Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators

Konstantinos Menychtas Kai Shen Michael L. Scott

University of Rochester

{kmenycht,kshen,scott}@cs.rochester.edu

Abstract

Today's operating systems treat GPUs and other computational accelerators as if they were simple devices, with bounded and predictable response times. With accelerators assuming an increasing share of the workload on modern machines, this strategy is already problematic, and likely to become untenable soon. If the operating system is to enforce fair sharing of the machine, it must assume responsibility for accelerator scheduling and resource management.

Fair, safe scheduling is a particular challenge on fast accelerators, which allow applications to avoid kernel-crossing overhead by interacting directly with the device. We propose a *disengaged* scheduling strategy in which the kernel intercedes between applications and the accelerator on an infrequent basis, to monitor their use of accelerator cycles and to determine which applications should be granted access over the next time interval.

Our strategy assumes a well defined, narrow interface exported by the accelerator. We build upon such an interface, systematically inferred for the latest Nvidia GPUs. We construct several example schedulers, including *Disengaged Timeslice* with overuse control that guarantees fairness and *Disengaged Fair Queueing* that is effective in limiting resource idleness, but probabilistic. Both schedulers ensure fair sharing of the GPU, even among uncooperative or adversarial applications; Disengaged Fair Queueing incurs a 4% overhead on average (max 18%) compared to direct device access across our evaluation scenarios.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Other Architecture Styles; D.4.1 [*Operating Systems*]: Process Management—Scheduling; D.4.8 [*Operating Systems*]: Performance

Keywords Operating system protection; scheduling; fairness; hardware accelerators; GPUs

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541963

1. Introduction

Future microprocessors seem increasingly likely to be highly heterogeneous, with large numbers of specialized computational accelerators. General-purpose GPUs are the canonical accelerator on current machines; other current or near-future examples include compression, encryption, XML parsing, and media transcoding engines, as well as reconfigurable (FPGA) substrates.

Current OS strategies, which treat accelerators as if they were simple devices, are ill suited to operations of unpredictable, potentially unbounded length. Consider, for example, a work-conserving GPU that alternates between requests (compute "kernels," rendering calls) from two concurrent applications. The application with larger requests will tend to receive more time. A greedy application may intentionally "batch" its work into larger requests to hog resources. A malicious application may launch a denial-of-service attack by submitting a request with an infinite loop.

Modern accelerator system architectures (Figure 1) pose at least two major challenges to fair, safe management. First, microsecond-level request latencies strongly motivate designers to avoid the overhead of user/kernel domain switching (up to 40% for small requests according to experiments later in this paper) by granting applications direct device access from user space. In bypassing the operating system, this direct access undermines the OS's traditional responsibility for protected resource management. Second, the application-hardware interface for accelerators is usually hidden within a stack of black-box modules (libraries, run-time systems, drivers, hardware), making it inaccessible to programmers outside the vendor company. To enable resource management in the protected domain of the OS kernel, but outside such black-box stacks, certain aspects of the interface must be disclosed and documented.

To increase fairness among applications, several research projects have suggested changes or additions to the application-library interface [12, 15, 19, 20, 32], relying primarily on open-source stacks for this purpose. Unfortunately, as discussed in Section 2, such changes invariably serve either to replace the direct-access interface, adding a per-request syscall overhead that can be significant, or to impose a voluntary level of cooperation in front of that interface. In this latter case, nothing prevents an application

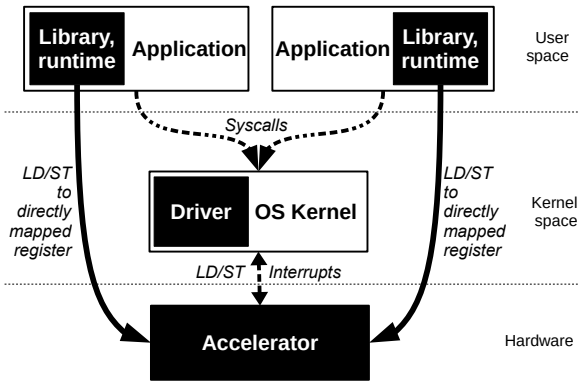

Figure 1. For efficiency, accelerators (like Nvidia GPUs) receive requests directly from user space, through a memory-mapped interface. The syscall-based OS path is only used for occasional maintenance such as the initialization setup. Commonly, much of the involved software (libraries, drivers) and hardware is unpublished (black boxes).

from ignoring the new interface and accessing the device directly. Simply put, the current state of the art with respect to accelerator management says "you can have at most two of protection, efficiency, and fairness—not all three."

While we are not the first to bemoan this state of affairs [29], we are, we believe, the first to suggest a viable solution, and to build a prototype system in which accelerator access is simultaneously protected, efficient, and fair. Our key idea, called *disengagement*, allows an OS resource scheduler to maintain fairness by interceding on a small number of acceleration requests (by disabling the direct-mapped interface and intercepting resulting faults) while granting the majority of requests unhindered direct access. To demonstrate the versatility and efficiency of this idea, we build a prototype: a Linux kernel module that enables disengaged scheduling on three generations of Nvidia GPUs. We construct our prototype for the proprietary, unmodified software stack. We leverage our previously proposed state machine inferencing approach to uncover the black-box GPU interface [25] necessary for scheduling.

We have built and evaluated schedulers that illustrate the richness of the design space. Our *Disengaged Timeslice* scheduler ensures fully protected fair sharing among applications, but can lead to underutilization when applications contain substantial "off" periods between GPU requests. Our *Disengaged Fair Queueing* scheduler limits idleness and maintains high efficiency with a strong probabilistic guarantee of fairness. We describe how we build upon the uncovered GPU interface; how we enable protection, efficiency, and fairness; and how hardware trends may influence—and be influenced by—our scheduling techniques.

2. Related Work

Some GPU management systems—e.g., PTask [32] and Pegasus [15]—provide a replacement user-level library that arranges to make a system or hypervisor call on each GPU request. In addition to the problem of per-request overhead, this approach provides safety from erroneous or malicious applications only if the vendor's direct-mapped interface is disabled—a step that may compromise compatibility with pre-existing applications. Elliott and Anderson [12] require an application to acquire a kernel mutex before using the GPU, but this convention, like the use of a replacement library, cannot be enforced. Though such approaches benefit from the clear semantics of the libraries' documented APIs, they can be imprecise in their accounting: API calls to start acceleration requests at this level are translated asynchronously to actual resource usage by the driver. We build our solution at the GPU software / hardware interface, where the system software cannot be circumvented and resource utilization accounting can be timely and precise.

Other systems, including GERM [11], TimeGraph/Gdev [19, 20] and LoGV [13] in a virtualized setting, replace the vendor black-box software stack with custom open-source drivers and libraries (e.g., Nouveau [27] and PathScale [31]). These enable easy operating system integration, and can enforce an OS scheduling policy if configured to be notified of all requests made to the GPU. For small, frequent acceleration requests, the overhead of trapping to the kernel can make this approach problematic. The use of a custom, modified stack also necessitates integration with user-level GPU libraries; for several important packages (e.g., OpenCL, currently available only as closed source), this is not always an option. Our approach is independent of the rest of the GPU stack: it intercedes at the level of the hardware-software interface. (For prototyping purposes, this interface is uncovered through systematic reverse engineering [25]; for production systems it will require that vendors provide a modest amount of additional documentation.)

Several projects have addressed the issue of fairness in GPU scheduling. GERM [11] enables fair-share resource allocation using a deficit round-robin scheduler [34]. TimeGraph [19] supports fairness by penalizing overuse beyond a reservation. Gdev [20] employs a non-preemptive variant of Xen's Credit scheduler to realize fairness. Beyond the realm of GPUs, fair scheduling techniques (particularly the classic fair queueing schedulers [10, 14, 18, 30, 33]) have been adopted successfully in network and I/O resource management. Significantly, all these schedulers track and/or control each individual device request, imposing the added cost of OS kernel-level management on fast accelerators that could otherwise process requests in microseconds. Disengaged scheduling enables us to redesign existing scheduling policies to meet fast accelerator requirements, striking a better balance between protection and efficiency while maintaining good fairness guarantees.

Beyond the GPU, additional computational accelerators are rapidly emerging for power-efficient computing through increased specialization [8]. The IBM PowerEN proces-

sor [23] contains accelerators for tasks (cryptography, compression, XML parsing) of importance for web services. Altera [1] provides custom FPGA circuitry for OpenCL. The faster the accelerator, and the more frequent its use, the greater the need for direct, low-latency access from user applications. To the extent that such access bypasses the OS kernel, protected resource management becomes a significant challenge. The ongoing trend toward deeper CPU/accelerator integration (e.g., AMD's APUs [7, 9] or HSA architecture [24], ARM's Mali GPUs [4], and Intel's QuickAssist [17] accelerator interface) and low-overhead accelerator APIs (e.g. AMD's Mantle [3] for GPUs), leads us to believe that disengaged scheduling can be an attractive strategy for future system design.

Managing fast devices for which per-request engagement is too expensive is reminiscent of the Soft Timers work [5]. Specifically, Aron and Druschel argued that per-packet interrupts are too expensive for fast networks and that batched processing through rate-based clocking and network polling is necessary for high efficiency [5]. Exceptionless system calls [35] are another mechanism to avoid the overhead of frequent kernel traps by batching requests. Our proposal also avoids per-request manipulation overheads to achieve high efficiency, but our specific techniques are entirely different. Rather than promote batched and/or delayed processing of system interactions, which co-designed user-level libraries already do for GPUs [21, 22, 28], we aim for synchronous involvement of the OS kernel in resource management, at times and conditions of its choosing.

Recent projects have proposed that accelerator resources be managed in coordination with conventional CPU scheduling to meet overall performance goals [15, 29]. Helios advocated the support of affinity scheduling on heterogeneous machines [26]. We recognize the value of coordinated resource management—in fact we see our work as enabling it: for accelerators that implement low-latency requests, coordinated management will require OS-level accelerator scheduling that is not only fair but also safe and fast.

3. Protected Scheduling with Efficiency

From the kernel's perspective, and for the purpose of scheduling, accelerators can be thought of as processing resources with an event-based interface. A "task" (the resource principal to which we wish to provide fair service—e.g., a process or virtual machine) can be charged for the resources occupied by its "acceleration request" (e.g., compute, graphics). To schedule such requests, we need to know when they are submitted for processing and when they complete.

Submission and completion are easy to track if the former employs the syscall interface and the latter employs interrupts. However, and primarily for reasons of efficiency, vendors like Nvidia avoid both, and build an interface that is directly mapped to user space instead. A request can be submitted by initializing buffers and a descriptor object in shared memory and then writing the address of the descriptor into a queue that is watched by the device. Likewise, completion can be detected by polling a flag or counter that is set by the device.

Leveraging our previous work on state-machine inferencing of the black-box GPU interface [25], we observe that the kernel can, when desired, learn of requests passed through the direct-mapped interface by means of *interception*—by (temporarily) unmapping device registers and catching the resulting page faults. The page fault handler can then pass control to an OS resource manager that may dispatch or queue/delay the request according to a desired policy. Request completion can likewise be realized through independent polling. In comparison to relying on a modified user-level library, interception has the significant advantage of enforcing policy even on unmodified, buggy, or selfish / malicious applications. All it requires of the accelerator architecture is a well-defined interface based on scheduling events—an interface that is largely independent of the internal details of black or white-box GPU stacks.

Cost of OS management Unfortunately, trapping to the OS—whether via syscalls or faults—carries nontrivial costs. The cost of a user / kernel mode switch, including the effects of cache pollution and lost user-mode IPC, can be thousands of CPU cycles [35, 36]. Such cost is a substantial addition to the base time (305 cycles on our Nvidia GTX670-based system) required to submit a request directly with a write to I/O space. In applications with small, frequent requests, high OS protection costs can induce significant slowdown. Figure 2 plots statistics for three realistic applications, selected for their small, frequent requests: glxgears (standard OpenGL [22] demo), Particles (OpenCL / GL from CUDA SDK [28]), and Simple 3D Texture (OpenCL / GL from CUDA SDK [28]). More than half of all GPU requests from these applications are submitted and serviced in less than 10 μs. With a deeply integrated CPU-GPU microarchitecture, request latency could be lowered even more—to the level of a simple memory/cache write/read, further increasing the relative burden of interception.

To put the above in perspective, we compared the throughput that can be achieved with an accelerator software / hardware stack employing a direct-access interface (Nvidia Linux binary driver v310.14) and with one that relies on traps to the kernel for every acceleration request (AMD Catalyst v13.8 Linux binary driver). The comparison uses the same machine hardware, and we hand-tune the OpenCL requests to our Nvidia (GTX670) or AMD (Radeon HD 7470) PCIe GPU such that the requests are equal-sized. We discover that, for requests in the order of 10–100 μs, a throughput increase of 8–35% can be expected by simply accessing the device directly, without trapping to the kernel for every request. These throughput gains are even higher (48–170%) if the traps to the kernel actually entail nontrivial processing in GPU driver routines.

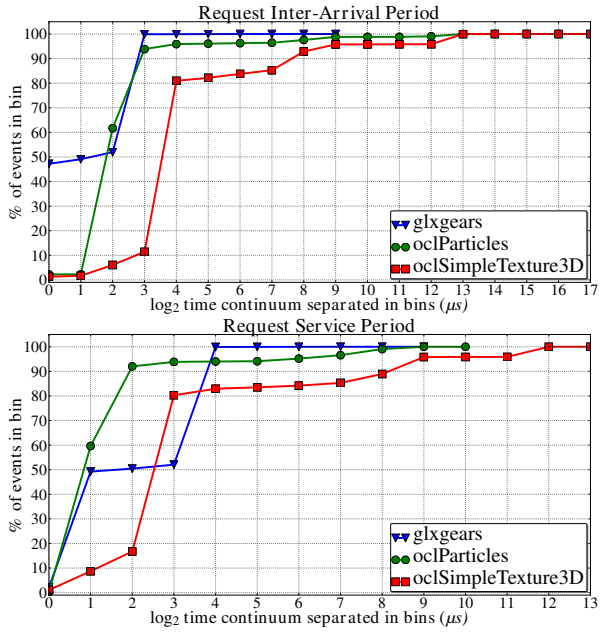

Figure 2. CDFs of request frequency and average service times for several applications on an Nvidia GTX670 GPU. A large percentage of arriving requests are short and submitted in short intervals ("back-to-back").

Overview of schedulers Sections 3.1, 3.2, and 3.3 present new request schedulers for fast accelerators that leverage interception to explicitly balance protection, fairness, and efficiency. The *Timeslice* scheduler retains the overhead of per-request kernel intervention, but arranges to share the GPU fairly among unmodified applications. The *Disengaged Timeslice* scheduler extends this design by eliminating kernel intervention in most cases. Both variants allow only one application at a time to use the accelerator. The *Disengaged Fair Queueing* scheduler eliminates this limitation, but provides only statistical guarantees of fairness.

Our disengaged schedulers maintain the high performance of the direct-mapped accelerator interface by limiting kernel involvement to a small number of interactions. In so doing, they rely on information about aspects of the accelerator design that can affect resource usage and accounting accuracy. For the purposes of the current description, we assume the characteristics of an Nvidia GPU, as revealed by reverse engineering and public documentation. We describe how device features affect our prototype design as we encounter them in the algorithm descriptions; we revisit and discuss how future software / hardware developments can influence and be influenced by our design in Section 6.

3.1 Timeslice with Overuse Control

To achieve fairness in as simple a manner as possible, we start with a standard token-based timeslice policy. Specifically, a token that indicates permission to access the shared accelerator is passed among the active tasks. Only the current token holder is permitted to submit requests, and all request events (submission, completion) are trapped and communicated to the scheduler. For the sake of throughput, we permit overlapping nonblocking requests from the task that holds the token.

Simply preventing a task from submitting requests outside its timeslice does not suffice for fairness. Imagine, for example, a task that issues a long series of requests, each of which requires 0.9 timeslice to complete. A naive scheduler might allow such a task to issue two requests in each timeslice, and to steal 80% of each subsequent timeslice that belongs to some other task. To avoid this overuse problem, we must, upon the completion of requests that overrun the end of a timeslice, deduct their excess execution time from future timeslices of the submitting task. We account for overuse by waiting at the end of a timeslice for all outstanding requests to complete, and charging the token holder for any excess. When a task's accrued overuse exceeds a full timeslice, we skip the task's next turn to hold the token, and subtract a timeslice from its accrued overuse.

We also need to address the problem that a single long request may monopolize the accelerator for an excessive amount of time, compromising overall system responsiveness. In fact, Turing-complete accelerators, like GPUs, may have to deal with malicious (or poorly written) applications that submit requests that never complete. In a timeslice scheduler, it is trivial to identify the task responsible for an over-long request: it must be the last token holder. Subsequent options for restoring responsiveness then depend on accelerator features.

From model to prototype Ideally, we would like to preempt over-long requests, or define them as errors, kill them on the accelerator, and prevent the source task from sending more. Lacking vendor support for preemption or a documented method to kill a request, we can still protect against over-long requests by killing the offending task, provided this leaves the accelerator in a clean state. Specifically, upon detecting that overuse has exceeded a predefined threshold, we terminate the associated OS process and let the existing accelerator stack follow its normal exit protocol, returning occupied resources back to the available pool. While the involved cleanup depends on (undocumented) accelerator features, it appears to work correctly on modern GPUs.

Limitations Our Timeslice scheduler can guarantee fair resource use in the face of unpredictable (and even unbounded) request run times. It suffers, however, from two efficiency drawbacks. First, its fault-based capture of each submitted request incurs significant overhead on fast accelerators. Second, it is not *work-conserving*: the accelerator may be idled during the timeslice of a task that temporarily has no requests to submit—even if other tasks are waiting for accelerator access. Our *Disengaged Timeslice* scheduler successfully tackles the overhead problem. Our *Disengaged Fair Queueing* scheduler also tackles the problem of involuntary resource idleness.

3.2 Disengaged Timeslice

Fortunately, the interception of every request is not required for timeslice scheduling. During a timeslice, direct, unmonitored access from the token holder task can safely be allowed. The OS will trap and delay accesses from all other tasks. When passing the token between tasks, the OS will update page tables to enable or disable direct access. We say such a scheduler is largely *disengaged*; it allows fast, direct access to accelerators most of the time, interceding (and incurring cost) on an infrequent basis only. Note that while requests from all tasks other than the token holder will be trapped and delayed, such tasks will typically stop running as soon as they wait for a blocking request. Large numbers of traps are thus unlikely.

When the scheduler re-engages at the start of a new timeslice, requests from the token holder of the last timeslice may still be pending. To account for overuse, we must again wait for these requests to finish.

From model to prototype Ideally, the accelerator would expose information about the status of request(s) on which it is currently working, in a location accessible to the scheduler. While we strongly suspect that current hardware has the capability to do this, it is not documented, and we have been unable to deduce it via reverse engineering. We have, however, been able to discover the semantics of data structures shared between the user and the GPU [2, 16, 19, 25, 27]. These structures include a reference counter that is written by the hardware upon the completion of each request. Upon re-engaging, we can traverse in-memory structures to find the reference number of the last submitted request, and then poll the reference counter for an indication of its completion. While it is at least conceivable that a malicious application could spoof this mechanism, it suffices for performance testing; we assume that a production-quality scheduler would query the status of the GPU directly.

Limitations Our Disengaged Timeslice scheduler does not suffer from high per-request management costs, but it may still lead to poor utilization when the token holder is unable to keep the accelerator busy. This problem is addressed by the third of our example schedulers.

3.3 Disengaged Fair Queueing

Disengaged resource scheduling is a general concept—allow fast, direct device access in the common case; intercede only when necessary to efficiently realize resource management objectives. Beyond Disengaged Timeslice, we also develop a disengaged variant of the classic fair queueing algorithm, which achieves fairness while maintaining work-conserving properties—i.e., it avoids idling the resource when there is pending work to do.

A standard fair queueing scheduler assigns a *start tag* and a *finish tag* to each resource request. These tags serve as surrogates for the request-issuing task's cumulative resource usage before and after the request's execution. Specifically, the start tag is the larger of the current system virtual time (as of request submission) and the finish tag of the most recent previous request by the same task. The finish tag is the start tag plus the expected resource usage of the request. Request dispatch is ordered by each pending request's finish tag [10, 30] or start tag [14, 18, 33]. Multiple requests (potentially from different tasks) may be dispatched to the device at the same time [18, 33]. While giving proportional resource uses to active tasks, a fair queueing scheduler addresses the concern that an inactive task—one that currently has no work to do—might build up its resource credit without bound and then reclaim it in a sudden burst, causing prolonged unresponsiveness to others. To avoid such a scenario, the scheduler advances the system virtual time to reflect only the progress of active tasks. Since the start tag of each submitted request is brought forward to at least the system virtual time, any claim to resources from a task's idle period is forfeited.

Design For the sake of efficiency, our disengaged scheduler avoids intercepting and manipulating most requests. Like other fair queueing schedulers, however, it does track cumulative per-task resource usage and system-wide virtual time. In the absence of per-request control, we replace the request start/finish tags in standard fair queueing with a probabilistically-updated per-task virtual time that closely approximates the task's cumulative resource usage. Time values are updated using statistics obtained through periodic engagement. Control is similarly coarse grain: the interfaces of tasks that are running ahead in resource usage may be disabled for the interval between consecutive engagements, to allow their peers to catch up. Requests from all other tasks are allowed to run freely in the disengaged time interval.

Conceptually, our design assumes that at each engagement we can acquire knowledge of each task's active/idle status and its resource usage in the previous time interval. Actions then taken at each engagement are as follows:

1. We advance each active task's virtual time by adding its resource use in the last interval. We then advance the system-wide virtual time to be the oldest virtual time among the active tasks.
2. If an inactive task's virtual time is behind the system virtual time, we move it forward to the system virtual time. As in standard fair queueing, this prevents an inactive task from hoarding unused resources.
3. For the subsequent time interval, up to the next engagement, we deny access to any tasks whose virtual time is ahead of the system virtual time by at least the length of the interval. That is, even if the slowest active task acquires exclusive resource use in the upcoming interval, it will at best catch up with the tasks whose access is being denied.

Compared to standard fair queueing, which captures and controls each request, our statistics maintenance and control are both coarse grained. Imbalance will be remedied only

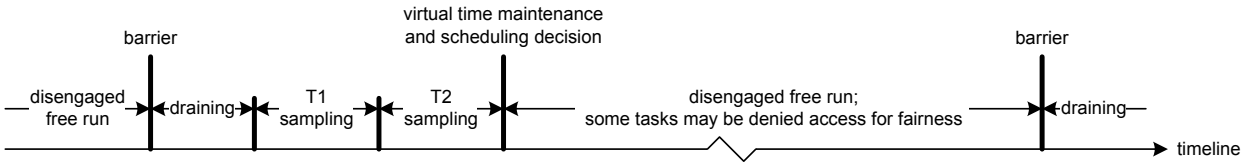

Figure 3. An illustration of periodic activities in Disengaged Fair Queueing. An engagement episode starts with a barrier and the draining of outstanding requests on the GPU, which is followed by the short sampling runs of active tasks (two tasks T_1 and T_2 in the case), virtual time maintenance and scheduling decision, and finally a longer disengaged free-run period.

when it exceeds the size of the inter-engagement interval. Furthermore, the correctness of our approach depends on accurate statistics gathering, which can be challenging in the absence of hardware assistance. We discuss this issue next.

From model to prototype Ideally, an accelerator would maintain and export the statistics needed for Disengaged Fair Queueing, in a location accessible to the scheduler. It could, for example, maintain cumulative resource usage for each task, together with an indication of which task(s) have requests pending and active. Lacking such hardware assistance, we have developed a software mechanism to estimate the required statistics. In general, accelerators (like our GPU) accept processing requests through independent queues mapped in each task's address space, cycling round-robin among queues with pending requests and processing each request separately, one after the other. Most times, there is (or can be assumed) a one-to-one mapping between tasks and queues, so the resource share attributed to each task in a given time interval is proportional to the average request run-time experienced from its requests queue. The design of our software mechanism relies on this observation. To obtain the necessary estimates, we cycle among tasks during a period of re-engagement, providing each with exclusive access to the accelerator for a brief interval of time and actively monitoring all of its requests. This interval is not necessarily of fixed duration: we collect either a predefined number of requests deemed sufficient for average estimation, or as many requests as can be observed in a predefined maximum interval length—whichever occurs first. As in the timeslice schedulers, we place a (documented) limit on the maximum time that any request is permitted to run, and kill any task that exceeds this limit.

Figure 3 illustrates key activities in Disengaged Fair Queueing. Prior to sampling, a barrier stops new request submission in every task, and then allows the accelerator to drain active requests. (For this we employ the same reference-counter-based mechanism used to detect overuse at the end of a timeslice in Section 3.2.) After sampling, we disengage for another free-run period. Blocking new requests while draining is free, since the device is known to be busy. Draining completes immediately if the device is found not to be working on queued requests at the barrier.

Because an engagement period includes a sampling run for every active task, its aggregate duration can be non-negligible. The disengaged free-run period has to be substantially longer to limit the draining and sampling cost. To ensure this, we set the free-run period to be several times the duration of the last engagement period.

Limitations The request-size estimate developed during the sampling period of Disengaged Fair Queueing acts as a resource-utilization proxy for the whole task's free-run period. In the common case, this heuristic is accurate, as accelerated applications tend to employ a single request queue and the GPU appears to schedule among queues with outstanding requests in a round-robin fashion. It is possible, however, to have applications that not only maintain multiple queues (e.g. separate compute and graphics rendering queues), but also keep only some of them active (i.e. holding outstanding requests) at certain times. When at least one task among co-runners behaves this way, estimating the average request size on every queue might not be enough for accurate task scheduling. In principle, better estimates might be based on more detailed reverse engineering of the GPU's internal scheduling algorithm. For now, we are content to assume that each task employs a single queue, in the hope that production systems will have access to statistics that eliminate the need for estimation.

Our estimation mechanism also assumes that observed task behavior during an engagement interval reflects behavior in the previous disengaged free-run. However, this estimate can be imprecise, resulting in imperfect fairness. One potential problem is that periodic behavior patterns—rather than overall task behavior—might be sampled. The non-fixed sampling period run time introduces a degree of randomness in the sampling frequency and duration, but further randomizing these parameters would increase confidence in our ability to thwart any attempt by a malicious application to subvert the scheduling policy. If the disengaged free-run period is considered to be too long (e.g., because it dilates the potential time required to identify and kill an infinite-loop request), we could shorten it by sampling only a subset of the active tasks in each engagement-disengagement cycle.

Our implementation of the Disengaged Fair Queueing scheduler comes very close to the work-conserving goal of not idling the resource when there is pending work to do. Specifically, it allows multiple tasks to issue requests in the dominant free-run periods. While exclusive access is enforced during sampling intervals, these intervals are

bounded, and apply only to tasks that have issued requests in the preceding free-run period; we do not waste sampling time on idle tasks. Moreover, sampling is only required in the absence of true hardware statistics, which we have already argued that vendors could provide.

The more serious obstacle to true work conservation is the possibility that all tasks that are granted access during a disengaged period will, by sheer coincidence, become simultaneously idle, while tasks that were not granted access (because they had been using more than their share of accumulated time) still have work to do. Since our algorithm can be effective in limiting idleness, even in the presence of "selfish," if not outright malicious applications, and on the assumption that troublesome scenarios will be rare and difficult to exploit, the degree to which our Disengaged Fair Queueing scheduler falls short of being fully work conserving seems likely to be small.

4. Prototype Implementation

Our prototype of interception-based OS-level GPU scheduling, which we call NEON, is built on the Linux 3.4.7 kernel. It comprises an autonomous kernel module supporting our request management and scheduling mechanisms (about 8000 lines of code); the necessary Linux kernel hooks (to `ioctl`, `mmap`, `munmap`, `copy_task`, `exit_task`, and `do_page_fault`—less than 100 lines); kernel-resident control structs (about 500 lines); and a few hooks to the Nvidia driver's binary interface (initialization, `ioctl`, `mmap` requests). Some of the latter hooks reside in the driver's binary-wrapper code (less than 100 lines); others, for licensing reasons, are inside the Linux kernel (about 500 lines).

Our request management code embodies a state-machine model of interaction among the user application, driver, and device [25], together with a detailed understanding of the structure and semantics of buffers shared between the application and device. Needed information was obtained through a combination of publicly available documentation and reverse engineering [2, 16, 19, 25, 27].

From a functional perspective, NEON comprises three principal components: an initialization phase, used to identify the virtual memory areas of all memory-mapped device registers and buffers associated with a given *channel* (a GPU request queue and its associated software infrastructure); a page-fault–handling mechanism, used to catch the device register writes that constitute request submission; and a polling-thread service, used within the kernel to detect reference-counter updates that indicate completion of previously submitted requests. The page-fault–handling mechanism and polling-thread service together define a kernel-internal interface through which any event-based scheduler can be coupled to our system.

The goal of the initialization phase is to identify the device, task, and GPU *context* (address space) associated with every channel, together with the location of three key virtual memory areas (VMAs). The GPU context encapsulates channels whose requests may be causally related (e.g., issued by the same process); NEON avoids deadlocks by avoiding re-ordering of requests that belong to the same context. The three VMAs contain the *command buffer*, in which requests are constructed; the *ring buffer*, in which pointers to consecutive requests are enqueued; and the *channel register*, the device register through which user-level libraries notify the GPU of ring buffer updates, which represent new requests. As soon as the NEON state machine has identified all three of these VMAs, it marks the channel as "active"—ready to access the channel registers and issue requests to the GPU.

During engagement periods, the page-fault–handling mechanism catches the submission of requests. Having identified the page in which the channel register is mapped, we protect its page by marking it as non-present. At the time of a fault, we scan through the buffers associated with the channel to find the location of the reference counter that will be used for the request, and the value that will indicate its completion. We then map the location into kernel space, making it available to the polling-thread service. The page-fault handler runs in process context, so it can pass control to the GPU scheduler, which then decides whether the current process should sleep or be allowed to continue and access the GPU. After the application is allowed to proceed, we single-step through the faulting instruction and mark the page again as "non-present."

Periodically, or at the scheduler's prompt, the polling-thread service iterates over the kernel-resident structures associated with active GPU requests, looking for the address/value pairs that indicate request completion. When an appropriate value is found—i.e., when the last submitted command's target reference value is read at the respective polled address—the scheduler is invoked to update accounting information and to block or unblock application threads as appropriate.

The page-fault–handling mechanism allows us to capture and, if desired, delay request submission. The polling-thread service allows us to detect request completion and, if desired, resume threads that were previously delayed. These request interception and completion notification mechanisms support the implementation of all our schedulers.

Disengagement is implemented by letting the scheduling policy choose if and when it should restore protection on channel register VMAs. A barrier requires that all the channel registers in use on the GPU be actively tracked for new requests—thus their respective pages must be marked "non-present." It is not possible to miss out on newly established channel mappings while disengaged: all requests to establish such a mapping are captured as syscalls or invocations of kernel-exposed functions that we monitor.

Though higher level API calls may be processed out of order, requests arriving at the same channel are always pro-

cessed in order by the GPU. We rely on this fact in our post–re-engagement status update mechanism, to identify the current status of the last submitted request. Specifically, we inspect the values of reference counters and the reference number values for the last submitted request in every active channel. Kernel mappings have been created upon first encounter for the former, but the latter do not lie at fixed addresses in the application's address space. To access them safely, we scan through the command queue to identify their locations and build temporary memory mappings in the kernel's address space. Then, by walking the page table, we can find and read the last submitted commands' reference values.

5. Experimental Evaluation

NEON is modularized to accommodate different devices; we have deployed it on several Nvidia GPUs. The most recent is a GTX670, with a "Kepler" microarchitecture (GK104 chip core) and 2 GB RAM. Older tested systems include the GTX275 and NVS295, with the "Tesla" microarchitecture (200b and G98 chip cores, respectively), and the GTX460, with the "Fermi" microarchitecture (F104 chip core). We limit our presentation here to the Kepler GPU, as its modern software / hardware interface (faster context switching among channels, faster memory subsystem) lets us focus on the cost of scheduling small, frequent requests. Our host machine has a 2.27 GHz Intel Xeon E5520 CPU (Nehalem microarchitecture) and triple-channel 1.066 GHz DDR3 RAM.

5.1 Experimental Workload

NEON is agnostic with respect to the type of workload: it can schedule compute or graphics requests, or even DMA requests. The performance evaluation in this paper includes compute, graphics, and combined (CUDA or OpenCL plus OpenGL) applications, but is primarily focused on compute requests—they are easy to understand, are a good proxy for broad accelerator workloads, and can capture generic, possibly unbounded requests. Our experiments employ the Nvidia 310.14 driver and CUDA 5.0 libraries, which support acceleration for OpenCL 1.1 and OpenGL 4.2.

Our intent is to focus on behavior appropriate to emerging platforms with on-chip GPUs [7, 9, 24]. Whereas the criss-crossing delays of discrete systems encourage programmers to create massive compute requests, and to pipeline smaller graphics requests asynchronously, systems with on-chip accelerators can be expected to accommodate applications with smaller, more interactive requests, yielding significant improvements in flexibility, programmability, and resource utilization. Of course, CPU-GPU integration also increases the importance of direct device access from user space [3], and thus of disengaged scheduling. In an attempt to capture these trends, we have deliberately chosen benchmarks with relatively small requests.

Our OpenCL applications are from the AMD APP SDK v2.6, with no modifications except for the use of high-resolution x86 timers to collect performance results. In addition to GPU computation, most of these applications also contain some setup and data transfer time. They generally make repetitive (looping) requests to the GPU of roughly similar size. We select inputs that are meaningful and non-trivial (hundreds or thousands of elements in arrays, matrices or lists) but do not lead to executions dominated by data transfer.

Application	Area	μs per round	μs per request
BinarySearch	Searching	161	57
BitonicSort	Sorting	1292	202
DCT	Compression	197	66
EigenValue	Algebra	163	56
FastWalshTransform	Encryption	310	119
FFT	Signal Processing	268	48
FloydWarshall	Graph Analysis	5631	141
LUDecomposition	Algebra	1490	308
MatrixMulDouble	Algebra	12628	637
MatrixMultiplication	Algebra	3788	436
MatrixTranspose	Algebra	1153	284
PrefixSum	Data Processing	157	55
RadixSort	Sorting	8082	210
Reduction	Data Processing	1147	282
ScanLargeArrays	Data Processing	197	72
glxgears	Graphics	72	37
oclParticles	Physics/Graphics	2006	12/302
simpleTexture3D	Texturing/Graphics	2472	108/171

Table 1. Benchmarks used in our evaluation. A "round" is the performance unit of interest to the user: the run time of compute "kernel(s)" for OpenCL applications, or that of a frame rendering for graphics or combined applications. A round can require multiple GPU acceleration requests.

Our OpenGL application (glxgears) is a standard graphics microbenchmark, chosen for its simplicity and its frequent short acceleration requests. We also chose a combined compute/graphics (OpenCL plus OpenGL) microbenchmark from Nvidia's SDK v4.0.8: oclParticles, a particles collision physics simulation. Synchronization to the monitor refresh rates is disabled for our experiments, both at the driver and the application level, and the performance metric used is the raw, average frame rate: our intent is to stress the system interface more than the GPU processing capabilities.

Table 1 summarizes our benchmarks and their characteristics. The last two columns list how long an execution of a "round" of computations or rendering takes for each application, and the average acceleration request size realized when the application runs alone using our Kepler GPU. A "round" in the OpenCL applications is generally the execution of one iteration of the main loop of the algorithm, including one or more "compute kernel(s)." For OpenGL applications, it is the work to render one frame. The run time of a round indicates performance from the user's perspective; we use it to quantify speedup or slowdown from the perspective of a

given application across different workloads (job mixes) or using different schedulers. A "request" is the basic unit of work that is submitted to the GPU at the device interface. We capture the average request size for each application through our request interception mechanism (but without applying any scheduling policy). We have verified that our GPU request size estimation agrees, within of 5% or less, to the time reported by standard profiling tools. Combined compute/graphics applications report two average request sizes, one per request type. Note that requests are not necessarily meaningful to the application or user, but they are the focal point of our OS-level GPU schedulers. There is no necessary correlation between the API calls and GPU requests in a compute/rendering round. Many GPU-library API calls do not translate to GPU requests, while other calls could translate to more than one. Moreover, as revealed in the course of reverse engineering, there also exist trivial requests, perhaps to change mode/state, that arrive at the GPU and are never checked for completion. NEON schedules requests without regard to their higher-level purpose.

To enable flexible adjustment of workload parameters, we also developed a "Throttle" microbenchmark. Throttle serves as a controlled means of measuring basic system overheads, and as a well-understood competitive sharer in multiprogrammed scenarios. It makes repetitive, blocking compute requests, which occupy the GPU for a user-specified amount of time. We can also control idle (sleep/think) time between requests, to simulate nonsaturating workloads. No data transfers between the host and the device occur during throttle execution; with the exception of a small amount of initial setup, only compute requests are sent to the device.

5.2 Overhead of Standalone Execution

NEON routines are invoked during context creation, to mark the memory areas to be protected; upon every tracked request (depending on scheduling policy), to handle the induced page fault; and at context exit, to safely return allocated system resources. In addition, we arrange for timer interrupts to trigger execution of the polling service thread, schedule the next task on the GPU (and perhaps dis/re-engage) for the Timeslice algorithm, and sample the next task or begin a free-run period for the Disengaged Fair Queueing scheduler.

We measure the overhead of each scheduler by comparing standalone application performance under that scheduler against the performance of direct device access. Results appear in Figure 4. For this experiment, and the rest of the evaluation, we chose the following configuration parameters: For all algorithms, the polling service thread is woken up when the scheduler decides, or at 1 ms intervals. The (engaged) Timeslice and Disengaged Timeslice use a 30 ms timeslice. For Disengaged Fair Queueing, a task's sampling period lasts either 5 ms or as much as is required to intercept a fixed number of requests, whichever is smaller. The follow-up free-run period is set to be 5× as long, so in the standalone application case it should be 25 ms (even if the fixed request number mark was hit before the end of 5 ms).

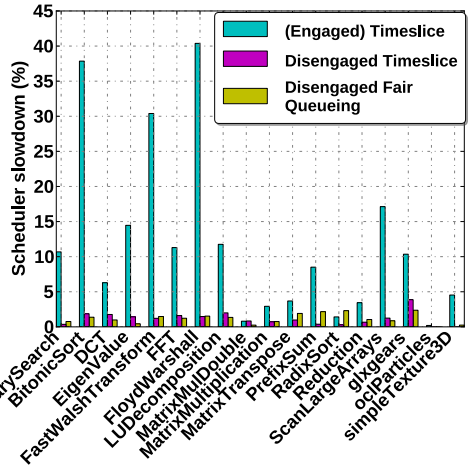

Figure 4. Standalone application execution slowdown under our scheduling policies compared to direct device access.

NEON is not particularly sensitive to configuration parameters. We tested different settings, but found the above to be sufficient. The polling thread frequency is fast enough for the average request size, but not enough to impose a noticeable load even for single-CPU systems. A 30 ms timeslice is long enough to minimize the cost of token passing, but not long enough to immediately introduce jitter in more interactive applications, which follow the "100 ms human perception" rule. For most of the workloads we tested, including applications with a graphics rendering component, 5 ms or 32 requests is enough to capture variability in request size and frequency during sampling. We increase this number to 96 requests for combined compute/graphics applications, to capture the full variability in request sizes. Though real-time rendering quality is not our goal, testing is easier when graphics applications remain responsive as designed; in our experience, the chosen configuration parameters preserved fluid behavior and stable measurements across all tested inputs.

As shown in Figure 4, the (engaged) Timeslice scheduler incurs low cost for applications with large requests—most notably MatrixMulDouble and oclParticles. Note, however, that in the latter application, the performance metric is frame rendering times, which are generally the result of longer non-blocking graphics requests. The cost of (engaged) Timeslice is significant for small-request applications (in particular, 38% slowdown for BitonicSort, 30% for FastWalshTransform, and 40% FloydWarshall). This is due to the large request interception and manipulation overhead imposed on all requests. Disengaged Timeslice suffers only the post re-engagement status update costs and the infrequent trapping to the kernel at the edges of the timeslice, which is generally no more than 2%. Similarly, Disengaged Fair Queueing incurs costs only on a small subset of GPU requests; its overhead is no more than 5% for all our appli-

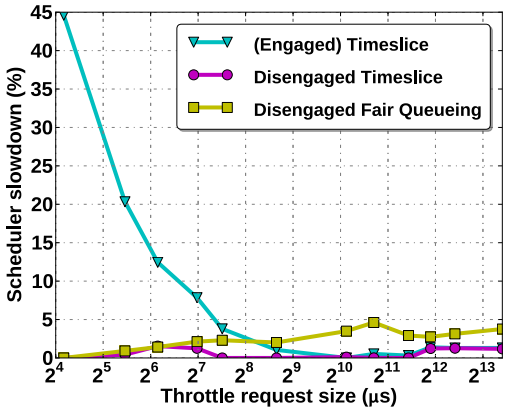

Figure 5. Standalone Throttle execution (at a range of request sizes) slowdown under our scheduling policies compared to direct device access.

cations. The principal source of extra overhead is idleness during draining, due to the granularity of polling.

Using the controlled Throttle microbenchmark, Figure 5 reports the scheduling overhead variation at different request sizes. It confirms that the (engaged) Timeslice scheduler incurs high costs for small-request applications, while Disengaged Timeslice incurs no more than 2% and Disengaged Fair Queueing no more than 5% overhead in all cases.

5.3 Evaluation on Concurrent Executions

To evaluate scheduling fairness and efficiency, we measure the slowdown exhibited in multiprogrammed scenarios, relative to the performance attained when applications have full, unobstructed access to the device. We first evaluate concurrent execution between two co-running applications—one of our benchmarks together with the Throttle microbenchmark at a controlled request size. The configuration parameters in this experiment are the same as for the overheads test, resulting in 50 ms "free-run" periods.

Fairness Figure 6 shows the fairness of our concurrent executions under different schedulers. The normalized run time in the direct device access scenario shows significantly unfair access to the GPU. The effect can be seen on both co-runners: a large-request-size Throttle workload leads to large slowdown of some applications (more than 10× for DCT, 7× for FFT, 12× for glxgears), while a smaller request size Throttle can suffer significant slowdown against other applications (more than 3× for DCT and 7× for FFT). The high-level explanation is simple—on accelerators that issue requests round-robin from different channels, the one with larger requests will receive a proportionally larger share of device time. In comparison, our schedulers achieve much more even results—in general, each co-scheduled task experiences the expected 2× slowdown. More detailed discussion of variations appears in the paragraphs below.

There appears to be a slightly uneven slowdown exhibited by the (engaged) Timeslice scheduler for smaller Throttle request sizes. For example, for DCT or FFT, the slowdown of Throttle appears to range from 2× to almost 3×. This variation is due to per-request management costs. Throttle, being our streamlined control microbenchmark, makes back-to-back compute requests to the GPU, with little to no CPU processing time between the calls. At a given request size, most of its co-runners generate fewer requests per timeslice. Since (engaged) Timeslice imposes overhead on every request, the aggressive Throttle microbenchmark tends to suffer more. By contrast, Disengaged Timeslice does not apply this overhead and thus maintains an almost uniform 2× slowdown for each co-runner and for every Throttle request size.

A significant anomaly appears when we run the OpenGL glxgears graphics application vs. Throttle (19 μs) under Disengaged Fair Queueing, where glxgears suffers a significantly higher slowdown than Throttle does. A closer inspection shows that glxgears requests complete at almost one third the rate that Throttle (19 μs) requests do during the free-run period. This behavior appears to contradict our (simplistic) assumption of round-robin channel cycling during the Disengaged Fair Queueing free-run period. Interestingly, the disparity in completion rates goes down with larger Throttle requests: Disengaged Fair Queueing appears to produce fair resource utilization between glxgears and Throttle with larger requests. In general, the internal scheduling of our Nvidia device appears to be more uniform for OpenCL than for graphics applications. As noted in the "Limitations" paragraph of Section 3.3, a production-quality implementation of our Disengaged Fair Queueing scheduler should ideally be based on vendor documentation of device-internal scheduling or resource usage statistics.

Finally, we note that combined compute/graphics applications, such as oclParticles and simpleTexture3D, tend to create separate channels for their different types of requests. For multiprogrammed workloads with multiple channels per task, the limitations of imprecise understanding of internal GPU scheduling are even more evident than they were for glxgears: not only do requests arrive on two channels, but also for some duration of time requests may be outstanding in one channel but not on the other. As a result, even if the GPU did cycle round-robin among active channels, the resource distribution among competitors might not follow their active channel ratio, but could change over time. In a workload with compute/graphics applications, our request-size estimate across the two (or more) channels of every task becomes an invalid proxy of resource usage during free-run periods and unfairness arises: in the final row of Figure 6, Throttle suffers a 2.3× to 2.7× slowdown, while oclParticles sees 1.3× to 1.5×. Completely fair scheduling with Disengaged Fair Queueing would again require accurate, vendor-provided information that our reverse-engineering effort in building this prototype was unable to provide.

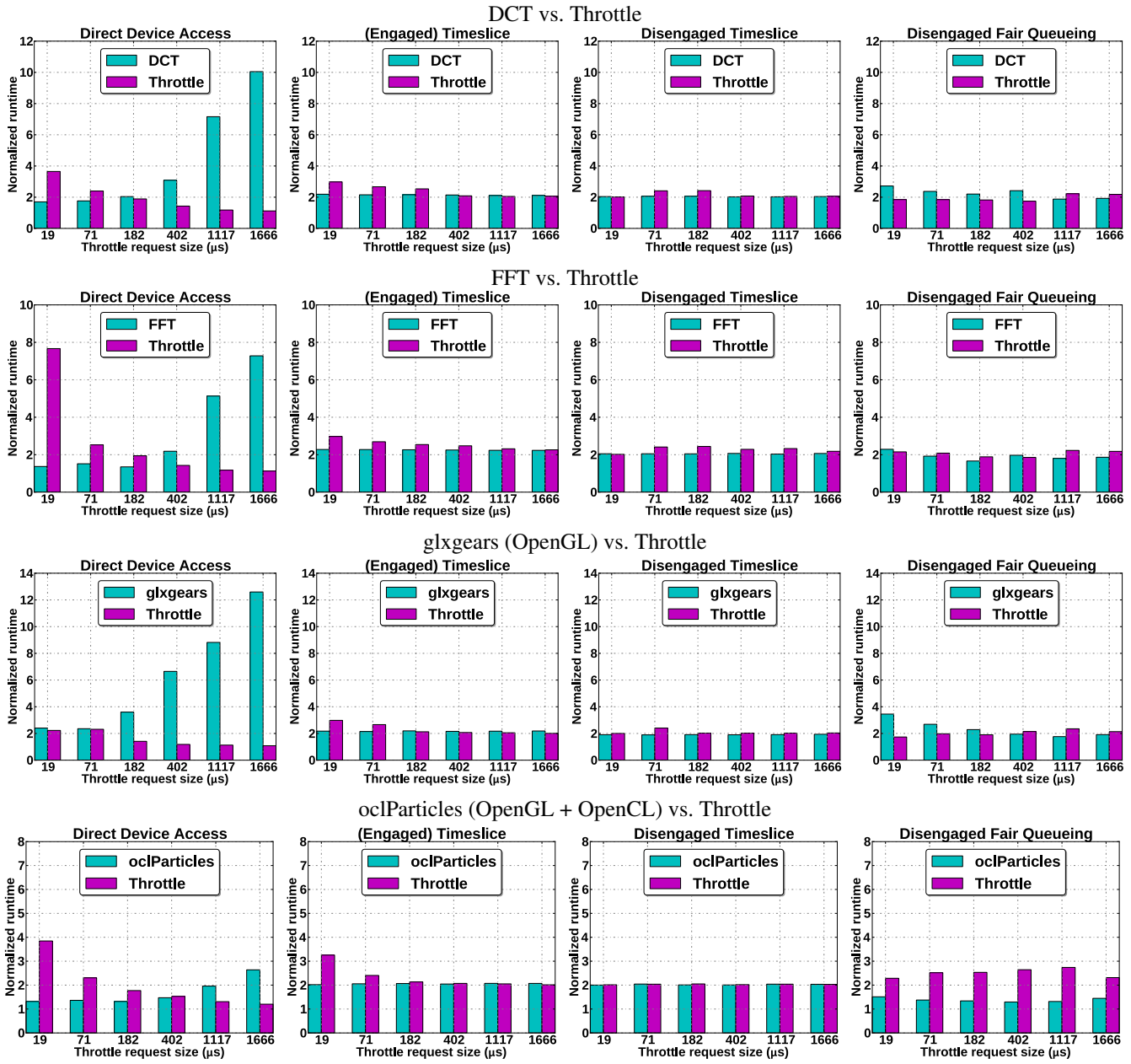

Figure 6. Performance and fairness of concurrent executions. Results cover four application-pair executions (one per row) and four schedulers (one per column). We use several variations of Throttle with different request sizes (from 19 μs to 1.7 ms). The application runtime is normalized to that when it runs alone with direct device access.

Efficiency In addition to fairness, we also assess the overall system efficiency of multiprogrammed workloads. Here we utilize a *concurrency efficiency* metric that measures the performance of the concurrent execution relative to that of the applications running alone. We assign a base efficiency of 1.0 to each application's running-alone performance (in the absence of resource competition). We then assess the performance of a concurrent execution relative to this base. Given N concurrent OpenCL applications whose per-round run times are $t_1, t_2, ..., t_N$ respectively when running alone on the GPU, and $t_1^c, t_2^c, ..., t_N^c$ when running together, the concurrency efficiency is defined as $\Sigma_{i=1}^{N}(t_i/t_i^c)$. The intent of the formula is to sum the resource shares allotted to each application. A sum of less than one indicates that resources have been lost; a sum of more than one indicates mutual synergy (e.g., due to overlapped computation).

Figure 7 reports the concurrency efficiency of direct device access and our three fair schedulers. It might at first be surprising that the concurrency efficiency of direct device access deviates from 1.0. For small requests, the GPU may fre-

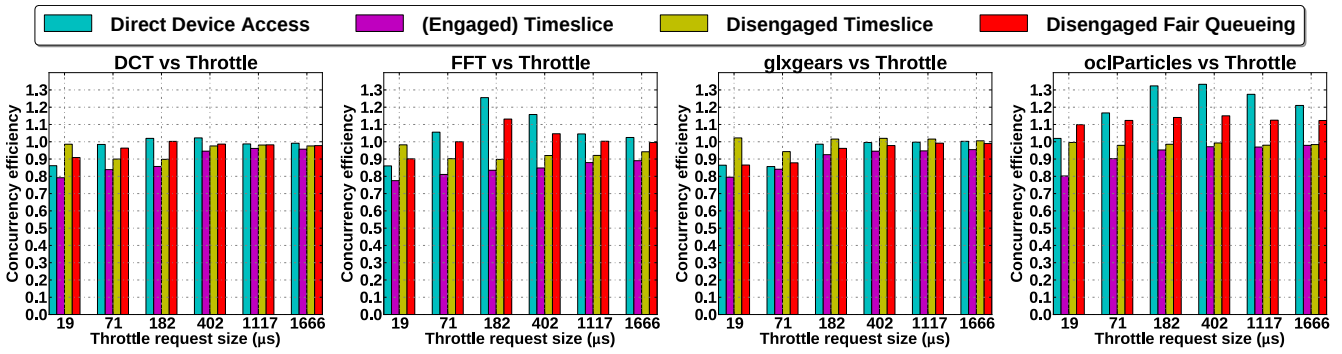

Figure 7. Efficiency of concurrent executions. The metric of concurrency efficiency is defined in the text.

quently switch between multiple application contexts, leading to substantial context switch costs and <1.0 concurrency efficiency. Conversely, the efficiency of direct device access may also be larger than 1.0, due to overlap between DMA and compute requests.

For its part, the (engaged) Timeslice scheduler incurs high request management costs. Its efficiency losses with respect to direct device access can be on average 19% or as high as 42%; specifically, average/max efficiency losses are 10%/16% (DCT vs. Throttle), 27/42% (FFT vs. Throttle), 5%/6% (glxgears vs. Throttle) and 35/37% (oclParticles vs. Throttle).

Disengaged Timeslice adds little per-request overhead while also limiting channel switching on the GPU, so it enjoys some efficiency advantage over (engaged) Timeslice: average/max losses with respect to direct device access are 10%/35% (3%/12%, 16%/35%, −5%/0%, and 27%/33% per respective test).

Overall, Disengaged Fair Queueing does a better job at allowing "background concurrency": its average/max losses with respect to direct device access are only 4%/18% (0%/3%, 6%/12%, 0%/2% and 11%/18% per respective test). Unfortunately, our incomplete understanding of GPU internal scheduling limits our achievable efficiency, as the imprecise slowdown applied for applications with multiple channels results in wasted opportunities for Disengaged Fair Queueing. This is why oclParticles exhibits the worst efficiency loss among the tested cases.

Scalability The performance, fairness, and efficiency characteristics of our schedulers are maintained under higher concurrency. Figure 8 shows that for four concurrent applications, including one making large requests (Throttle) and three making smaller ones (BinarySearch, DCT, FFT), the average slowdown remains at 4–5×. Following the pattern we have observed before, the efficiency of concurrent runs drops more when the scheduler is fully engaged (13% over direct device access), but less so when we use disengaged scheduling: 8% for Disengaged Timeslice and 7% for Disengaged Fair Queueing.

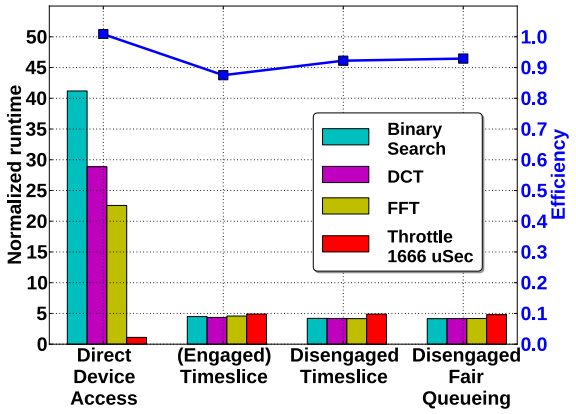

Figure 8. Fairness (bars, left Y axis) and efficiency (line, right Y axis) when running four concurrent applications.

5.4 Evaluation on Nonsaturating Workloads

Next we assess our scheduler performance under nonsaturating workloads—those that interleave GPU requests with "off" periods of CPU-only work. Nonsaturating workloads lead to GPU under-utilization with non-work-conserving schedulers. In timeslice scheduling, for instance, resources are wasted if a timeslice is only used partially. As explained in Section 3.3, Disengaged Fair Queueing maintains work-conserving properties, and therefore wastes less time on nonsaturating workloads. Direct device access, of course, is fully work-conserving.

To create nonsaturating workloads, we interleave sleep time in our Throttle microbenchmark to reach a specified ratio (the proportion of "off" time under standalone execution). Figure 9 illustrates the results for DCT and the nonsaturating Throttle workload. Note that fairness does not necessarily require co-runners to suffer equally: execution is fair as long as no task slows down significantly more than 2×. The results for Disengaged Fair Queueing should thus be regarded as better than those for the timeslice schedulers, because Throttle does not suffer, while DCT actually benefits from its co-runner's partial idleness. Figure 10 further shows the efficiency losses for our three schedulers and direct device access. At an 80% Throttle sleep ratio, the losses

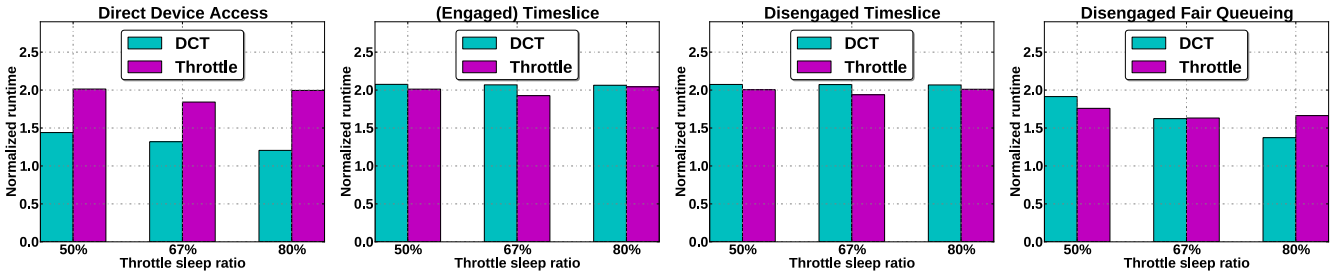

Figure 9. Performance and fairness of concurrent executions for nonsaturating workloads, demonstrated for DCT vs. Throttle with GPU "off" periods.

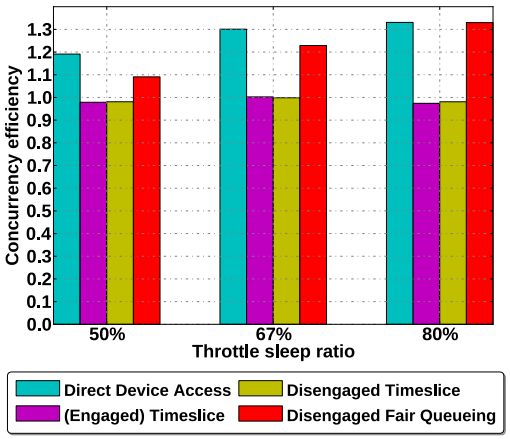

Figure 10. Efficiency of concurrent executions for nonsaturating workloads.

relative to direct access are 36%, 34%, and essentially 0% for (engaged) Timeslice, Disengaged Timeslice, and Disengaged Fair Queueing, respectively.

6. Future System Considerations

6.1 Interface Specification

Protected OS-level accelerator management requires relatively little information: an event-based scheduling interface that allows us to identify when acceleration requests have been made or completed, utilization information that allows us to account for per-channel resource usage, and sufficient semantic understanding of requests to avoid deadlocks and any unnecessary resource idleness.

Both standard interfaces (system calls) and alternative approaches (virtual memory) can capture request submission and completion. We believe that the increasing integration of CPUs and accelerators will only increase the need for fast, memory-based interfaces. By documenting basic information regarding the layout of accelerator request queues and core component interactions, and by exposing necessary performance counters to obtain accurate utilization statistics, vendors could enable OS kernel involvement in accelerator scheduling without high overheads, enabling full protection and efficiency through disengaged scheduling. Vendor assistance at this level is necessary, as systems like NEON, relying on reverse engineering and inference techniques, cannot reach production-level quality.

A "partial opening" of interface specifications, as envisioned here, would still allow vendors to retain proprietary control over much of the accelerator software / hardware system. For instance, while a request submission event needs to be identifiable for manipulation, much of the data layout and semantics within a request data structure can remain unpublished. On the other hand, in order to enable deep scheduling optimizations, vendors would have to reveal request-ordering semantics and synchronization models. For our work with NEON we assumed no request reordering within GPU contexts, an assumption that flows naturally from the semantics of the context concept. For a higher degree of optimizations (e.g., reordering of requests across CUDA streams [28]), vendors could tag requests to indicate the requirements to be met.

A hardware accelerator may accept multiple requests simultaneously to exploit internal parallelism or to optimize for data affinity. The OS desires additional information to manage such parallelism. Most obviously, completion events need to carry labels that link them to the corresponding requests. For NEON, the task, context, channel, and request reference number have served as surrogate labels. More subtly, under concurrent submission, an outstanding request may be queued in the device, so that the time between submission and completion does not accurately reflect its resource usage. The hardware can facilitate OS accounting by including resource usage information in each completion event.

Direct accelerator access from other (e.g., I/O) devices is currently limited to DMA (e.g., RDMA for GPUDirect in the CUDA SDK [28]). Applications and services using the GPU and I/O devices will still initiate DMA and compute requests through the same memory-mapped interface, and thus be amenable to scheduling. Appropriate specifications for any alternative, peer-to-peer communication path to the accelerator would be necessary if devices other than the CPU were to enjoy rich, direct accelerator interactions.

6.2 Hardware Preemption Support

The reverse-engineering efforts of the open-source community [27, 31] suggest that some type of request arbitration is possible for current GPUs, but true hardware preemption is officially still among the list of upcoming GPU features (e.g., in the roadmap of AMD HSA [24]). In its absence, tricks, like cutting longer requests into smaller pieces, have been shown to enhance GPU interactivity, at least for mutually cooperative applications [6]. True hardware preemption support would save state and safely context-switch from an ongoing request to the next in the GPU queue. If the preemption decision were left to a hardware scheduler, it would be fast but oblivious to system scheduling objectives. Disengaged scheduling can help make context-switching decisions appropriately coarse grain, so they can be deferred to the OS. Simple documentation of existing mechanisms to identify and kill the currently running context would enable full protection for schedulers like Disengaged Fair Queueing. True hardware preemption would allow us to tolerate requests of arbitrary length, without sacrificing interactivity or becoming vulnerable to infinite loops.

6.3 Protection of Other Resources

Beyond the scheduling of cycles, OS-level protection can enforce the safe use of other GPU resources, such as channels and memory, to prevent unfair or abusive behavior. Channels establish a user-mapped address range through which commands are submitted to the hardware. Existing GPUs do not multiplex requests from different tasks on a channel; instead, an established channel is held by the task that created it (and its associated context) for as long as the application deems necessary. It is thus possible to devise a simple denial-of-service attack by creating many contexts and occupying all available GPU channels, thus denying subsequent requests from other applications; on our Nvidia GTX670 GPU, after 48 contexts had been created (with one compute and one DMA channel each), no other application could use the GPU.

An OS-level resource manager may enable protected allocation of GPU channels to avoid such DoS attacks. We describe a possible policy below. First, we limit the number of channels in any one application to a small constant C, chosen to match the characteristics of anticipated applications. Attempts to allocate more than C channels return an "out of resources" error. Second, with a total of D channels available on the GPU, we permit no more than D/C applications to use the GPU at any given time. More elaborate policies, involving prioritization, preemption, etc., could be added when appropriate.

Our GTX670 GPU has 2 GB of onboard RAM that is shared by active tasks. This memory is managed with the help of the system's virtual memory manager and the IOMMU, providing already sufficient isolation among tasks. It is conceivable that an erroneous or malicious application might exhaust the GPU memory and prevent normal use by others. In theory, our OS-level protection framework could prevent such abuses by accounting for application uses of GPU memory and blocking excessive consumption. This, however, requires uncovering additional GPU semantics on memory buffer management that we have not yet explored.

7. Conclusion

This paper has presented a framework for fair, safe, and efficient OS-level management of GPU resources—and, by extension, of other fast computational accelerators. We believe our request interception mechanism, combined with disengagement, to be the first resource management technique capable of approaching the performance of direct device access from user space. Our Timeslice scheduler with overuse control can guarantee fairness during multiprogrammed executions. Its disengaged variant improves performance without sacrificing either fairness or protection, by intercepting only a small subset of requests, allowing most to reach the device directly. Our Disengaged Fair Queueing scheduler is effective at limiting idleness, and incurs only 4% average overhead (max 18%) relative to direct device access across our evaluation cases.

While our prototype Disengaged Fair Queueing scheduler ensures fairness between multiple compute (OpenCL) applications, its fairness support is less than ideal for graphics (OpenGL)-related applications. This is due to our insufficient understanding of the device-internal scheduling of concurrent graphics/compute requests during free-run periods; a production-quality implementation based on vendor-supplied information would not suffer from this limitation. Even in our prototype, the limitation does not affect the fairness of our Disengaged Timeslice scheduler, which grants GPU access to no more than one application at a time.

We have identified necessary and/or desirable hardware features to support disengaged resource management, including the ability to identify the currently running GPU request, to account for per-context resource usage during disengaged execution, to cleanly terminate an aberrant context, and (ideally) to suspend and resume requests. For the purposes of prototype construction, we have identified surrogates for some of these features via reverse engineering. We encourage device vendors to embrace a limited "opening" of the hardware-software interface specification sufficient for production use.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCF-0937571, CCR-0963759, CCF-1116055, CNS-1116109, CNS-1217372, CNS-1239423, CCF-1255729, and CNS-1319417. Our thanks as well to Shinpei Kato and the anonymous reviewers, for comments that helped to improve this paper.

References

[1] Altera. www.altera.com.

[2] AMD. Radeon R5xx Acceleration: v1.2, Feb. 2008.

[3] AMD's revolutionary Mantle. http://www.amd.com/us/products/technologies/mantle.

[4] ARM graphics plus GPU compute. www.arm.com/products/multimedia/mali-graphics-plus-gpu-compute.

[5] M. Aron and P. Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Trans. on Computer Systems*, 18(3):197–228, Aug. 2000.

[6] C. Basaran and K.-D. Kang. Supporting preemptive task executions and memory copies in gpgpus. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 287–296. IEEE, 2012.

[7] N. Brookwood. AMD Fusion^{TM} family of APUs. AMD White Paper, Mar. 2010.

[8] C. Cascaval, S. Chatterjee, H. Franke, K. Gildea, and P. Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54(5):5–1, 2010.

[9] M. Daga, A. M. Aji, and W.-C. Feng. On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. In *Proc. of the 2011 Symp. on Application Accelerators in High-Performance Computing*, Knoxville, TN, July 2011.

[10] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of the ACM SIGCOMM Conf.*, Austin, TX, Sept. 1989.

[11] A. Dwarakinath. A fair-share scheduler for the graphics processing unit. Master's thesis, Stony Brook University, Aug. 2008.

[12] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, Jan. 2012.

[13] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. LoGV: Low-overhead GPGPU virtualization. In *Proceedings of the 4th Intl. Workshop on Frontiers of Heterogeneous Computing*. IEEE, 2013.

[14] A. G. Greenberg and N. Madras. How fair is fair queuing. *Journal of the ACM*, 39(3):568–598, July 1992.

[15] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proc. of the USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[16] Intel. OpenSource HD Graphics Programmer's Reference Manual: vol. 1, part 2, May 2012.

[17] Intel Corporation. Enabling consistent platform-level services for tightly coupled accelerators. White Paper, 2008.

[18] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proc. of the ACM SIGMETRICS Conf.*, New York, NY, June 2004.

[19] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. of the USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[20] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *Proc. of the USENIX Annual Technical Conf.*, Boston, MA, June 2012.

[21] Khronos OpenCL Working Group and others. The OpenCL specification, v1.2, 2011. A. Munshi, Ed.

[22] Khronos OpenCL Working Group and others. The OpenGL graphics system: A specification, v4.2, 2012. Mark Segal and Kurt Aekeley, Eds.

[23] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Minneapolis, MN, Sept. 2012.

[24] G. Kyriazis. Heterogeneous system architecture: A technical review. Technical report, AMD, Aug. 2012. Rev. 1.0.

[25] K. Menychtas, K. Shen, and M. L. Scott. Enabling OS research by inferring interactions in the black-box GPU stack. In *Proc. of the USENIX Annual Technical Conf.*, San Jose, CA, June 2013.

[26] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[27] Nouveau: Accelerated open source driver for Nvidia cards. nouveau.freedesktop.org.

[28] NVIDIA Corp. CUDA SDK, v5.0. http://docs.nvidia.com/cuda/.

[29] S. Panneerselvam and M. M. Swift. Operating systems should manage accelerators. In *Proc. of the Second USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, June 2012.

[30] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Trans. on Networking*, 1(3), June 1993.

[31] Nouveau, by PathScaleInc. github.com/pathscale/pscnv.

[32] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proc. of the 23rd ACM Symp. on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[33] K. Shen and S. Park. FlashFQ: A fair queueing I/O scheduler for Flash-based SSDs. In *Proc. of the USENIX Annual Technical Conf.*, San Jose, CA, June 2013.

[34] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proc. of the ACM SIGCOMM Conf.*, Cambridge, MA, Aug. 1995.

[35] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proc. of the 9th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, Oct. 2010.

[36] P. M. Stillwell, V. Chadha, O. Tickoo, S. Zhang, R. Illikkal, R. Iyer, and D. Newell. HiPPAI: High performance portable accelerator interface for SoCs. In *Proc. of the 16th IEEE Conf. on High Performance Computing (HiPC)*, Kochi, India, Dec. 2009.