Devices, Inc., 2012.

[42] León, E. A., Karlin, I., and Moody, A. T. System Noise Revisited: Enabling Application Scalability and Reproducibility with SMT. In

Proc. of the Int'l Parallel and Distributed Processing Symp. (IPDPS) (2016).

[43] MOGUL, J. C. TCP Offload is a Dumb Idea Whose Time Has Come. In *Proc. of the Workshop on Hot Topics in Operating Systems* (2003).

[44] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proc. of the USENIX Annual Technical Conference (USENIX ATC)* (1996).

[45] NACHIAPPAN, N. C., ZHANG, H., RYOO, J., SOUNDARARAJAN, N., SIVASUBRAMANIAM, A., KANDEMIR, M. T., IYER, R., AND DAS, C. R. VIP: Virtualizing IP Chains on Handheld Platforms. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)* (2015).

[46] OLSON, L. E., HILL, M. D., AND WOOD, D. A. Crossing Guard: Mediating Host-Accelerator Coherence Interactions. In *Proc. of Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[47] OLSON, L. E., POWER, J., HILL, M. D., AND WOOD, D. A. Border Control: Sandboxing Accelerators. In *Proc. of the Int'l. Symp. on Microarchitecture (MICRO)* (2015).

[48] OLSON, L. E., SETHUMADHAVAN, S., AND HILL, M. D. Security Implications of Third-Party Accelerators. *IEEE Computer Architecture Letters (CAL) 15*, 1 (Jan 2016), 50–53.

[49] OPENCAPI CONSORTIUM. Open Coherent Accelerator Processor Interface (OpenCAPI). http://opencapi.org/. Accessed: 2017-08-11.

[50] PAUL, I., MANNE, S., ARORA, M., BIRCHER, W. L., AND YALAMANCHILI, S. Cooperative Boosting: Needy Versus Greedy Power Management. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)* (2013).

[51] PAUL, I., RAVI, V., MANNE, S., ARORA, M., AND YALAMANCHILI, S. Coordinated Energy Management in Heterogeneous Processors. In *Proc. of the Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC)* (2013).

[52] PUTNAM, A., CAULFIELD, A. M., CHUNG, E. S., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., HASELMAN, M., HAUCK, S., HEIL, S., HORMATI, A., KIM, J.-Y., LANKA, S., LARUS, J., PETERSON, E., POPE, S., SMITH, A., THONG, J., XIAO, P. Y., AND BURGER, D. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)* (2014).

[53] SILBERSTEIN, M., FORD, B., KEIDAR, I., AND WITCHEL, E. GPUfs: Integrating a File System with GPUs. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[54] SIVARAMAKRISHNAN, R., AND JAIRATH, S. Next Generation SPARC Processor Cache Hierarchy. Presented at Hot Chips, 2014.

[55] SOARES, L., AND STUMM, M. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proc. of the Conf. on Operating Systems Design and Implementation (OSDI)* (2010).

[56] SPLIET, R., HOWES, L., AND VARBANESCU, A. L. KMA: A Dynamic Memory Manager for OpenCL. In *Proc. of the Workshop on General Purpose Processing Using GPUs (GPGPU)* (2014).

[57] STUECHELI, J., BLANER, B., JOHNS, C. R., AND SIEGEL, M. S. CAPI: A Coherent Accelerator Processor Interfaces. *IBM Journal of Research and Development 59*, 1 (January/February 2015), 7:1–7:7.

[58] SUN, Y., GONG, X., ZIABARI, A. K., YU, L., LI, X., MUKHERJEE, S., MCCARDWELL, C., VILLEGAS, A., AND KAELI, D. Hetero-Mark, a Benchmark Suite for CPU-GPU Collaborative Computing. In *Proc. of the IEEE Int'l Symp on Workload Characterization (IISWC)* (2016).

[59] TRAMM, J. R., SIEGEL, A. R., ISLAM, T., AND SCHULZ, M. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future* (2014).

[60] VESELÝ, J., BASU, A., BHATTACHARJEE, A., LOH, G. H., OSKIN, M., AND REINHARDT, S. K. Generic System Calls for GPUs. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)* (2018).

[61] VESELÝ, J., BASU, A., OSKIN, M., LOH, G., AND BHATTACHARJEE, A. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *Proc. of the Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)* (2016).

[62] WILCOX, M. I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers. Presented at linux.conf.au, 2003.

[63] WU, L., LOTTARINI, A., PAINE, T. K., KIM, M. A., AND ROSS, K. A. Q100: The Architecture and Design of a Database Processing Unit. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014).