Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems

Vishakha Gupta
Georgia Institute of Technology

Karsten Schwan
Georgia Institute of Technology

Niraj Tolia
Maginatics

Vanish Talwar
HP Labs

Parthasarathy Ranganathan
HP Labs

Abstract

Heterogeneous multi-cores—platforms comprised of both general purpose and accelerator cores—are becoming increasingly common. While applications wish to freely utilize all cores present on such platforms, operating systems continue to view accelerators as specialized devices. The Pegasus system described in this paper uses an alternative approach that offers a uniform resource usage model for all cores on heterogeneous chip multiprocessors. Operating at the hypervisor level, its novel scheduling methods fairly and efficiently share accelerators across multiple virtual machines, thereby making accelerators into first class schedulable entities of choice for many-core applications. Using NVIDIA GPGPUs coupled with x86-based general purpose host cores, a Xen-based implementation of Pegasus demonstrates improved performance for applications by better managing combined platform resources. With moderate virtualization penalties, performance improvements range from 18% to 140% over base GPU driver scheduling when the GPUs are shared.

1 Introduction

Systems with specialized processors like those used for accelerating computations, network processing, or cryptographic tasks [27, 34] have proven their utility in terms of higher performance and lower power consumption. This is not only causing tremendous growth in accelerator-based platforms, but it is also leading to the release of heterogeneous processors where x86-based cores and on-chip network or graphics accelerators [17, 31] form a common pool of resources. However, operating systems and virtualization platforms have not yet adjusted to these architectural trends. In particular, they continue to treat accelerators as secondary devices and focus scheduling and resource management on their general purpose processors, supported by vendors that shield developers from the complexities of accelerator hardware by 'hiding' it behind drivers that only expose higher level programming APIs [19, 28]. Unfortunately, technically, this implies that drivers rather than operating systems or hypervisors determine how accelerators are shared, which restricts scheduling policies and thus, the optimization criteria applied when using such heterogeneous systems.

A driver-based execution model can not only potentially hurt utilization, but also make it difficult for applications and systems to obtain desired benefits from the combined use of heterogeneous processing units. Consider, for instance, an advanced image processing service akin to HP's Snapfish [32] or Microsoft's PhotoSynth [25] applications, but offering additional computational services like complex image enhancement and watermarking, hosted in a data center. For such applications, the low latency responses desired by end users require the combined processing power of both general purpose and accelerator cores. An example is the execution of sequences of operations like those that first identify spatial correlation or correspondence [33] between images prior to synthesizing them [25]. For these pipelined sets of tasks, some can efficiently run on multicore CPUs, whereas others can substantially benefit from acceleration [6, 23]. However, when they concurrently use both types of processing resources, low latency is attained only when different pipeline elements are appropriately co- or gang-scheduled onto both CPU and GPU cores. As shown later in this paper, such co-scheduling is difficult to perform with current accelerators when used in consolidated data center settings. Further, it is hard to enforce fairness in accelerator use when the many clients in typical web applications cause multiple tasks to compete for both general purpose and accelerator resources,

The *Pegasus* project addresses the urgent need for systems support to smartly manage accelerators. It does this by leveraging the new opportunities presented by increased adoption of virtualization technology in commercial, cloud computing [1], and even high performance infrastructures [22, 35]: *the Pegasus hypervisor*

extensions (1) make accelerators into first class schedulable entities and (2) support scheduling methods that enable efficient use of both the general purpose and accelerator cores of heterogeneous hardware platforms. Specifically, for platforms comprised of x86 CPUs connected to NVIDIA GPUs, these extensions can be used to manage all of the platform's processing resources, to address the broad range of needs of GPGPU (general purpose computation on graphics processing units) applications, including the high throughput requirements of compute intensive web applications like the image processing code outlined above and the low latency requirements of computational finance [24] or similarly computationally intensive high performance codes. For high throughput, platform resources can be shared across many applications and/or clients. For low latency, resource management with such sharing also considers individual application requirements, including those of the inter-dependent pipeline-based codes employed for the financial and image processing applications.

The Pegasus hypervisor extensions described in Sections 3 and 5 do not give applications direct access to accelerators [28], nor do they hide them behind a virtual file system layer [5, 15]. Instead, similar to past work on self-virtualizing devices [29], Pegasus exposes to applications a virtual accelerator interface, and it supports existing GPGPU applications by making this interface identical to NVIDIA's CUDA programming API [13]. As a result, whenever a virtual machine attempts to use the accelerator by calling this API, control reverts to the hypervisor. This means, of course, that the hypervisor 'sees' the application's accelerator accesses, thereby getting an opportunity to regulate (schedule) them. A second step taken by Pegasus is to then explicitly *coordinate* how VMs use general purpose and accelerator resources. With the Xen implementation [7] of Pegasus shown in this paper, this is done by explicitly scheduling guest VMs' accelerator accesses in Xen's Dom0, while at the same time controlling those VMs' use of general purpose processors, the latter exploiting Dom0's privileged access to the Xen hypervisor and its VM scheduler.

Pegasus elevates accelerators to first class schedulable citizens in a manner somewhat similar to the way it is done in the Helios operating system [26], which uses satellite kernels with standard interfaces for XScale-based IO cards. However, given the fast rate of technology development in accelerator chips, we consider it premature to impose a common abstraction across all possible heterogeneous processors. Instead, Pegasus uses a more loosely coupled approach in which it assumes systems to have different 'scheduling domains', each of which is adept at controlling its own set of resources, e.g., accelerator vs. general purpose cores. Pegasus scheduling, then, coordinates when and to what extent, VMs use the resources managed by these multiple scheduling domains. This approach leverages notions of 'cellular' hypervisor structures [11] or federated schedulers that have been shown useful in other contexts [20]. Concurrent use of both CPU and GPU resources is one class of coordination methods Pegasus implements, with other methods aimed at delivering both high performance and fairness in terms of VM usage of platform resources.

Pegasus relies on application developers or toolchains to identify the right target processors for different computational tasks and to generate such tasks with the appropriate instruction set architectures (ISAs). Further, its current implementation does not interact with tool chains or runtimes, but we recognize that such interactions could improve the effectiveness of its runtime methods for resource management [8]. An advantage derived from this lack of interaction, however, is that Pegasus does not depend on certain toolchains or runtimes, nor does it require internal information about accelerators [23]. As a result, Pegasus can operate with both 'closed' accelerators like NVIDIA GPUs and with 'open' ones like IBM Cell [14], and its approach can easily be extended to support other APIs like OpenCL [19].

Summarizing, the Pegasus hypervisor extensions make the following contributions:

Accelerators as first class schedulable entities— accelerators (accelerator physical CPUs or aPCPUs) can be managed as first class schedulable entities, i.e., they can be shared by multiple tasks, and task mappings to processors are dynamic, within the constraints imposed by the accelerator software stacks.

Visible heterogeneity—Pegasus respects the fact that aPCPUs differ in capabilities, have different modes of access, and sometimes use different ISAs. Rather than hiding these facts, Pegasus exposes heterogeneity to the applications and the guest virtual machines (VMs) that are capable of exploiting it.

Diversity in scheduling—accelerators are used in multiple ways, e.g., to speedup parallel codes, to increase throughput, or to improve a platform's power/performance properties. Pegasus addresses differing application needs by offering a diversity of methods for scheduling accelerator and general purpose resources, including co-scheduling for concurrency constraints.

'Coordination' as the basis for resource management—internally, accelerators use specialized execution environments with their own resource managers [14, 27]. Pegasus uses *coordinated scheduling methods* to align accelerator resource usage with platform-level management. While coordination applies external controls to control the use of 'closed' accelerators, i.e., accelerators with resource managers that do not export coordination interfaces, it could interact more intimately with 'open' managers as per their internal scheduling methods.

Novel scheduling methods—current schedulers on parallel machines assume complete control over their underlying platforms' processing resources. In contrast, Pegasus recognizes and deals with heterogeneity not only in terms of differing resource capabilities, but also in terms of the diverse scheduling methods these resources may require, an example being the highly parallel internal scheduling used in GPGPUs. Pegasus coordination methods, therefore, differ from traditional co-scheduling in that they operate above underlying native techniques. Such meta-scheduling, therefore, seeks to influence the actions of underlying schedulers rather than replacing their functionality. This paper proposes and evaluates new coordination methods that are geared to dealing with diverse resources, including CPUs vs. GPUs and multiple generations of the latter, yet at the same time, attempting to preserve desired virtual platform properties, including fair-sharing and rioritization.

The current Xen-based Pegasus prototype efficiently virtualizes NVIDIA GPUs, resulting in performance competitive with that of applications that have direct access to the GPU resources, as shown in Section 6. More importantly, when the GPGPU resources are shared by multiple guest VMs, online resource management becomes critical. This is evident from the performance benefits derived from the coordination policies described in Section 4, which range from 18% to 140% over base GPU driver scheduling. An extension to the current, fully functional, single-node Pegasus prototype will be deployed to a large-scale GPU-based cluster machine, called Keeneland, under construction at Oak Ridge National Labs [35], to further validate our approach and to better understand how to improve the federated scheduling infrastructures needed for future larger scale heterogeneous systems.

In the remaining paper, Section 2 articulates the need for smart accelerator sharing. Section 3 outlines the Pegasus architecture. Section 4 describes its rich resource management methods. A discussion of scheduling policies is followed by implementation details in Section 5, and experimental evaluation in Section 6. Related work is in Section 7, followed by conclusions and future work.

2 Background

This section offers additional motivation for the Pegasus approach on a heterogeneous multi-core platforms.

Value in sharing resources—Accelerator performance and usability (e.g., the increasing adoption of CUDA) are improving rapidly. However, even for today's platforms, the majority of applications do not occupy the entire accelerator [2, 18]. In consequence and despite continuing efforts to improve the performance of single accelerator applications [12], resource sharing is now supported in NVIDIA's Fermi architecture [27], IBM's Cell, and others. These facts are the prime drivers behind our decision to develop scheduling methods that can efficiently utilize both accelerator and general purpose cores. However, as stated earlier, for reasons of portability across different accelerators and accelerator generations, and to deal with their proprietary nature, Pegasus resource sharing across different VMs is implemented at a layer above the driver, leaving it up to the individual applications running in each VM to control and optimize their use of accelerator resources.

Limitations of traditional device driver based solutions—Typical accelerators have a sophisticated and often proprietary device driver layer, with an optional runtime. While these efficiently implement the computational and data interactions between accelerator and host cores [28], they lack support for efficient resource sharing. For example, first-come-first-serve issue of CUDA calls from 'applications-to-GPU' through a centralized NVIDIA-driver can lead to possibly detrimental call interleavings, which can cause high variances in call times and degradation in performance, as shown by measurements of the NVIDIA driver in Section 6. Pegasus can avoid such issues and use a more favorable call order, by introducing and regulating time-shares for VMs to issue GPU-requests. This leads to significantly improved performance even for simple scheduling schemes.

3 Pegasus System Architecture

Designed to generalize from current accelerator-based systems to future heterogeneous many-core platforms, Pegasus creates the logical view of computational resources shown in Figure 1. In this view, general purpose and accelerator tasks are schedulable entities mapped to VCPUs (virtual CPUs) characterized as general purpose or as 'accelerator'. Since both sets of processors can be scheduled independently, platform-wide scheduling, then, requires Pegasus to federate the platform's general purpose and accelerator schedulers. Federation is implemented by coordination methods that provide the serviced virtual machines with shares of physical processors based on the diverse policies described in Section 4. *Coordination is particularly important for closely coupled tasks running on both accelerator and general purpose cores*, as with the image processing application explained earlier. Figure 1 shows virtual machines running on either one or both types of processors, i.e., the CPUs and/or the accelerators. The figure also suggests the relative rarity of VMs running solely on accelerators (grayed out in the figure) in current systems. We segregate the privileged software components shown for the host and accelerator cores to acknowledge that the accelerator could have its own privileged runtime.

The following questions articulate the challenges in achieving the vision shown in Figure 1.

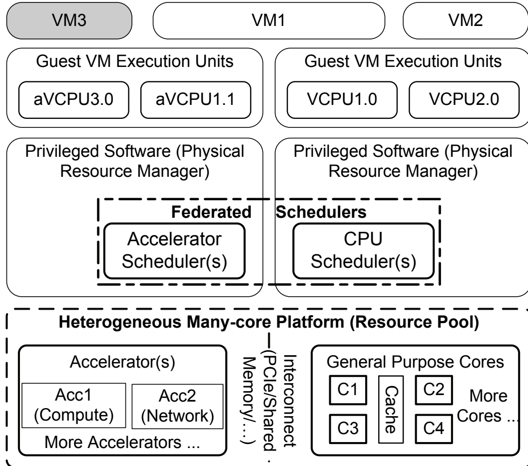


Figure 1: Logical view of Pegasus architecture

How can heterogeneous resources be managed?: Hardware heterogeneity goes beyond varying compute speeds to include differing interconnect distances, different and possibly disjoint memory models, and potentially different or non-overlapping ISAs. This makes it difficult to assimilate these accelerators into one common platform. Exacerbating these hardware differences are software challenges, like those caused by the fact that there is no general agreement about programming models and runtimes for accelerator-based systems [19, 28].

Are there efficient methods to utilize heterogeneous resources?: The hypervisor has limited control over how the resources internal to closed accelerators are used, and whether sharing is possible in time, space, or both because there is no direct control over scheduler actions beyond the proprietary interfaces. The concrete question, then, is whether and to what extent the coordinated scheduling approach adopted by Pegasus can succeed.

Pegasus therefore allows schedulers to run resource allocation policies that offer diversity in how they maximize application performance and/or fairness in resource sharing.

3.1 Accelerator Virtualization

With GViM [13], we outline methods for low-overhead virtualization of GPUs for the Xen hypervisor, addressing heterogeneous hardware with general purpose and accelerator cores, used by VMs with suitable codes (e.g., for Larrabee or Tolapai cores, codes that are IA instruction set compatible vs. non-IA compatible codes for NVIDIA or Cell accelerators). Building on this approach and acknowledging the current off-chip nature of accelerators, Pegasus assumes these hardware resources to be managed by both the hypervisor and Xen's 'Dom0' management (and driver) domain. Hence, Pegasus uses front end/back end split drivers [3] to mediate all accesses to GPUs connected via PCIe. Specifically, the requests for GPU usage issued by guest VMs (i.e., CUDA tasks) are contained in call buffers shared between guests and Dom0, as shown in Figure 2, using a separate buffer for each guest. Buffers are inspected by 'poller' threads that pick call packets from per-guest buffers and issue them to the actual CUDA runtime/driver resident in Dom0. These poller threads can be woken up whenever a domain has call requests waiting. This model of execution is well-matched with the ways in which guests use accelerators, typically wishing to utilize their computational capabilities for some time and with multiple calls.

For general purpose cores, a VCPU as the (virtual) CPU representation offered to a VM embodies the state representing the execution of the VM's threads/processes on physical CPUs (PCPUs). As a similar abstraction, Pegasus introduces the notion of an *accelerator VCPU (aVCPU)*, which embodies the VM's state concerning the execution of its calls to the accelerator. For the Xen/NVIDIA implementation, this abstraction is a combination of state allocated on the host and on the accelerator (i.e., Dom0 polling thread, CUDA calls, and driver context form the execution context while the data that is operated upon forms the data portion, when compared with the VCPUs). By introducing aVCPUs, Pegasus can then explicitly schedule them, just like their general purpose counterparts. Further, and as seen from Section 6, virtualization costs are negligible or low and with this API-based approach to virtualization, Pegasus leaves the use of resources on the accelerator hardware up to the application, ensures portability and independence from low-level changes in NVIDIA drivers and hardware.

3.2 Resource Management Framework

For VMs using both VCPUs and aVCPUs, resource management can explicitly track and schedule their joint use of both general purpose and accelerator resources. Technically, such management involves scheduling their VCPUs and aVCPUs to meet desired Service Level Objectives (SLOs), concurrency constraints, and to ensure fairness in different guest VMs' resource usage.

For high performance, Pegasus distinguishes two phases in accelerator request scheduling. First, the **accelerator selection module** runs in the Accelerator Domain—which in our current implementation is Dom0—henceforth, called *DomA*. This module associates a domain, i.e., a guest VM, with an accelerator that has available resources, by placing the domain into an 'accelerator ready queue', as shown in Figure 2. Domains are selected from this queue when they are ready to issue requests. Second, it is only after this selection that actual usage requests are forwarded to, i.e., scheduled and run on, the selected accelerator. There are multiple reasons for this difference in accelerator vs. CPU scheduling. (1) An accelerator like the NVIDIA GPU

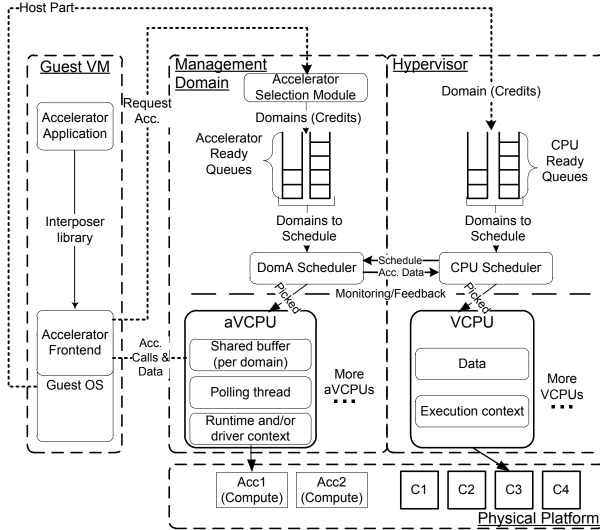


Figure 2: Logical view of the resource management framework in Pegasus

has limited memory, and it associates a context with each 'user' (e.g., a thread) that locks some of the GPU's resources. (2) Memory swapping between host and accelerator memory over an interconnect like PCIe is expensive, which means that it is costly to dynamically change the context currently running on the GPU. In response, Pegasus GPU scheduling restricts the number of domains simultaneously scheduled on each accelerator and in addition, it permits each such domain to use the accelerator for some extensive time duration. The following parameters are used for accelerator selection.

Accelerator profile and queue—accelerators vary in terms of clock speed, memory size, in-out bandwidths and other such physical characteristics. These are static or hardware properties that can identify capability differences between various accelerators connected in the system. There also are dynamic properties like allocated memory, number of associated domains, etc., at any given time. This static and dynamic information is captured in an 'accelerator profile'. An 'accelerator weight' computed from this profile information determines current hardware capabilities and load characteristics for the accelerator. These weights are used to order accelerators in a priority queue maintained within the DomA Scheduler, termed as 'accelerator queue'. For example, the more an accelerator is used, the lower its weight becomes so that it does not get oversubscribed. The accelerator with the highest weight is the most capable and is the first to be considered when a domain requests accelerator use.

Domain profile—domains may be more or less demanding of accelerator resources and more vs. less capable of using them. The 'domain profiles' maintained by Pegasus describe these differences, and they also quantitatively capture domain requirements. Concretely, the current implementation expects credit assignments [7] for each domain that gives it proportional access to the accelerator. Another example is to match the domain's expected memory requirements against the available memory on an accelerator (with CUDA, it is possible to determine this from application metadata). Since the execution properties of domains change over time, domain execution characteristics should be determined dynamically, which would then cause the runtime modification of a domain's accelerator credits and/or access privileges to accelerators. Automated methods for doing so, based on runtime monitoring, are subject of our future work, with initial ideas reported in [8]. This paper lays the groundwork for such research: (1) we show coordination to be a fundamentally useful method for managing future heterogeneous systems, and (2) we demonstrate the importance of these runtime-based techniques and performance advantages derived from their use in a coordinated scheduling environment.

Once a domain has been associated with an accelerator, the **DomA Scheduler** in Figure 2 schedules execution of individual domain requests per accelerator by activating the corresponding domain's aVCPU. For all domains in its ready queue, the 'DomA Scheduler' has complete control over which domain's requests are submitted to the accelerator(s), and it can make such decisions in coordination with the hypervisor's VCPU scheduler, by exchanging relevant accelerator and schedule data. Scheduling in this second phase, can thus be enhanced by *coordinating* the actions of the hypervisor and DomA scheduler(s) present on the platform, as introduced in Figure 1. In addition, certain coordination policies can use the monitoring/feedback module, which currently tracks the average values of wait times for accelerator requests, the goal being to detect SLO (service level objective) violations for guest requests. Various policies supported by the DomA scheduler are described in the following section.

4 Resource Management Policies for Heterogeneity-aware Hypervisors

Pegasus contributes its novel, federated, and heterogeneity-aware scheduling methods to the substantive body of past work in resource management. The policies described below, and implemented by the DomA scheduler, are categorized based on their level of interaction with the hypervisor's scheduler. They range from simple and easily implemented schemes offering basic scheduling properties to coordination-based policies that exploit information sharing between the hypervisor and accelerator subsystems. *Policies are designed to demonstrate the range of achievable coordination between the two scheduler subsystems*

Algorithm 1: Simplified Representation of Scheduling Data and Functions for Credit-based Schemes

```
/* D = Domain being considered */
/* X = Domain cpu or accelerator credits */
/* T = Scheduler timer period */
/* Tc = Ticks assigned to next D */
/* Tm = maximum ticks D gets based on X */
Data: Ready queue RQ_A of domains (D)
/* RQ is ordered by X */
Data: Accelerator queue AccQ of accelerators
/* AccQ is ordered by accelerator weight */

InsertDomainforScheduling(D)
  if D not in RQ_A then
      Tc ← 1, Tm ← X / X_min
      A ← PickAccelerator(AccQ,D)
      InsertDomainInRQ_CreditSorted(RQ_A,D)
  else
      /* D already in some RQ_A */
      if ContextEstablished then
          Tc ← Tm
      else
          Tc ← 1

DomASchedule(RQ_A)
  InsertDomainforScheduling(Curr_Dom)
  D ← RemoveHeadandAdvance(RQ_A)
  Set D's timer period to Tc; Curr_dom ← D
```

and the benefits seen by such coordination for various workloads. The specific property offered by each policy is indicated in square brackets.

4.1 Hypervisor Independent Policies

The simplest methods do not support scheduler federation, limiting their scheduling logic to DomA.

No scheduling in backend (None) [first-come-first-serve]—provides base functionality that assigns domains to accelerators in a round robin manner, but relies on NVIDIA's runtime/driver layer to handle all request scheduling. DomA scheduler plays no role in domain request scheduling. This serves as our baseline.

AccCredit (AccC) [proportional fair-share]—recognizing that domains differ in terms of their desire and ability to use accelerators, accelerator credits are associated with each domain, based on which different domains are polled for different time periods. This makes the time given to a guest proportional to how much it desires to use the accelerator, as apparent in the pseudo-code shown in Algorithm 1, where the requests from the domain at the head of the queue are handled until it finishes its awarded number of ticks. For instance, with credit assignments (Dom1,1024), (Dom2,512), (Dom3,256), and (Dom4,512), the number of ticks will be 4, 2, 1, and 2, respectively.

Because the accelerators used with Pegasus require their applications to explicitly allocate and free accelerator state, it is easy to determine whether or not a domain currently has context (state) established on an accelerator. The DomA scheduler, therefore, interprets a domain in a ContextEstablished state as one that is actively using the accelerator. When in a NoContextEstablished state, a minimum time tick (1) is assigned to the domain for the next scheduling cycle (see Algorithm 1).

Algorithm 2: Simplified Representation of CoSched and AugC Schemes

```
/* RQ_cpu=Per CPU ready q in hypervisor */
/* HS=VCPU-PCPU schedule for next period */
/* X = domain credits */

HypeSchedule(RQ_cpu)
  Pick VCPUs for all PCPUs in system
  ∀D, AugCredit_D = RemainingCredit
  Pass HS to DomA scheduler

DomACoSchedule(RQ_A, HS)
  /* To handle #cpus > #accelerators */
  ∀D ∈ (RQ_A ∩ HS)
    Pick D with highest X
  if D = null then
      /* To improve GPU utilization */
      Pick D with highest X in RQ_A

DomAAugSchedule(RQ_A, HS)
  foreach D ∈ RQ_A do
    Pick D with highest (AugCredit + X)
```

4.2 Hypervisor Controlled Policy

The rationale behind coordinating VCPUs and aVCPUs is that the overall execution time of an application (comprised of both host and accelerator portions) can be reduced if its communicating host and accelerator tasks are scheduled at the same time. We implement one such method described next.

Strict co-scheduling (CoSched) [latency reduction by occasional unfairness]—an alternative to the accelerator-centric policies shown above, this policy gives complete control over scheduling to the hypervisor. Here, accelerator cores are treated as slaves to host cores, so that VCPUs and aVCPUs are scheduled at the same time. This policy works particularly well for latency-sensitive workloads like certain financial processing codes [24] or barrier-rich parallel applications. It is implemented by permitting the hypervisor scheduler to control how DomA schedules aVCPUs, as shown in Algorithm 2. For 'singular VCPUs', i.e., those without associated aVCPUs, scheduling reverts to using a standard credit-based scheme.

4.3 Hypervisor Coordinated Policies

A known issue with co-scheduling is potential unfairness. The following methods have the hypervisor actively participate in making scheduling decisions rather than governing them:

Augmented credit-based scheme (AugC) [throughput improvement by temporary credit boost]—going

beyond the proportionality approach in AccC, this policy uses active coordination between the DomA scheduler and hypervisor (Xen) scheduler in an attempt to better co-schedule domains on a CPU and GPU. To enable coscheduling, the Xen credit-based scheduler provides to the DomA scheduler, as a hint, its CPU schedule for the upcoming period, with remaining credits for all domains in the schedule as shown in Algorithm 2. The DomA scheduler uses this schedule to add temporary credits to the corresponding domains in its list (i.e., to those that have been scheduled for the next CPU time period). This boosts the credits of those domains that have their VCPUs selected by CPU scheduling, thus increasing their chances for getting scheduled on the corresponding GPU. While this effectively co-schedules these domains' CPU and GPU tasks, the DomA scheduler retains complete control over its actions; no domain with high accelerator credits is denied its eventual turn due to this temporary boost.

SLA feedback to meet QoS requirements (SLAF) [feedback-based proportional fair-share]—this is an adaptation of the AccC scheme as shown in Algorithm 1, with feedback control. (1) We start with an SLO defined for a domain (statically profiled) as the expected accelerator utilization—e.g., 0.5sec every second. (2) As shown in Algorithm 1, once the domain moves to a ContextEstablished state, it is polled, and its requests are handled for its assigned duration. In addition, a sum of domain poll time is maintained. (3) Ever so often, all domains associated with an accelerator are scanned for possible SLO violations. Domains with violations are given extra time ticks to compensate, one per scheduling cycle. (4) In high load conditions, there is a trigger that increases accelerator load in order to avoid new domain requests, which in the worst case, forces domains with comparatively low credits to wait longer to get compensated for violations seen by higher credit domains.

For generality in scheduling, we have also implemented: (1) Round robin (RR) [fair-share] which is hypervisor independent, and (2) XenoCredit (XC) [proportional fair-share] which is similar to AccC except it depends on CPU credits assigned to the corresponding VM, making it a hypervisor coordinated policy.

5 System Implementation

The current Pegasus implementation operates with Xen and NVIDIA GPUs. As a result, resource management policies are implemented within the management framework (Section 3.2) run in DomA (i.e., Dom0 in the current implementation), as shown in Figure 2.

Discovering GPUs and guest domains: the management framework discovers all of the GPUs present in the system, assembles their static profiles using *cudaGetDeviceProperties()* [28], and registers them with the Pegasus hypervisor scheduling extensions. When new guest domains are created, Xen adds them to its hypervisor scheduling queue. Our management framework, in turn, discovers them by monitoring XenStore.

Scheduling: the scheduling policies RR, AccC, XC, and SLAF are implemented using timer signals, with one tick interval equal to the hypervisor's CPU scheduling timer interval. There is one timer handler or scheduler for each GPU, just like there is one scheduling timer interrupt per CPU, and this function picks the next domain to run from corresponding GPU's ready queue, as shown in Algorithm 1. AugC and CoSched use a thread in the backend that performs scheduling for each GPU by checking the latest schedule information provided by the hypervisor, as described in Section 4. It then sleeps for one timer interval. The per domain pollers are woken up or put to sleep by scheduling function(s), using real time signals with unique values assigned to each domain. This restricts the maximum number of domains supported by the backend to the Dom0 operating system imposed limit, but results in bounded/prioritized signal delivery times.

Two entirely different scheduling domains, i.e., DomA and the hypervisor, control the two different kinds of processing units, i.e., GPUs and x86 cores. This poses several implementation challenges for the AugC and CoSched policies such as: (1) What data needs to be shared between extensions and the hypervisor scheduler and what additional actions to take, if any, in the hypervisor scheduler, given that this scheduler is in the critical path for the entire system? (2) How do we manage the differences and drifts in these respective schedulers' time periods?

Concerning (1), the current implementation extends the hypervisor scheduler to simply have it share its VCPU-PCPU schedule with the DomA scheduler, which then uses this schedule to find the right VM candidates for scheduling. Concerning (2), there can be a noticeable timing gap between when decisions are made and then enacted by the hypervisor scheduler vs. the DomA extensions. The resulting delay as to when or how soon a VCPU and an aVCPU from same domain are co-scheduled can be reduced with better control over the use of GPU resources. Since NVIDIA drivers do not offer such control, there is notable variation in co-scheduling. Our current remedial solution is to have each aVCPU be executed for 'some time', i.e., to run multiple CUDA call requests, rather than scheduling aVCPUs at a per CUDA call granularity, thereby increasing the possible overlap time with its 'co-related' VCPU. This does not solve the problem, but it mitigates the effects of imprecision, particularly for longer running workloads.

6 Experimental Evaluation

Key contributions of Pegasus are (1) accelerators as first class schedulable entities and (2) coordinated scheduling to provide applications with the high levels of performance sought by use of heterogeneous processing resources. This section first shows that the Pegasus way of virtualizing accelerators is efficient, next demonstrates the importance of coordinated resource management, and finally, presents a number of interesting insights about how diverse coordination (i.e., scheduling) policies can be used to address workload diversity.

Testbed: All experimental evaluations are conducted on a system comprised of (1) a 2.5GHz Xeon quad-core processor with 3GB memory and (2) an NVIDIA 9800 GTX card with 2 GPUs and the v169.09 GPU driver. The Xen 3.2.1 hypervisor and the 2.6.18 Linux kernel are used in Dom0 and guest domains. Guest domains use 512MB memory and 1 VCPU each, the latter pinned to certain physical cores, depending on the experiments being conducted.

6.1 Benchmarks and Applications

Pegasus is evaluated with an extensive set of benchmarks and with emulations of more complex computationally expensive enterprise codes like the web-based image processing application mentioned earlier. Benchmarks include (1) parallel codes requiring low levels of deviation for highly synchronous execution, and (2) throughput-intensive codes. A complete listing appears in Table 1, identifying them as belonging to either the parboil benchmark suite [30] or the CUDA SDK 1.1. Benchmark-based performance studies go beyond running individual codes to using representative code mixes that have varying needs and differences in behavior due to different dataset sizes, data transfer times, iteration complexity, and numbers of iterations executed for certain computations. The latter two are a good measure of GPU 'kernel' size and the degree of coupling between CPUs orchestrating accelerator use and the GPUs running these kernels respectively. Depending on their outputs and the number of CUDA calls made, (1) throughput-sensitive benchmarks are MC, BOp, PI, (2) latency-sensitive benchmarks include FWT, and scientific, and (3) some benchmarks are both, e.g., BS, CP. A benchmark is throughput-sensitive when its performance is best evaluated as the number of some quantity processed or calculated per second, and a benchmark is latency-sensitive when it makes frequent CUDA calls and its execution time is sensitive to potential virtualization overhead and/or delays or 'noise' in accelerator scheduling. The image processing application, termed PicSer, emulates web codes like PhotoSynth. BlackScholes represents financial codes like those run by option trading companies [24].

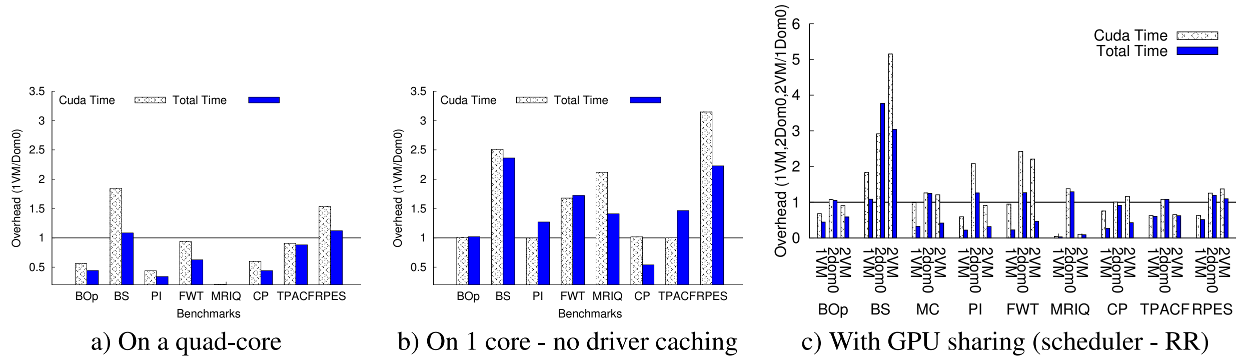
Category	Source	Benchmarks
Financial	SDK	Binomial(BOp), BlackScholes(BS), MonteCarlo(MC)
Media processing	SDK or parboil	ProcessImage(PI)=matrix multiply+DXTC, MRIQ, FastWalshTransform(FWT)
Scientific	parboil	CP, TPACF, RPES

Table 1: Summary of Benchmarks

6.2 GPGPU Virtualization

Virtualization overheads when using Pegasus are depicted in Figures 3(a)–(c), using the benchmarks listed in Table 1. Results show the overhead (or speedup) when running the benchmark in question in a VM vs. when running it in Dom0. The overhead is calculated as the time it takes the benchmark to run in a VM divided by the time to run it in Dom0. We show the overhead (or speedup) for the average total execution time (Total Time) and the average time for CUDA calls (Cuda Time) across 50 runs of each benchmark. Cuda Time is calculated as the time to execute all CUDA calls within the application. Running the benchmark in Dom0 is equivalent to running it in a non-virtualized setting. For the 1VM numbers in Figure 3(a) and (c), all four cores are enabled, and to avoid scheduler interactions, Dom0 and the VM are pinned on separate cores. The experiments reported in Figure 3(b) have only 1 core enabled and the execution times are not averaged over multiple runs, with a backend restart for every run. This is done for reasons explained next. All cases use an equal number of physical GPUs, and Dom0 tests are run with as many cores as the Dom0–1VM case.

An interesting observation about these results is that sometimes, it is better to use virtualized rather than non-virtualized accelerators. This is because (1) the Pegasus virtualization software can benefit from the concurrency seen from using different cores for the guest vs. Dom0 domains, and (2) further advantages are derived from additional caching of data due to a constantly running—in Dom0—backend process and NVIDIA driver. This is confirmed in Figure 3(b), which shows higher overheads when the backend is stopped before every run, wiping out any driver cache information. Also of interest is the speedup seen by say, BOp or PI vs. the performance seen by say, BS or RPES, in Figure 3(a). This is due to an increase in the number of calls per application, seen in BOp/PI vs. BS/RPES, emphasizing the virtualization overhead added to each executed CUDA call. In these cases, the benefits from caching and the presence of multiple cores are outweighed by the per call overhead multiplied by the number of calls made.



a) On a quad-core b) On 1 core - no driver caching c) With GPU sharing (scheduler - RR)

Figure 3: Evaluation of GPU virtualization overhead (lower is better)

6.3 Resource Management

The Pegasus framework for scheduling coordination makes it possible to implement diverse policies to meet different application needs. Consequently, we use multiple metrics to evaluate the policies described in Section 4. They include (1) throughput (*Quantity/sec*) for throughput-sensitive applications, (2) work done (*Quantity work/sec*) (which is the sum of the calculations done over all runs divided by the total time taken), and/or (3) per call latency (Latency) observed for CUDA calls (latency is reported including the CUDA function execution time to account for the fact that we cannot control how the driver orders the requests it receives from Pegasus).

Experimental Methodology: To reduce scheduling interference from the guest OS, each VM runs only a single benchmark. Each sample set of measurements, then, involves launching the required number of VMs, each of which repeatedly runs its benchmark. To evaluate accelerator sharing, experiments use 2, 3, or 4 domains, which translates to configurations with no GPU/CPU sharing, sharing of one GPU and one CPU by two of the three domains, and sharing of two CPUs and both GPUs by pairs of domains, respectively. In all experiments, Dom1 and Dom3 share a CPU as well as a GPU, and so do Dom2 and Dom4, when present. Further, to avoid non-deterministic behavior due to actions taken by the hypervisor scheduler, and to deal with the limited numbers of cores and GPGPUs available on our experimental platform, we pin the domain VCPUs to certain CPUs, depending on the experiment scenario. These CPUs are chosen based on the workload distribution across CPUs (including backend threads in Dom0) and the concurrency requirements of VCPU and aVCPU from the same domain (simply put, VCPU from a domain and the polling thread forming its aVCPU cannot be co-scheduled if they are bound to the same CPU).

For brevity, the results shown in this section focus on the BS benchmark, because of (1) its closeness to real world financial workloads, (2) its tunable iteration count argument that varies its CPU-GPU coupling and can highlight the benefits of coordination, (3) its easily varied data sizes and hence GPU computation complexity, and (4) its throughput as well as latency sensitive nature. Additional reported results are for benchmarks like PicSer, CP and FWT in order to highlight specific interesting/different cases, like those for applications with low degrees of coupling or with high latency sensitivity. For experiments that assign equal credits to all domains, we do not plot RR and AccC, since they are equivalent to XC. Also, we do not show AccC if accelerator credits are equal to Xen credits.

Observations at an early stage of experimentation showed that the CUDA driver introduces substantial variations in execution time when a GPU is shared by multiple applications (shown by the NoVirt graph in Figure 9). This caused us to use a large sample size of 50 runs per benchmark per domain, and we report either the h-spread[1] or work done which is the sum of total output divided by total elapsed time over those multiple runs. For throughput and latency based experiments, we report values for 85% of the runs from the execution set, which prunes some outliers that can greatly skew results and thus, hide the important insights from a particular experiment. These outliers are typically introduced by (1) a serial launch of domains causing the first few readings to show non-shared timings for certain domains, and (2) some domains completing their runs earlier due to higher priority and/or because the launch pattern causes the last few readings for the remaining domains to again be during the unshared period. Hence, all throughput and latency graphs represent the distribution of values across the runs, with a box in the graph representing 50% of the samples around the median (or h-spread) and the lower and upper whiskers encapsulating 85% of the readings, with the minimum and maximum quantities as delimiters. It is difficult, however, to synchronize the launches of domains' GPU kernels with the execution of their threads on CPUs, leading to different orderings of CUDA calls in each run. Hence, to show cumulative performance over the entire experiment, for some experimental results, we also show the 'work done' over all of

[1] http://mathworld.wolfram.com/H-Spread.html

the runs.

Scheduling is needed when sharing accelerators: Figure 3(c) shows the overhead of sharing the GPU when applications are run both in Dom0 and in virtualized guests. In the figure, the 1VM quantities refer to overhead (or speedup) seen by a benchmark running in 1VM vs. when it is run nonvirtualized in Dom0. 2dom0 and 2VM values are similarly normalized with respect to the Dom0 values. 2dom0 values indicate execution times observed for a benchmark when it shares a GPU running in Dom0, i.e., in the absence of GPU virtualization, and 2VM values indicate similar values when run in two guest VMs sharing the GPU. For the 2VM case, the Backend implements RR, a scheduling policy that is completely fair to both VMs, and their CPU credits are set to 256 for equal CPU sharing. These measurements show that (1) while the performance seen by applications suffers from sharing (due to reduced accelerator access), (2) a clear benefit is derived for most benchmarks from using even a simple scheduling method for accelerator access. This is evident from the virtualized case that uses a round robin scheduler, which shows better performance compared with the nonvirtualized runs in Dom0 for most benchmarks, particularly the ones with lower numbers of CUDA call invocations. This shows that *scheduling is important to reduce contention in the NVIDIA driver* and thus helps minimize the resulting performance degradation. Measurements report Cuda Time and Total Time, which is the metric used in Figures 3(a)–(b).

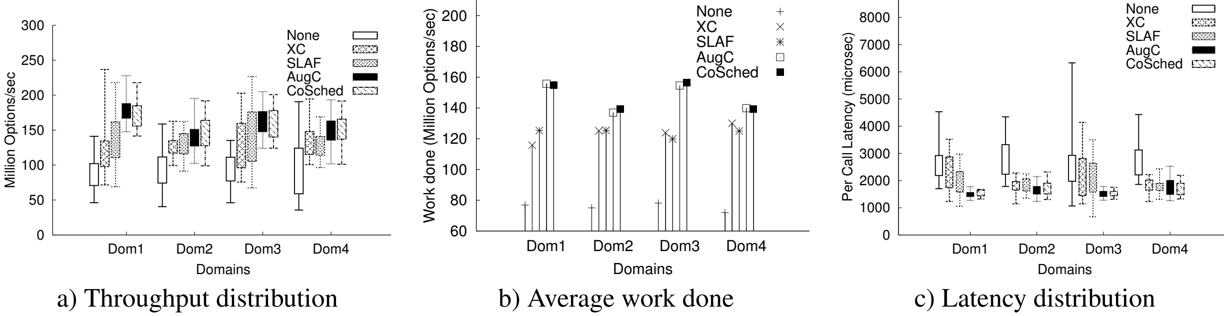
We speculate that sharing overheads could be reduced further if Pegasus was given more control over the way GPU resources are used. Additional benefits may arise from improved hardware support for sharing the accelerator, as expected for future NVIDIA hardware [27].

Coordination can improve performance: With encouraging results from the simple RR scheduler, we next experiment with the more sophisticated policies described in Section 4. In particular, we use BlackScholes (outputs options and hence its throughput is given by Options/sec) which, with more than 512 compute kernel launches and a large number of CUDA calls, has a high degree of CPU-GPU coupling. This motivates us to also report the latency numbers seen by BS.

An important insight from these experiments is that coordination in scheduling is particularly important for tightly coupled codes, as demonstrated by the fact that our base case, None, shows large variations and worse overall performance, whereas AugC and CoSched show the best performance due to their higher degrees of coordination. Figures 4(a)–(c) show that these policies perform well even when domains have equal credits. The BlackScholes run used in this experiment generates 2 million options over 512 iterations in all our domains. Figure 4(a) shows the distribution of throughput values in Million options/sec, as explained earlier. While XC and SLAF see high variation due to higher dependence on driver scheduling and no attempt for CPU and GPU coscheduling, they still perform at least 33% better than None when comparing the medians. AugC and CoSched add an additional 4%–20% improvement as seen from Figure 4(a). The higher performance seen with Dom1 and Dom3 for total work done in Figure 4(b) in case of AugC and CoSched is because of the lower signaling latency seen by the incoming and outgoing domain backend threads, due to their co-location with the scheduling thread and hence, the affected call ordering done by the NVIDIA driver (which is beyond our control).

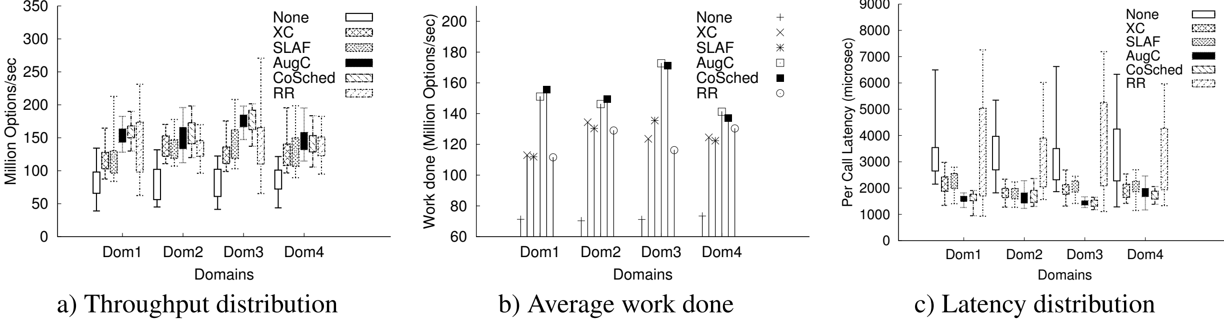
Beyond the improvements shown above, future deployment scenarios in utility data centers suggest the importance of supporting prioritization of domains. This is seen by experiments in which we modify the credits assigned to a domain, which can further improve performance (see Figure 5). We again use BlackScholes, but with Domain credits as (1) (Dom1,256), (2) (Dom2,512), (3) (Dom3,1024), and (4) (Dom4,256), respectively. The effects of such scheduling are apparent from the fact that, as shown in Figure 5(b), Dom3 succeeds in performing 2.4X or 140% more work when compared with None, with its minimum and maximum throughput values showing 3X to 2X improvement respectively. This is because domains sometimes complete early (e.g., Dom3 completes its designated runs before Dom1) which then frees up the accelerator for other domains (e.g., Dom1) to complete their work in a mode similar to non-shared operation, resulting in high throughput. The 'work done' metric captures this because average throughput is calculated for the entire application run. Another important point seen from Figure 5(c) is that the latency seen by Dom4 varies more as compared to Dom2 for say AugC because of the temporary unfairness resulting from the difference in credits between the two domains. A final interesting note is that scheduling becomes less important when accelerators are not highly utilized, as evident from other measurements not reported here.

Coordination respects proportional credit assignments: The previous experiments use equal amounts of accelerator and CPU credits, but in general, not all guest VMs need equal accelerator vs. general purpose processor resources. We demonstrate the effects of discretionary credit allocations using the BS benchmark, since it is easily configured for variable CPU and GPU execution times, based on the expected number of call and put options and the number of iterations denoted by BS⟨#options,#iterations⟩. Each domain is assigned different GPU and CPU credits denoted by Dom#⟨AccC,XC,SLA proportion⟩. This results in the configuration for this experiment being: Dom1⟨1024,256,0.2⟩ running BS⟨2mi,128⟩,



a) Throughput distribution b) Average work done c) Latency distribution

Figure 4: Performance of different scheduling schemes [BS]—equal credits for four guests



a) Throughput distribution b) Average work done c) Latency distribution

Figure 5: Performance of scheduling schemes [BS]—Credits: Dom1=256, Dom2=512, Dom3=1024, Dom4=256

Dom2⟨1024,1024,0.8⟩ running BS⟨2mi,512⟩, Dom3⟨256,1024,0.8⟩ running BS⟨0.8mi,512⟩, and Dom4⟨768,256,0.2⟩ running BS⟨1.6mi,128⟩, where mi stands for million.

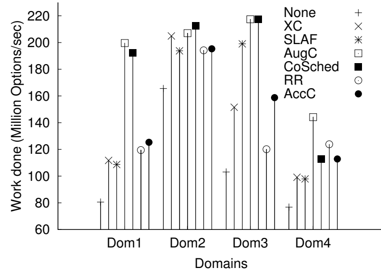
Results depicting 'total work done' in Figure 6 demonstrate that coordinated scheduling methods AugC and CoSched deal better with proportional credit assignments. Results show that domains with balanced CPU and GPU credits are more effective in getting work done—Dom2 and Dom3 (helped by high Xen credits)—than others. SLAF shows performance similar to CoSched and AugC due to its use of a feedback loop that tries to attain 80% utilization for Dom2 and Dom3 based on Xen credits. Placement of Dom4 with a high credit domain Dom2 somewhat hurts its performance, but its behavior is in accordance with its Xen credits and SLAF values, and it still sees a performance improvement of at least 18% compared to XC (lowest performance improvement among all scheduling schemes for the domain) with None. Dom1 benefits from coordination due to earlier completion of Dom3 runs, but is affected by its low CPU credits for the rest of the schemes.

One lesson from these runs is that the choice of credit assignment should be based on the expected outcome and the amount of work required by the application. How to make suitable choices is a topic for future work, particularly focusing on the runtime changes in application needs and behavior. We also realize that we cannot control the way the driver ultimately schedules requests possibly introducing high system noise and limiting achievable proportionality.

Coordination is important for latency sensitive codes: Figure 8 corroborates our earlier statement about the particular need for coordination with latency-intolerant codes. When FWT is run in all domains, first with equal CPU and GPU credits, then with different CPU credits per domain, it is apparent that 'None' (no scheduling) is inappropriate. Specifically, as seen in Figure 8, all scheduling schemes see much lower latencies and latency variations than None. Another interesting point is that the latencies seen for Dom2 and Dom3 are almost equal, despite a big difference in their credit values, for all schemes except RR (which ignores credits). This is because latencies are reduced until reaching actual virtualization overheads and thereafter, are no longer affected by differences in credits per domain. The other performance effects seen for total time can be attributed to the order in which calls reach the driver.

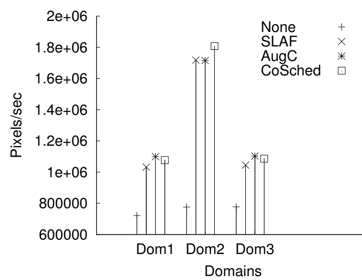
Scheduling is not always effective: There are situations in which scheduling is not effective. We have observed this when a workload is very short lived or when it shows a high degree of variation, as shown in Figure 9. These variations can be attributed to driver processing, with evidence for this attribution being that the same variability is observed in the absence of Pegasus, as seen from the 'NoVirt' bars in the figure. An idea for future work with Pegasus is to explicitly evaluate this via runtime monitoring, to establish and track penalties due to sharing, in order to then adjust scheduling to avoid such penalties whenever possible.

Scheduling does not affect performance in the absence of sharing; scheduling overheads are low: When

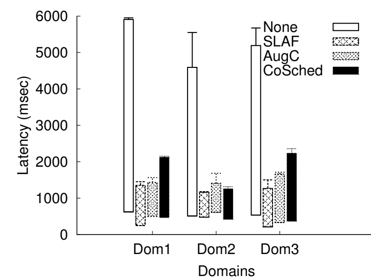


Different Xen and accelerator credits for domains

Figure 6: Performance of different scheduling schemes [BS]

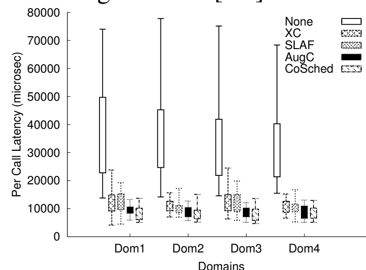


a) PicSer Work done

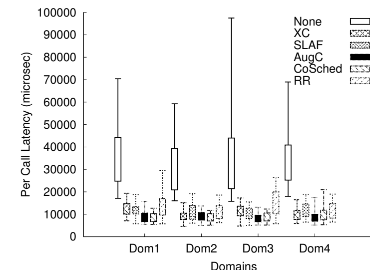


b) Latency for GPU processing

Figure 7: Performance of selected scheduling schemes for real world benchmark



a) All Doms = 256



b) Dom1=256,Dom2=1024,Dom3=2048,Dom4=256

Figure 8: Average latencies seen for [FWT]

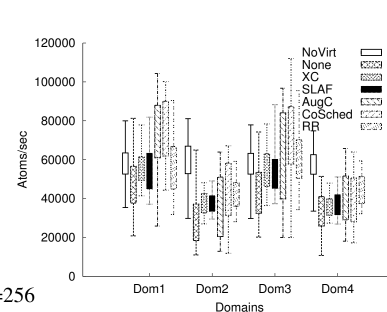


Figure 9: [CP] with sharing

using two, three, and four domains assigned equal credits, with a mix of different workloads, our measurements show that in general, scheduling works well and exhibits little variation, especially in the absence of accelerator sharing. While those results are omitted due to lack of space, we do report the worst case scheduling overheads seen per scheduler call in Table 2, for different scheduling policies. MS in the table refers to the Monitor and Sweep thread responsible for monitoring credit value changes for guest VMs and cleaning out state for non-existing VMs. Xen kernel refers to the changes made to the hypervisor CPU scheduling method. Acc0 and Acc1 refer to the schedulers (for timer based schemes like RR, XC, SLAF) in our dual accelerator testbed. Hype refers to the user level thread run for policies like AugC and CoSched for coordinating CPU and GPU activities.

Policy	MS (μsec)	Xen Kernel (μsec)	Acc0/Hype (μsec)	Acc1 (μsec)
None	272	0.85	0	0
XC	1119	0.85	507	496
AugC	1395	0.9	3.36	0
SLAF	1101	0.95	440	471
CoSched	1358	0.825	2.71	0

Table 2: Backend scheduler overhead

As seen from the table, the Pegasus backend components have low overhead. For example, XC sees ∼0.5ms per scheduler call per accelerator, compared to a typical execution time of CUDA applications of between 250ms to 5000ms and with typical scheduling periods of 30ms. The most expensive component, with an overhead of ∼1ms, is MS, which runs once every second.

Scheduling complex workloads: When evaluating scheduling policies with the PicSer application, we run three dual-core, 512MB guests on our testbed. One VM (Dom2) is used for priority service and hence given 1024 credits and 1 GPU, while the remaining two are assigned 256 credits, and they share the second GPU. VM2 is latency-sensitive, and all of the VMs care about throughput. Scheduling is important because CPUs are shared by multiple VMs. Figure 7(a) shows the average throughput (Pixels/sec to incorporate different image sizes) seen by each VM with four different policies. We choose AugC and CoSched to highlight the co-scheduling differences. None is to provide a baseline, and SLAF is an enhanced version of all of the credit based schemes. AugC tries to improve the throughput of all VMs, which results in a somewhat lower value for Dom2. CoSched gives priority to Dom2 and can penalize other VMs, as evident from the GPU latencies shown in Figure 7(b). 'No scheduling' does not perform well. More generally, it is clear that coordinated scheduling can be effective in meeting the requirements of multi-VM applications sharing CPU and GPU resources.

6.4 Discussion

Experimental results show that the Pegasus approach efficiently virtualizes GPUs and in addition, can effectively schedule their use. Even basic accelerator request scheduling can improve sharing performance, with additional benefits derived from active scheduling coordi-

nation schemes. Among these methods, XC can perform quite well, but fails to capitalize on CPU-GPU coordination opportunities for tightly coupled benchmarks. SLAF, when applied to CPU credits, has a smoothing effect on the high variations of XC, because of its feedback loop. For most benchmarks, especially those with a high degree of coupling, AugC and CoSched perform significantly better that other schemes, but require small changes to the hypervisor. More generally, scheduling schemes work well in the absence of over-subscription, helping regulate the flow of calls to the GPU. Regulation also results in lowering the degrees of variability caused by un-coordinated use of the NVIDIA driver.

AugC and CoSched, in particular, constitute an interesting path toward realizing our goal of making accelerators first class citizens, and further improvements to those schemes can be derived from gathering additional information about accelerator resources. There is not, however, a single 'best' scheduling policy. Instead, there is a clear need for diverse policies geared to match different system goals and to account for different application characteristics.

Pegasus scheduling uses global platform knowledge available at hypervisor level, and its implementation benefits from hypervisor-level efficiencies in terms of resource access and control. As a result, it directly addresses enterprise and cloud computing systems in which virtualization is prevalent. Yet, clearly, methods like those in Pegasus can also be realized at OS level, particularly for the high performance domain where hypervisors are not yet in common use. In fact, we are currently constructing a CUDA interposer library for non-virtualized, native guest OSes, which we intend to use to deploy scheduling solutions akin to those realized in Pegasus at large scale on the Keeneland machine.

7 Related Work

The importance of dealing with the heterogeneity of future multi-core platforms is widely recognized. Cypress [10] has expressed the design principles for hypervisors actually realized in Pegasus (e.g., partitioning, localization, and customization), but Pegasus also articulates and evaluates the notion of coordinated scheduling. Multikernel [4] and Helios [26] change system structures for multicores, advocating distributed system models and satellite kernels for processor groups, respectively. In comparison, Pegasus retains the existing operating system stack, then uses virtualization to adapt to diverse underlying hardware, and finally, leverages the federation approach shown scalable in other contexts to deal with multiple resource domains.

Prior work on GPU virtualization has used the OpenGL API [21] or 2D-3D graphics virtualization (DirectX, SVGA) [9]. In comparison, Pegasus operates on entire computational kernels more readily co-scheduled with VCPUs running on general purpose CPUs. This approach to GPU virtualization is outlined in an earlier workshop paper, termed GViM [13], which also presents some examples that motivate the need for QoS-aware scheduling. In comparison, this paper thoroughly evaluates the approach, develops and explores at length the notion of coordinated scheduling and the scheduling methods we have found suitable for GPGPU use and for latency- vs. throughput-intensive enterprise codes.

While similar in concept, Pegasus differs from coordinated scheduling at the data center level, in that its deterministic methods with predictable behavior are more appropriate at the fine-grained hypervisor level than the loosely-coordinated control-theoretic or statistical techniques used in data center control [20]. Pegasus co-scheduling differs in implementation from traditional gang scheduling [36] in that (1) it operates across multiple scheduling domains, i.e., GPU vs. CPU scheduling, without direct control over how each of those domains schedules its resources, and (2) because it limits the idling of GPUs, by running workloads from other aVCPUs when a currently scheduled VCPU does not have any aVCPUs to run. This is appropriate because Pegasus co-scheduling schemes can afford some skew between CPU and GPU components, since their aim is not to solve the traditional locking issue.

Recent efforts like Qilin [23] and predictive runtime code scheduling [16] both aim to better distribute tasks across CPUs and GPUs. Such work is complementary and could be used combined with the runtime scheduling methods of Pegasus. Upcoming hardware support for accelerator-level contexts, context isolation, and context-switching [27] may help in terms of load balancing opportunities and more importantly, it will help improve accelerator sharing [9].

8 Conclusions and Future Work

This paper advocates making all of the diverse cores of heterogeneous manycore systems into first class schedulable entities. The Pegasus virtualization-based approach for doing so, is to abstract accelerator interfaces through virtualization and then devise scheduling methods that coordinate accelerator use with that of general purpose host cores. The approach is applied to a combined NVIDIA- and x86-based GPGPU multicore prototype, enabling multiple guest VMs to efficiently share heterogenous platform resources. Evaluations using a large set of representative GPGPU benchmarks and computationally intensive web applications result in insights that include: (1) the need of coordination when sharing accelerator resources, (2) its critical importance for applications that frequently interact across the CPU-GPU boundary, and (3) the need for diverse policies when co-

ordinating the resource management decisions made for general purpose vs. accelerator cores.

Certain elements of Pegasus remain under development and/or are subject of future work. Admission control methods can help alleviate certain problems with accelerator sharing, such as those caused by insufficient accelerator resources (e.g., memory). Runtime load balancing across multiple accelerators would make it easier to deal with cases in which GPU codes do not perform well when sharing accelerator resources. Static profiling and runtime monitoring could help identify such codes. There will be some limitations in load balancing, however, because of the prohibitive costs in moving the large amounts of memory allocated on completely isolated GPU resources. This restricts load migration to cases in which the domain has no or little state on a GPU. As a result, the first steps in our future work will be to provide Pegasus scheduling methods with additional options for accelerator mappings and scheduling, by generalizing our implementation to use both local and non-local accelerators (e.g., when they are connected via high end network links like Infiniband). Despite these shortcomings, the current implementation of Pegasus not only enables multiple VMs to efficiently share accelerator resources, but also achieves considerable performance gains with its coordination methods.

Acknowledgments

We would like to thank our shepherd Mr. Muli Ben-Yehuda and other anonymous reviewers for their insight on improving the paper. We would also like to thank Priyanka Tembey, Romain Cledat and Tushar Kumar for their feedback and assistance with the paper.

References

[1] AMAZON INC. High Performance Computing Using Amazon EC2. http://aws.amazon.com/ec2/hpc-applications/.

[2] BAKHODA, A., YUAN, G. L., FUNG, W. W., ET AL. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS* (Boston, USA, 2009).

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., ET AL. Xen and the art of virtualization. In *SOSP* (Bolton Landing, USA, 2003).

[4] BAUMANN, A., BARHAM, P., DAGAND, P. E., ET AL. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP* (Big Sky, USA, 2009).

[5] BERGMANN, A. The Cell Processor Programming Model. In *LinuxTag* (2005).

[6] BORDAWEKAR, R., BONDHUGULA, U., AND RAO, R. Believe It or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application! Tech. Report RC24982, IBM T. J. Watson Research Center,, 2010.

[7] CHISNALL, D. *The Definitive Guide to the Xen Hypervisor*, 1st ed. Prentice Hall, 2008.

[8] DIAMOS, G., AND YALAMANCHILI, S. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems. In *HPDC Hot Topics* (Boston, USA, 2008).

[9] DOWTY, M., AND SUGERMAN, J. GPU Virtualization on VMware's Hosted I/O Architecture. In *WIOV* (San Diego, USA, 2008).

[10] FEDOROVA, A., KUMAR, V., KAZEMPOUR, V., ET AL. Cypress: A Scheduling Infrastructure for a Many-Core Hypervisor. In *MMCS* (Boston, USA, 2008).

[11] GOVIL, K., TEODOSIU, D., HUANG, Y., ET AL. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *SOSP* (Charleston, USA, 1999).

[12] GUEVARA, M., GREGG, C., HAZELWOOD, K., ET AL. Enabling Task Parallelism in the CUDA Scheduler. In *PMEA* (Raleigh, USA, 2009).

[13] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., ET AL. GViM: GPU-accelerated Virtual Machines. In *HPCVirt* (Nuremberg, Germany, 2009).

[14] GUPTA, V., XENIDIS, J., TEMBEY, P., ET AL. Cellule: Lightweight Execution Environment for Accelerator-based Systems. Tech. Rep. GIT-CERCS-10-03, Georgia Tech, 2010.

[15] HEINIG, A., STRUNK, J., REHM, W., ET AL. *ACCFS - Operating System Integration of Computational Accelerators Using a VFS Approach*, vol. 5453. Springer Berlin, 2009.

[16] JIMÉNEZ, V. J., VILANOVA, L., GELADO, I., ET AL. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *HiPEAC* (Paphos, Cyprus, 2009).

[17] JOHNSON, C., ALLEN, D. H., BROWN, J., ET AL. A Wire-Speed PowerTM Processor: 2.3GHz 45nm SOI with 16 Cores and 64 Threads. In *ISSCC* (San Francisco, USA, 2010).

[18] KERR, A., DIAMOS, G., AND YALAMANCHILI, S. A Characterization and Analysis of PTX Kernels. In *IISWC* (Austin, USA, 2009).

[19] KHRONOS GROUP. The OpenCL Specification. http://tinyurl.com/OpenCL08, 2008.

[20] KUMAR, S., TALWAR, V., KUMAR, V., ET AL. vManage: Loosely Coupled Platform and Virtualization Management in Data Centers. In *ICAC* (Barcelona, Spain, 2009).

[21] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., ET AL. VMM-independent graphics acceleration. In *VEE* (San Diego, CA, 2007).

[22] LANGE, J., PEDRETTI, K., DINDA, P., ET AL. Palacios: A New Open Source Virtual Machine Monitor for Scalable High Performance Computing. In *IPDPS* (Atlanta, USA, 2010).

[23] LUK, C.-K., HONG, S., AND KIM, H. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Micro-42* (New York, USA, 2009).

[24] MARCIAL, E. The ICE Financial Application. http://www.theice.com, 2010. Private Communication.

[25] MICROSOFT CORP. What is Photosynth? http://photosynth.net/about.aspx, 2010.

[26] NIGHTINGALE, E. B., HODSON, O., MCILROY, R., ET AL. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP* (Big Sky, USA, 2009).

[27] NVIDIA CORP. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://tinyurl.com/nvidia-fermi-whitepaper.

[28] NVIDIA CORP. NVIDIA CUDA Compute Unified Device Architecture. http://tinyurl.com/cx3tl3, 2007.

[29] RAJ, H., AND SCHWAN, K. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC* (Monterey, USA, 2007).

[30] RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., ET AL. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP* (Salt Lake City, USA, 2008).

[31] SHIMPI, A. L. Intel's Sandy Bridge Architecture Exposed. http://tinyurl.com/SandyBridgeArch.

[32] SNAPFISH. About Snapfish. http://www.snapfish.com.

[33] SNAVELY, N., SEITZ, S. M., AND SZELISKI, R. Modeling the World from Internet Photo Collections. *International Journal of Computer Vision 80*, 2 (2008).

[34] TURNER, J. A. The Los Alamos Roadrunner Petascale Hybrid Supercomputer: Overview of Applications, Results, and Programming. Roadrunner Technical Seminar Series, 2008.

[35] VETTER, J., GLASSBROOK, D., DONGARRA, J., ET AL. Keeneland - Enabling Heterogeneous Computing For The Open Science Community. http://tinyurl.com/KeenelandSC10, 2010.

[36] VMWARE CORP. VMware vSphere 4: The CPU Scheduler in VMware ESX 4. http://tinyurl.com/ykenbjw, 2009.