Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers

Quan Chen[†1] Hailong Yang[‡1] Minyi Guo[†]
Ram Srivatsa Kannan[⋆] Jason Mars[⋆] Lingjia Tang[⋆]
[†]Department of Computer Science and Engineering, Shanghai Jiao Tong University
[‡]School of Computer Science and Engineering, Beihang University
[⋆]Department of Computer Science, University of Michigan - Ann Arbor
{chen-quan, guo-my}@cs.sjtu.edu.cn, hailong.yang@buaa.edu.cn, {ramsri, profmars, lingjia}@umich.edu

Abstract

Guaranteeing Quality-of-Service (QoS) of latency-sensitive applications while improving server utilization through application co-location is important yet challenging in modern datacenters. The key challenge is that when applications are co-located on a server, performance interference due to resource contention can be detrimental to the application QoS. Although prior work has proposed techniques to identify "safe" co-locations where application QoS is satisfied by predicting the performance interference on multicores, no such prediction technique on accelerators such as GPUs.

In this work, we present Prophet, an approach to precisely predict the performance degradation of latency-sensitive applications on accelerators due to application co-location. We analyzed the performance interference on accelerators through a real system investigation and found that unlike on multicores where the key contentious resources are shared caches and main memory bandwidth, the key contentious resources on accelerators are instead processing elements, accelerator memory bandwidth and PCIe bandwidth. Based on this observation, we designed interference models that enable the precise prediction for processing element, accelerator memory bandwidth and PCIe bandwidth contention on real hardware. By using a novel technique to forecast solo-run execution traces of the co-located applications using interference models, Prophet can accurately predict the performance degradation of latency-sensitive applications on non-preemptive accelerators. Using Prophet, we can identify "safe" co-locations on accelerators to improve utilization without violating the QoS target. Our evaluation shows that Prophet can predict the performance degradation with an average prediction error 5.47% on real systems. Meanwhile, based on the prediction, Prophet achieves accelerator utilization improvements of 49.9% on average while maintaining the QoS target of latency-sensitive applications.

Keywords Quality-of-Service Prediction; Non-Preemptive Accelerators; Warehouse-Scale Computers

ASPLOS '17, April 08-12, 2017, Xi'an, China
© 2017 ACM. ISBN 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037700

1. Introduction

Latency-sensitive applications (**LS applications**) running on *Warehouse-Scale Computers* have stringent *Quality-of-Service* (**QoS**) requirements to provide the satisfactory user experience. To satisfy a QoS target, servers hosting LS applications are commonly over-provisioned and remain underutilized [7, 16, 34]. With accelerators gaining popularity in large-scale datacenters, similar over-provisioning is also applied to accelerators [19, 20, 43]. In addition, the diurnal user load pattern often leaves servers underutilized during the off-peak hours [8, 34], which exacerbates the low utilization problem in datacenters. Co-locating multiple applications onto fewer hardware resources alleviates the low utilization problem [15, 33]. However, the contention on shared resources can cause severe performance interference, which is detrimental to meeting QoS targets [16, 31, 33]. Therefore, it is critical to improve resource utilization without violating the QoS target of LS applications.

A large body of prior work [14, 16, 32, 33, 46, 50, 53] focus on precise prediction of performance interference due to the shared resource contention on CPUs. Such QoS prediction is then used to identify "safe" co-locations where predicted interference to the LS application is within a safe

[1] Part of the work was conducted as a postdoc fellow of Clarity Lab at the University of Michigan.

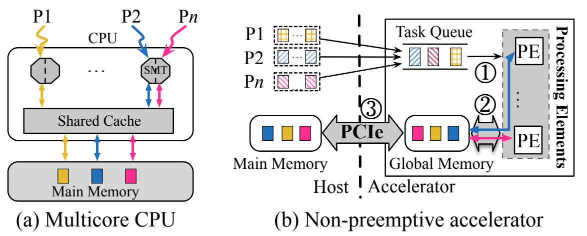


Figure 1: Performance interference on multicore CPU and non-preemptive accelerator.

margin. These "safe" co-locations can effectively improve server utilization while satisfying QoS targets. However, existing techniques only consider interference in the context of multicore CPUs, and are not applicable to non-preemptive accelerators such as GPUs. As shown in Figure 1(a), prior work focuses on shared resources on multicores including shared caches, memory controller and *simultaneous multi-threading* (SMT). But, as shown in Figure 1(b), for non-preemptive accelerators, performance interference originates from a different set of shared resources including ① **processing elements** (**PE**s), ② **global memory bandwidth**, and ③ **PCIe bandwidth**. To achieve precise QoS prediction for these accelerators, it is necessary to design novel approaches to account for these new sources of contention.

Through real system investigation, we found that precisely predicting the QoS interference between co-located applications on non-preemptive accelerators is especially challenging due to the complex effects and interactions of several runtime factors affecting the contention behaviors. As shown in Figure 1(b), when multiple applications submit tasks to the same accelerator, performance interference originates from the contention on PE, global memory bandwidth and PCIe bandwidth. The contention on PEs depends on the extent computational tasks overlap with each other. The overlapping in turn depends on several factors such as occupancy and the amount of computational resources. The effect of the overlapping is unknown as most accelerators use proprietary driver software for scheduling tasks. If multiple tasks run on different PEs concurrently, they access data from the shared global memory and contend for the limited global memory bandwidth. Concurrent data transfers generate contention on the PCIe bandwidth, which is determined by the type of data transfers (shared or exclusive) as well as the amount of overlapping transfers. Moreover, the complex interaction of interference on different resources further exacerbates the difficulty to precisely predict the QoS of LS applications. For instance, PCIe interference delays the start time of task execution, which in turn introduces variability into the overlapping proportion of concurrent task execution and eventually alters the contention characteristics.

These findings motivate our design for a new methodology, **Prophet**, to precisely predict the performance degradation for co-locations on non-preemptive accelerators. Prophet carefully models the performance interference on PEs, global memory bandwidth and PCIe bandwidth. The performance interference model for PEs considers several runtime factors such as occupancy and utilization of PEs to calculate the portion of the tasks capable for concurrent execution and predict the queuing delay due to resource contention. The global memory bandwidth interference model considers the bandwidth requirements of concurrent tasks and the number of PEs each task occupies to calculate task execution slowdown due to bandwidth contention. The PCIe performance interference model distinguishes different data transfer tasks and quantitatively identifies the potential bandwidth contention, especially when multiple data transfer tasks execute concurrently. To capture the contention interaction between the shared resources, Prophet uses a holistic approach combining interference models for PEs, global memory bandwidth and PCIe bandwidth to synthesize and forecast the execution trace of co-located applications. With the precise prediction of performance interference using the above approach, Prophet identifies the applications that can be safely co-located on accelerators to improve utilization while satisfying the QoS target of LS applications.

Specifically, the contributions of this paper are as follows:

- **Investigating Performance Interference on Accelerators Comprehensively-** The real system investigation studies the performance interference across several shared resources on accelerators and demonstrates that it is fundamentally different from CPUs. In addition, an in-depth analysis is provided to understand the contention characteristics across different resources.
- **Modeling Performance Interference across Various Resources -** Novel interference models are proposed to predict the contention behaviors on PEs, global memory bandwidth and PCIe bandwidth. We consider the overlapping execution on the PEs as well as different types of data transfer tasks in our models to accurately estimate the amount of interference on individual resource.
- **Predicting Performance Interference through Synthesizing the Co-located Execution -** We present a synthesis approach that leverages all the interference models to model the compound effect of interference across multiple resources during the entire execution of the co-located applications. By synthesizing the execution traces, Prophet is able to precisely predict the performance degradation under co-location.
- **Improving Accelerator Utilization -** Based on the precise interference prediction from Prophet, we can steer the application scheduling on accelerators with "safe" co-locations that achieve utilization improvement without violating the QoS of LS applications.

2. Real System Investigation

In this section, we investigate the types of resource contention that lead to significant performance interference for

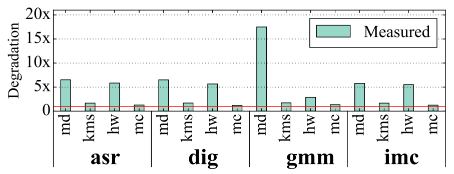


Figure 2: Performance degradation of compute-intensive LS applications when they are co-located with compute-intensive batch applications.

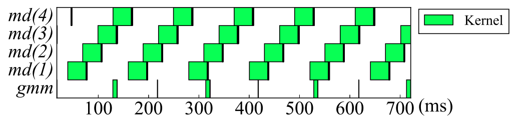


Figure 3: Snapshot of task execution on real system GPU.

co-located applications on accelerators using real systems. Specifically, we seek to answer the following questions:

- Will LS applications suffer from large performance degradation due to application co-locations on GPUs?
- If so, what are the root causes of the large interference?
- What are the key requirements for designing a mechanism to precisely predict such performance interference?

2.1 Experimental Setup

As prior work has shown that GPUs have become evermore important for future datacenter scaling [23, 43], we use GPUs as our target accelerator in this work. Our LS applications include an emerging *Intelligent Personal Assistant* (IPA) application Sirius [20] and a *Deep Neural Network* (DNN) service DjiNN [19]. These applications are executed as resident services on the accelerator. There is a stringent QoS target for these applications. We use *Rodinia* [12] as batch applications. In our experiment, LS applications and batch applications submit tasks to the same accelerator concurrently. In order to investigate the impact of various interference on the performance degradation, we use both compute-intensive and PCIe-intensive applications in our investigation. More details of the experimental hardware and benchmarks are described in Section 7.1.

2.2 Task Execution Interference

The contention on PE and global memory bandwidth often causes task execution interference. We first investigate the impact of the interference on the performance of LS applications. For this experiment, we use compute-intensive co-runners with negligible PCIe bandwidth requirements.

Figure 2 presents the performance degradation of LS applications when they are co-located with compute-intensive batch applications. In this figure, the *x*-axis shows the co-location pairs while the *y*-axis presents the performance degradation of the LS application. For example, *md* under

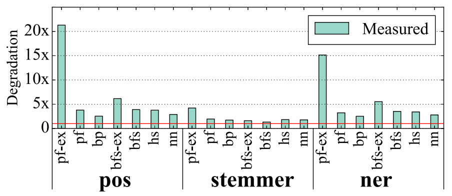


Figure 4: Performance degradation of PCIe-intensive LS applications when they are co-located with PCIe-intensive batch applications.

asr presents the performance degradation of the LS application *asr* when co-located with *lavaMD*. As shown in the figure, different batch applications cause varying amounts of performance degradation to the co-located LS applications, ranging from 1.1x to 17.5x. The performance degradation in this experiment is mainly due to the queuing delay at the PEs of the accelerator, and the global memory bandwidth contention between PEs.

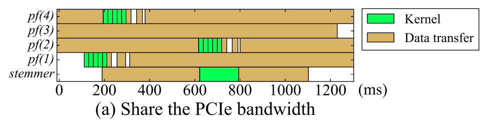
Because existing commodity accelerators such as GPUs do not support task preemption between co-located applications to avoid the costly context switch overhead [39], tasks from LS applications need to wait for all previously submitted tasks to complete before it can be executed. This queuing delay leads to large performance degradation when the co-located applications contend for PEs. Modern accelerators, such as Nvidia Kepler GPUs allow multiple tasks run on different PEs concurrently [1]. In this case, the concurrent tasks access data from global memory simultaneously and the contention may also degrade the performance of co-located applications.

Figure 3 presents an example execution trace of compute-intensive tasks on a Nvidia GPU. We can see that the tasks of LS application *gmm* are delayed significantly by the tasks of batch application *md*. Although part of the execution has concurrent tasks by utilizing Nvidia's *Multi-Process Service* (MPS) [1], the queuing delay created by PE contention is not fully eliminated. Meanwhile, with MPS, *gmm*'s tasks and *md*'s tasks may run concurrently. In this case, the contention on the global memory bandwidth between concurrent tasks also contributes to the performance interference.

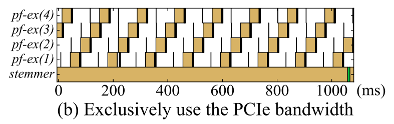
2.3 Data Transfer Interference

Besides the contention for PEs and global memory bandwidth, co-located applications also contend for the limited PCIe bandwidth when transferring data between the host and the accelerator. To investigate whether and how PCIe bandwidth contention affects the QoS of LS applications, we colocate PCIe-intensive applications, and present the performance degradation in Figure 4. We observe that the contention on PCIe bandwidth also leads to significant performance degradation of LS applications.

When multiple applications transfer data through PCIe simultaneously and share bandwidth, the reduced bandwidth



(a) Share the PCIe bandwidth



(b) Exclusively use the PCIe bandwidth

Figure 5: Snapshots of data transfers over PCIe bus.

slows down the data transfer, resulting in large performance degradation. In addition, certain programming interfaces (e.g., CUDA [36]) allow programmers to create data transfer tasks that fully occupy the PCIe bandwidth (e.g., memory copy tasks from/to *pinned memory* in CUDA), causing even more significant slowdown to other co-runners. As shown in Figure 4, batch applications that use *bandwidth-exclusive data transfer tasks* (**BE tasks**) such as *pf-ex* and *bfs-ex* result in much larger performance degradation than the ones that use *bandwidth-sharing data transfer tasks* (**BS tasks**).

Figure 5 presents the contention behaviors for PCIe bandwidth between latency-sensitive (*stemmer*) and batch (*pf*) applications. In Figure 5(a), both applications are using BS tasks. Although the performance of LS application degrades due to contention, both applications can transfer data concurrently. On the other hand, as shown in Figure 5(b), *pf-ex* uses bandwidth-exclusive transfer, and causes much higher delay (2.7x) to *stemmer* compared to *pf*.

2.4 Challenges for Precise QoS Prediction on Non-Preemptive Accelerators

Our real system investigation has shown that task execution interference and data transfer interference together result in significant performance degradation. However, precisely predicting the degradation is not trivial due to the complex interference behaviors on non-preemptive accelerators. Specifically, there are several key challenges to achieve precise prediction.

(1) Task execution interference varies due to the overlapped task execution - The amount of degradation depends on the overlapped/concurrent execution during runtime between co-located applications (PE contention). Long queuing delay occurs to the LS application if the tasks cannot run concurrently due to the large occupancy. Otherwise, when tasks run concurrently, the global memory bandwidth contention could increase the duration of concurrent tasks.

(2) Data transfer interference varies - There are two types of data transfer tasks on accelerators that lead to different PCIe contention behaviors. BS task allows sharing the PCIe bandwidth between co-located applications, however bandwidth-exclusive data transfer preempts all other data transfer until it completes. In order to predict the performance interference due to PCIe bandwidth contention, it is necessary to model the contention behaviors of different types of data transfer tasks.

(3) A holistic approach is required to model the complex interference behaviors - Contentions across multiple shared resources on accelerators exhibit complex interactions and collectively affect the actual performance degradation experienced by LS applications. Therefore, a holistic approach is required to capture the compound effects of contentions among multiple shared resources and model the interference behaviors.

3. Prophet Methodology

In this section, we present Prophet, which enables precise QoS prediction for co-locations on non-preemptive accelerators by considering contention for PEs, global memory bandwidth, and PCIe bandwidth.

3.1 Design Principals of Prophet

To address the challenges for precise QoS prediction, we design and implement Prophet based on three principals.

- Prophet should be able to predict when the co-located tasks can and cannot run concurrently, capture the queuing effect of non-overlapped execution due to PE contention, model the interference of overlapped execution due to global memory bandwidth contention.
- Prophet should be able to model interference behaviors between data transfer tasks and quantify the performance degradation due to PCIe bandwidth contention.
- Based on the precise QoS prediction, Prophet should be able to identify "safe" co-location pairs to steer application scheduling on accelerators, improving accelerator utilization without violating the QoS of LS applications.

3.2 Prophet Overview

Figure 6 presents the overview of Prophet. Prophet takes real-system solo-run profiles of applications as input to predict the performance interference when these applications are co-located on the accelerator. Prophet is composed of four main components: *schedule predictor*, *task queues*, *task execution model*, and *data transfer model*.

Solo-run profiles - Similar to previous work on QoS prediction for CMP and SMT [33, 50, 53], Prophet profiles LS applications and batch applications separately. During profiling, each application runs alone and its execution profile is collected for interference prediction. The solo-run prevents the execution profile being interfered by other co-runners. We collect solo-run profiles of applications when they accept actual query requests, which have different inputs.

The profile includes task order and task duration of each application, profiled using Nvidia GPUs automatically (more details in Table 1). Prophet predicts the performance degradation when multiple applications are co-located by replaying and interleaving solo-run profiles of co-located applications. In addition, because hardware configurations such as the number of PEs and core frequency vary across

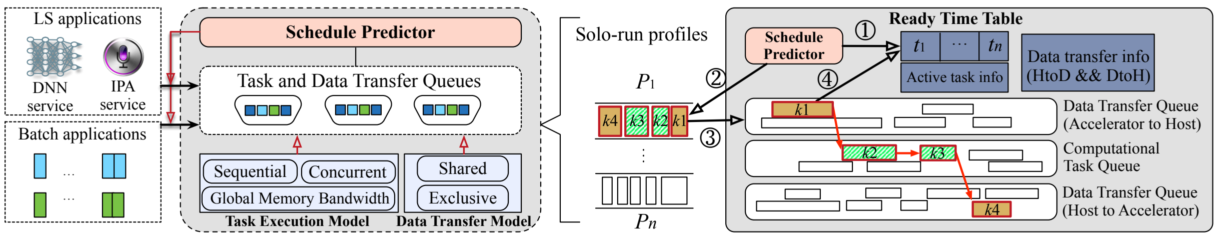


Figure 6: Design of Prophet.

platforms, to achieve the most accurate prediction, Prophet collects solo-run trace on each hardware platform. That being said, techniques that predict cross-platform solo performance [49] can be leveraged by Prophet to further predict cross-platform co-run performance.

Schedule predictor and task queues - As illustrated in Figure 6, Prophet has a schedule predictor that estimates how tasks (both computational and data transfer tasks) from co-located applications are scheduled on real hardware and when the task will complete. When the schedule predictor schedules a task for execution, the task is first pushed into the corresponding task queue. For tasks in the queues, the predictor then uses the task execution model and data transfer model to predict when each task will complete. This task completion information is used to update a ready time table to track the next ready tasks. The process is illustrated on the right side of Figure 6:

(1) Schedule predictor maintains a ready time table to keep track of task completion information and next ready tasks. (2) Schedule predictor fetches the next ready tasks from the solo-run profiles. (3) Schedule predictor pushes tasks to corresponding task queues. (4) For tasks in the queues, task execution model and data transfer model predict the task completion time based on its solo-profile. The completion time is used to update the ready time table.

Once all the tasks in co-located application's solo-run profiles complete, Prophet reports the latency of LS applications in the presence of interference. More details of the predictor and the system are described in Section 6.

Task execution model and data transfer model - In Prophet, two key components that are critical for precise prediction are the task execution model and the data transfer model, which model task execution and data transfer behaviors on real accelerator hardware. To achieve precise prediction, these two models need to accurately reflect real hardware behaviors. In Prophet, we propose a concurrent task execution model to predict the actual task execution pattern in emerging non-preemptive accelerators. Both PE contention and global memory bandwidth contention are considered in this model. In the data transfer model, based on our real system investigation, we model two types of data transfer tasks (bandwidth-sharing and bandwidth-exclusive), which use PCIe bandwidth in different manners. Prophet adopts plug-and-play design so that programmers can develop their own models and plug them into Prophet if their accelerators use different task scheduling or data transfer techniques.

4. Task Execution Model

Task execution interference in accelerators is one key factor that results in the QoS violation of LS applications when co-located. While some accelerators still use the traditional *sequential task execution* model to schedule tasks, more and more emerging commodity accelerators, such as Nvidia Kepler GPUs, can already execute multiple independent tasks simultaneously using *Concurrent Task Execution* [1] to increase the hardware utilization. However, the design details of how these independent tasks are executed concurrently, including how the global memory bandwidth and PEs are shared between tasks, are kept as a black-box. In this section, we first provide a brief background of task execution on accelerators; then present the traditional sequential model and our novel model that precisely captures the concurrent task execution behaviors on commodity accelerators. Our task execution model allows Prophet to predict the completion time of a computational task.

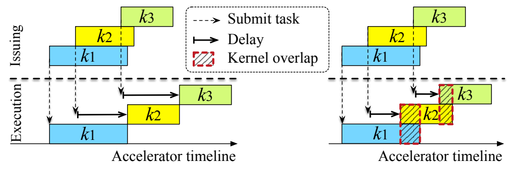
4.1 Background of Task Execution on Accelerators

In popular accelerators such as GPUs, a large amount of parallel threads in a task are scheduled in the granularity of a warp (32 threads on Nvidia GPU [27]). Threads in the same warp execute the same instruction but process different data sets. A PE can only execute warps in the same task.

Non-preemptive accelerators execute tasks in the *first-come-first-serve* manner [9, 24]. When a task k is submitted to the accelerator, if the accelerator is free, the task will be executed immediately. However, if the accelerator is currently executing other tasks, k needs to wait for all the previous tasks complete if sequential task execution is adopted by the accelerator. On the other hand, if the accelerator schedules tasks using concurrent task execution, k may run earlier whenever there are free PEs available. In Figure 7, we show the difference between sequential task execution and concurrent task execution.

4.2 Sequential Task Execution Model

As shown in Figure 7(a), the end-to-end latency of a computational task consists of two parts: the *queuing time* and the actual *execution time* for sequential task execution [54].



(a) Sequential execution (b) Concurrent execution

Figure 7: Comparison between sequential task execution and concurrent task execution on non-preemptive accelerator.

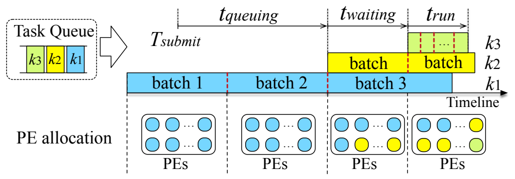


Figure 8: Modeling concurrent task execution.

When a task k of duration t_{run} is submitted to the accelerator at time T, suppose there are still m tasks queued for the accelerator. Let t_1, ..., t_m represent the duration of the m tasks and t_{remain} represent the remaining time of the task currently running on the accelerator. Because tasks are executed sequentially, the completion time of k (denoted by T^k_{comp}) can be calculated using Equation 1.

$$T^k_{comp} = T_{submit} + t_{run} + \sum_{i=1}^{m} t_i + t_{remain} \tag{1}$$

Note that, the actual execution time of a computational task when it is co-located with other applications equals to its execution time while it runs alone. This is because task k can use all the PEs it needs once it is executing, no matter whether it is co-located with other tasks. The performance degradation of k comes from the queuing time for the accelerator.

4.3 Concurrent Task Execution Model

In addition to sequential task execution, tasks can also be executed concurrently on accelerators with such support [1] (Figure 7(b)). However, it is much more challenging to model concurrent task execution because the execution details, such as the kernel overlapping time and the PE assignment policy are kept as an industry secret and remains as a blackbox. In the concurrent task execution model, we consider both contention on PEs and contention on global memory bandwidth.

We made three observations during the investigation regarding the behaviors of concurrent tasks:

- **Observation 1 -** A task will use as many PEs (e.g., streaming multiprocessors in GPUs) as possible if it has enough warps.
- **Observation 2 -** Only when a task cannot fully utilize all the PEs, a queued task can start to use the free PEs.

Table 1: Parameters used in the concurrent execution model

Type	Parameter Name	Symbol
Task Configuration	Duration of the task	t
	Theoretical occupancy	*Occu* [37]
	Number of warps	N_w
	Global memory bandwidth req. per PE	bw
Hardware Configuration	Number of PEs	N_p
	Maximum number of warps per PE	N_{mw}
	Effective global memory bandwidth	GB

- **Observation 3 -** If there are PEs available, the queued tasks use these PEs in *first-come-first-server* manner.

Based on our real system investigation, we identify a set of hardware and task information that our concurrent task execution model uses to predict the completion time of each task. The task information can be automatically collected using a profiler such as *nvprof* [2] for Nvidia GPUs. Table 1 describes this task information used in our concurrent task execution model as parameters. In the table, the theoretical occupancy [37] of a task is the ratio of active warps to the maximum number of warps supported on a PE (streaming multiprocessor). The theoretical occupancy can be impacted by many factors, such as the amount of registers used by each thread, the amount of shared cache used by each warp, etc. Based on the information in the solo-run profiles, The theoretical occupancy of a task can be calculated as described in Nvidia's occupancy calculator [37]. In addition, to obtain the parameter "global memory bandwidth requirement per PE" bw for each task in Table 1, we also profile each application alone with a small input data with which the application can only utilize a small number of PEs. The input is small enough so that the PEs do not saturate the global memory bandwidth. In this case, the parameter of a task can be calculated as the overall global memory bandwidth usage divided by the number of PEs the task used. We do not extract this parameter from actual profiles is because if the input data is large (the case of some actual queries), most PEs are active and they may contend for the global memory bandwidth, which results in the under-estimation of this parameter.

Based on aforementioned observations, Figure 8 illustrates the lifetime of several concurrent tasks. As shown here, task k_3 on a non-preemptive accelerator consists of three phases: waiting to arrive at the head of the task queue on the accelerator ($t_{queuing}$), waiting for free PEs at the head of task queue ($t_{waiting}$), and running on the PEs (t_{run}). The completion time of task k can be calculated by $T_{submit} + t_{queuing} + t_{waiting} + t_{run}$. In this equation, T_{submit} and $t_{queuing}$ can be directly collected during solo run of the task, while $t_{waiting}$ and t_{run} need to be predicted.

4.3.1 Calculating Task Start Time:

As presented in observation 2, when a task arrives the head of the queue, it can start to run only when there are free PEs. To model this behavior, for each active computational task, we calculate how many PEs it will use at different time when it is scheduled to run. Suppose the specification of task k is summarized in Table 1. Each PE can execute $N_{mw} \times Occu$ warps of k in a batch, and the optimal duration of each batch from task k when it runs alone and there is no global memory bandwidth contention, can be calculated as Equation 2.

$$Solo_k = \frac{t}{\lceil N_w/(N_{mw} \times Occu \times N_p) \rceil} \times min\{1, \frac{GB}{bw \times N_p}\} \quad (2)$$

In this equation, we assume different batches of warps in the same task take the same time to run on the accelerator because they execute the same instructions. Note that, for a task, its occupancy is roughly constant across different inputs.

During k's execution, the concurrent task execution model checks how many PEs are available for a batch, and calculates the number of warps executed in that batch. If there are N_{free} PEs available for a batch of warps in k, then the accelerator can complete $N_{free} \times N_{mw} \times Occu$ warps in that batch. In addition, it is also possible that the warps of k cannot fully utilize the available N_{free} PEs. In that case, k only occupies the needed PEs and leaves the rest for later tasks. For example, in Figure 8, k_3 can only start to run when there are free PEs left by k_1 and k_2 (e.g., when k_2 starts its second batch of warps).

By keeping track of when PEs are available, N_{free}, and the number of unprocessed warps in k, the start time of task k and the number of batches in k can be determined.

4.3.2 Calculating Task Complete Time:

When task k starts to run, we predict its duration and update the duration of all the active tasks by modeling global memory bandwidth contention. Global memory requests are served in First-Ready First-Come-First-Serve manner or its variations [51]. Therefore, concurrent tasks are slowed down by the same times when they contend for global memory bandwidth. Let $Solo_k$ represent the duration of a warp batch from task k when it runs alone as calculated in Equation 2. Suppose there are nb batches in task k as calculated in previous section. We iterate through every batch, and predict its actual duration.

For the j-th ($1 \leq j \leq nb$) batch in task k, suppose k uses pe PEs in this batch, and n other tasks k_1, ..., k_n are still active and run on other PEs. For these active tasks, we further use pe_1, ... pe_n to represent the number of PEs they are using, and bw_1, ..., bw_n represent their global memory bandwidth requirement per PE respectively.

Equation 3 predicts the actual duration of the j-th batch of k. In this equation, $bw \times pe$ is the global memory bandwidth required by task k, and $\sum_{i=1}^{n}(bw_i \times pe_i)$ is the global memory bandwidth required by the other n active tasks. If $bw \times pe + \sum_{i=1}^{n}(bw_i \times pe_i)$ is larger than the effective hardware global memory bandwidth GB, the execution of the j-th batch is slowed down proportionally. Otherwise, the duration of the batch at co-location equals to its solo-run duration $Solo_k$.

$$T_j = Solo_k \times max\{\frac{bw \times pe + \sum_{i=1}^{n}(bw_i \times pe_i)}{GB}, 1\} \quad (3)$$

Finally, the completion time for task k, denoted by T_{comp}^{k}, can be calculated in Equation 4. In the equation, T_{submit} and $T_{queuing}$ are collected directly through profiling and $t_{waiting}$ is calculated by keeping track of when PEs are available.

$$T_{comp}^{k} = T_{submit} + t_{queuing} + t_{waiting} + \sum_{i=1}^{nb} T_j \quad (4)$$

It is worth noting that, executing task k may slow down other active tasks, and later tasks may also slow down k due to global memory bandwidth contention. Deduced from aforementioned observation 1 and 2, when k starts to run, all the previous active tasks are running in the their last batches, otherwise the warps of these tasks would take all the PEs and k cannot start. Therefore, when iterating through the batches of k to calculate its complete time, we update the complete time of all the active tasks as well. For task k_i ($1 \leq i \leq n$) that is active duration the j-th batch of task k, let t_{ol} represent the overlap time of its last batch with the j-th batch of k. t_{ol} can be calculated by subtracting the start time of the batch from $T_{comp}^{k_i}$. Equation 5 calculates the new complete time of task k_i (denoted by $T_{comp}^{k_i}$).

$$T_{comp}^{k_i} += t_{ol} \times (\frac{max\{\frac{bw \times pe + \sum_{i=1}^{n}(bw_i \times pe_i)}{GB}, 1\}}{max\{\frac{\sum_{i=1}^{n}(bw_i \times pe_i)}{GB}, 1\}} - 1) \quad (5)$$

5. Data Transfer Model

In addition to contention for PEs and global memory bandwidth, contention for PCIe bandwidth, which significantly impacts the data transfer speed, can also lead to significant performance degradation. In this section, we propose models to predict the duration of a data transfer task when it contends PCIe bandwidth with other data transfer tasks.

5.1 Background of Data Transfer Tasks

Our real system study shows that there are two types of data transfer between the host and the accelerator, *bandwidth sharing data transfer task (BS task)* and *bandwidth exclusive data transfer task (BE task)*, to transfer data through the PCIe bus [10]. A BS task can run concurrently with other BS tasks and they share the PCIe bandwidth evenly. On the other hand, a BE task consumes all the PCIe bandwidth, and has a higher priority than BS tasks. For example, in CUDA for GPU, memory copy tasks from/to pageable memory are BS tasks and memory copy tasks from/to pinned memory are BE tasks.

It is challenging to predict the duration of a data transfer task d directly when it is launched, because its duration can be affected by various factors including the number of active transfer tasks, the data transfer overlapping pattern, the type of each data transfer task, etc. Making prediction even more

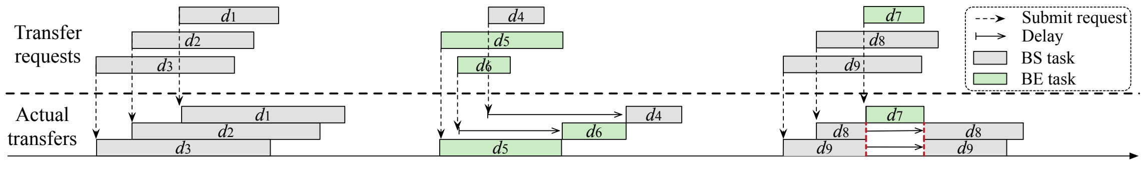


Figure 9: Scenarios of PCIe bandwidth contention when BS tasks and BE tasks share the PCIe bus.

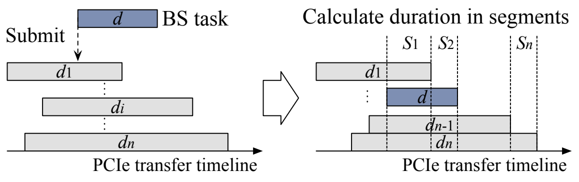


Figure 10: Calculate the duration of BS data transfer tasks when a new BS data transfer task is submitted.

challenging, the data transfer tasks submitted after d will also impact d's duration. To solve this problem, our proposed model updates the duration of all active data transfer tasks whenever a new data transfer task is submitted.

5.2 Modeling Data Transfer over PCIe

Figure 9 illustrates three scenarios of PCIe bandwidth contention that can happen when BS task and BE task co-exist. We have empirically verified these scenarios on our GPU platform using the profiler, which is also consistent with prior work's observation [10]. In Figure 9(a), all the active data transfer tasks are BS tasks, sharing the bandwidth evenly. In Figure 9(b), the newly submitted data transfer task d_4 is delayed until all previous BE data transfer tasks complete. In Figure 9(c), the newly submitted task d_7 is BE and has a higher priority. In this case, it preempts the existing BS data transfer tasks.

5.2.1 Scenario 1: Share PCIe Bandwidth

Figure 10 shows how PCIe bandwidth is shared among the concurrent data transfer tasks. Suppose when a new BS data transfer task d is submitted, there are still n BS data transfer tasks are active. If the PCIe bandwidth is high enough to support all the tasks to transfer data in full speed, their execution time is the same as their solo-run execution time. Otherwise, they need longer time to transfer data due to smaller bandwidth. In this case, let d_1, ..., d_n represent the n tasks in the ascending order of their expected complete time. When d is submitted, it is inserted into the data transfer tasks according to its original expected complete time ($T_{submit} + t_{run}$), where T_{submit} is the time when d is submitted and t_{run} is d's duration when it runs alone. As shown in Figure 10, because d shares the PCIe bandwidth with different number of data transfer tasks in different stages, the model needs to calculate the duration of each active data transfer task by segments. To this end, for all the active data transfer tasks, the transferring is divided into n segments based on the expected complete time of every active data transfer task, and the number of tasks that share the bandwidth in segment S_i (denoted by N_i) can be calculated by Equation 6.

$$N_i = \begin{cases} n-i+2 & \text{, if } S_i \text{ covers part of } d \\ n-i+1 & \text{, if } S_i \text{ does not cover any part of } d \end{cases} \tag{6}$$

Let OD_i represent the time duration of segment S_i before d is submitted. Note that, before d is submitted, the number of tasks sharing the bandwidth in segment S_i (denoted by ON_i) is $n-i+1$.

In our model, all the BS data transfer tasks have the same priority and they share the PCIe bandwidth fairly. Therefore, the new duration of segment S_i after d is submitted, denoted by D_i, can be calculated by Equation 7.

$$D_i = OD_i \times \frac{N_i}{ON_i} \tag{7}$$

Once the duration of each segment is known, the complete time of active task d_i can be calculated by Equation 8.

$$T_{comp}^{d_i} = T_{submit} + \sum_{j=1}^{i} D_j \tag{8}$$

In our model, once a new BS data transfer task is submitted, the complete time of all the active data transfer tasks have to be updated because the newly submitted task affect the PCIe bandwidth that each task can use.

5.2.2 Scenario 2: Queued by BE Tasks

As illustrated in Figure 9(b), if a data transfer task (no matter bandwidth-sharing or bandwidth-exclusive) finds that a BE data transfer task is transferring data, it cannot start to transfer data until all the previous BE data transfer tasks complete. For example, in Figure 9(b), d_6 has to wait for d_5's completion before it can transfer data; and d_4 has to wait for both d_5 and d_6's completion before it can transfer data.

In this scenario, the actual time used by each task to transfer data is not changed. The performance degradation of data transferring is due to the queuing delay in this case.

Suppose there are n BE data transfer tasks are queued for PCIe bus usage when a new data transfer task d of duration t is submitted at T_{submit}. Let t_1, ..., t_n represent the duration of the queued data transfer tasks, and let t_{remain} represent the remaining time needed for the currently running data transfer task. Note that, in this scenario, the currently running data transfer task can only be a BE data transfer task, because the queued BE data transfer tasks can preempt the PCIe bandwidth otherwise (as shown in Figure 9(c)). Equation 9 calculates the expected start time of d (denoted by T^d_{start}).

$$T^d_{start} = T_{submit} + t_{remain} + \sum_{i=1}^{n} t_i \tag{9}$$

Once d starts to run at T^d_{start}, if d is a BE data transfer task, then its complete time can be calculated as $T^d_{start} + t$. On the other hand, if d is a BS data transfer task, its end time can be calculated as described in previous section, because it could share PCIe bandwidth with other BS data transfer tasks.

5.2.3 Scenario 3: Preempt PCIe Bandwidth

As shown in Figure 9(c), when a BE data transfer task is submitted at T_{submit}, it is also possible that the active data transfer tasks are BS data transfer tasks. Because BE data transfer task has higher priority, it will pause all the BS data transfer tasks and start to transfer data immediately. For example, in Figure 9(c), d_7 pauses the on-going tasks, d_8 and d_9, (or limit d_8 and d_9 to use very small bandwidth) and starts to transfer its own data in full speed. d_8 and d_9 can resume the data transfer once d_7 completes.

In this case, the complete time of the newly submitted task can be calculated as $T_{submit} + t$, where t is the data transfer time when the task runs alone. Meanwhile, the completion time of every on-going BS data transfer task is increased by t as well.

6. Putting It All Together

Prophet presents a holistic approach to predict and combine task execution interference and data transfer interference due to the PE contention, global memory bandwidth contention and PCIe bandwidth contention.

The design of Prophet is already presented on the right side of Figure 6. Prophet simulates how real-system accelerators process concurrent tasks submitted by multiple applications. On a real-system accelerator, tasks in the same application are submitted and executed in a FIFO order. On the other hand, tasks in different applications can run concurrently on the accelerator if there are enough PEs, because there is no dependency between these tasks.

To schedule tasks in the same way as the real accelerator, the schedule predictor schedules tasks based on a *ready time table* that contains the ready time of the first un-executed task in each of the co-located applications. The task that has smaller ready time is submitted earlier to the predictor. When the prediction starts, every application is given an initial start time and stored in the ready time table. As illustrated in Figure 6, the schedule predictor first checks the ready time table to decide where to obtain the next task. If the schedule predictor decides to obtain the next task from application P, then the first un-executed task k in P is pushed into the corresponding queue according to its task type (computational task, data transfer task from host to accelerator or data transfer task from accelerator to host). After that, Prophet uses either task execution model or data transfer model to calculate when k completes. The complete time of k is used to update the ready time of next un-executed task from application P stored in the ready time table.

In addition to the ready time table, Prophet maintains execution information for all the active tasks. For computational tasks, the information includes when the task starts, the period that the task cannot fully utilize all the PEs, the number of PEs used in that period, the global memory bandwidth requirement per PE of the task. For active data transfer tasks, the transfer start time and the expected complete time are also tracked. This information is used by the task execution and data transfer models for predicting the complete time of later tasks. Once all the tasks complete, Prophet can report the latency of LS applications in the presence of interference.

Table 2: Hardware, software, and benchmarks

	Specifications
Hardware	CPU: Intel Xeon E5-2620 @ 2.10GHz Accelerator: Nvidia GPU Tesla K40
Software	OS: Ubuntu 14.04 x86_64 with kernel 3.16.0-41 Accelerator driver: CUDA 346.46, SDK 6.5, CUDA MPS
Benchmark Suite	**Workloads**
Sirius Suite [20]	asr, gmm, stemmer
Tonic Suite [19]	dig, imc, ner, pos
Rodinia [12]	heartwall (*hw*), lavaMD (*md*), kmeans (kms), myocyte (*mc*), backprop (*bp*), bfs, pathfinder (*pf*), hotspot (*hs*), nn, bfs-ex, pathfinder-ex (*pf-ex*)

7. Evaluation

In this section, we evaluate the accuracy of Prophet for predicting the performance degradation due to task execution interference and data transfer interference. In addition, we show the effectiveness of Prophet to guide application co-locations, achieving high accelerator utilization without violating the QoS of LS applications.

7.1 Experimental Setup

We evaluate Prophet on Nvidia GPU K40. Hardware specification, software specification, and benchmarks are summarized in Table 2. MPS [1] is used to enable concurrent task execution on GPU. We use workloads from *Tonic Suite* in DNN service DjiNN [19] and *Sirius Suite* in IPA application Sirius [20] as LS applications, and both compute-intensive and PCIe-intensive workloads from *Rodinia* [12] as batch applications. We implement two data transfer modes in the benchmarks. For example, *bfs-ex* and *pf-ex* use BE data transfer tasks to transfer data between CPU and GPU, while *bfs* and *pf* use BS data transfer tasks. The ratio of execution time for computation and PCIe data transfer of each workload is shown in Figure 11. The LS applications *gmm*, *asr*, *dig* and *imc* are more compute-intensive, while *pos*, *ner* and *stemmer* are more PCIe-intensive.

In our evaluation, the performance degradation *Deg* of LS applications due to co-location and the prediction error *Err* for the degradation are defined in Equation 10. In the equation, L_{solo} and L_{colo} are the latency of LS application

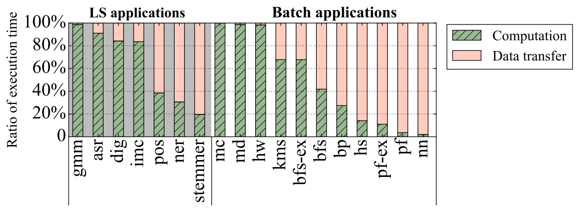


Figure 11: Percentage of execution time spent on computational tasks and data transfer tasks in each benchmark.

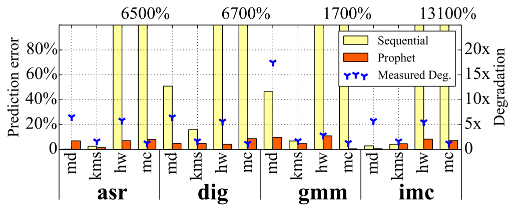


Figure 12: Performance prediction accuracy of Prophet for LS applications in compute-intensive co-location pairs. The average prediction error of Prophet is 5.9%.

when it runs alone, and when it is co-located with batch applications respectively.

$$Deg = \frac{|L_{solo} - L_{colo}|}{L_{solo}}, \quad Err = \frac{|Deg_{pred} - Deg_{measured}|}{Deg_{measured}} \tag{10}$$

7.2 Performance Interference Prediction

We report prediction accuracy for all the $7 \times 11 = 77$ combinations of the 7 LS applications and the 11 batch applications in this section. In the following figures, the x-axis shows the co-location pairs. For example, *md* under *asr* means LS application *asr* is co-located with batch application *md*.

7.2.1 Prediction for Compute-Intensive Co-locations

In this experiment, we evaluate the accuracy of Prophet when predicting performance degradation due to task execution interference caused by PE and global memory bandwidth contentions. We co-locate compute-intensive LS applications with compute-intensive batch applications. Figure 12 presents the resulting performance degradation for LS applications measured on the real system. In addition, we present the degradation predicted using Prophet as well as the sequential task execution model (described in Section 4.2, and used in I-Torque [41]).

The average performance degradation of LS applications due to PE and global memory bandwidth contentions is 4.0x and with the worst case being 17.5x (co-locating *gmm* and *md*). When using Prophet to predict the performance interference, the prediction error ranges from 0.6% to 10.9% with an average prediction error of 5.9%. On the contrary, the sequential task execution model's prediction error is up to 13100%. The main cause of the poor prediction accuracy of sequential model is the lack of consideration of the concurrent task execution on real emerging non-preemptive accelerators. In a sequential model, a kernel cannot run until previous kernels complete, resulting in severe overestimation of latency-sensitive queries' latency especially if kernels of batch applications are long running but have low occupancy. Prophet exhibits much higher prediction accuracy due to its capability to capture the contention behaviors including the queuing effect, concurrent task execution, and global memory bandwidth sharing.

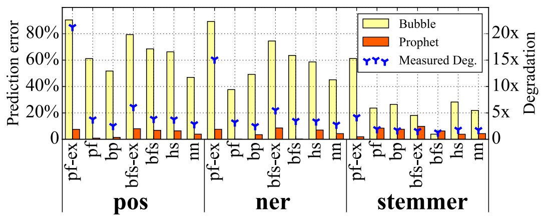


Figure 13: Performance prediction accuracy for LS applications in PCIe-intensive co-location pairs. The average prediction error of Prophet is 5.1%.

7.2.2 Prediction for PCIe-Intensive Co-locations

Similarly, we evaluate the accuracy of Prophet for predicting the performance degradation due to data transfer interference caused by PCIe bandwidth contention. To achieve this, we co-locate applications that exhibit intensive PCIe data transfers in this experiment. Because Bubble-Up [33] and Bubble-Flux [50] can precisely predict main memory bandwidth contention, as a baseline, we design and evaluate bubble-based prediction for PCIe bandwidth contention (denoted by "Bubble" in Figure 13). According to the design of Bubble-Up and Bubble-Flux, we implement PCIe bubble to be a micro-benchmark that keeps transfer data via PCIe bus and does not do any computation. The pressure on PCIe bandwidth is changed by adjusting the number of instances of the micro-benchmark.

As shown in Figure 13, PCIe-intensive LS applications suffer from up to 21x performance degradation (co-locating *pos* and *pf-ex*) and 4.5x on average. For all the co-located applications, Prophet performs well in predicting the performance degradation, with the prediction error ranging from 0.01% to 9.8% and on average 5.1%. In particular, the average prediction error for Sirius and Tonic applications is 6% and 4.4% respectively. On the contrary, bubble-based approach performs much worse than Prophet, with the prediction error ranging from 3.9% to 90.5% and on average 50.8%. This is because memory bandwidth is shared in *first-ready first-come-first-serve* manner on real systems [35], but data transfers through PCIe bus have priorities. Bubble-based approach designed for memory bandwidth contention is not able to capture this complex data transfer behavior through PCIe bus.

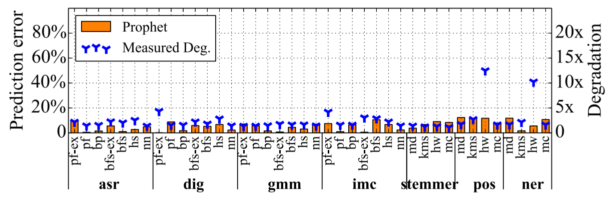


Figure 14: Performance prediction accuracy for LS applications in low contention co-location pairs. The average prediction error of Prophet is 5.5%.

During the evaluation, we observe that although the BE data transfer tasks provide higher data transfer speed when running alone, the exclusive transfer may severely slow down other data transfer tasks, and result in severe performance degradation of LS applications. This is because BE data transfer tasks do not allow concurrent data transfer for co-located applications, while BS data transfer task does not have such restriction. Therefore, a long PCIe data transfer task from batch applications could significantly delay the data transfer task from LS application and cause severe QoS violation. This observation provides useful insight to bound the QoS of LS application under co-location. It is beneficial for LS applications to use bandwidth-exclusive PCIe data transfer for higher bandwidth. However, batch applications should always use bandwidth-sharing transfer since it reduces the likelihood to significantly delay the data transfer of LS applications under co-location.

7.2.3 Prediction for Low Contention Co-locations

In previous sections, we present results when both LS and batch applications contend for the same resource. To better understand the opportunity for utilization improvement without severely degrading the performance, we co-locate applications with different contention points, in this case, compute-intensive LS applications with PCIe-intensive batch applications and vice versa. As shown in Figure 14, the average performance degradation for these co-locations is 2.39x, which is smaller than degradation when applications suffer from serious interference due to the contention on the same shared resources (4.0x for task execution interference, 4.5x for data transfer interference). For most co-locations that are stressing on different resources, the performance degradation is smaller with an average of 1.8x degradation. For these co-locations, Prophet still maintains the similarly high prediction accuracy with the average prediction error of 5.5%. In particular, for Sirius and Tonic workloads, the average prediction error is 4.6% and 6.3% respectively.

As confirmed by this experiment, there is optimization space to steer the application co-location. Co-locating applications that are stressing on different resources alleviates the performance interference. Therefore, precise prediction of performance interference is critical to guarantee that application co-locations satisfy the QoS target while improving utilization.

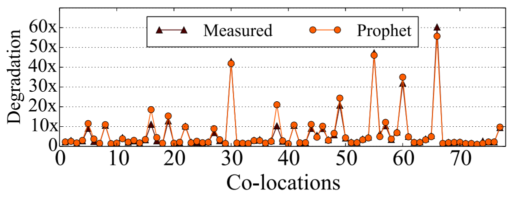


Figure 15: The 95%-ile latency degradation for LS applications in all the 77 co-locations.

7.2.4 Prediction Accuracy for Tail Latency

Previous sections focus on predicting the average latency degradation, in this section, we evaluate the accuracy of Prophet for predicting the 95%-ile latency. Figure 15 shows the 95%-ile latency degradation of LS applications in all the 77 co-locations. The figure demonstrates that for most of the co-locations, Prophet is able to predict the tail latency accurately. The average prediction error of Prophet for 95%-ile latency is 13.5%. Note that this is slightly higher than the average latency prediction. This is because the tail latency is much more affected by the interference than the average latency. We measured up to 60x tail latency degradation due to interference. Thus a 13.5% prediction error can still accurately identify reasonable co-location scenarios. In Section 7.4, we show that the percentage of QoS violation in all the co-locations identified by Prophet is smaller than 7.8%.

7.3 Improving Utilization and Maintaining QoS

With Prophet's precise interference prediction for accelerators, we can enable "safe" co-locations in order to improve utilization without violating the QoS requirement. In this experiment, for each LS application, we set its QoS target to be under 2x of its solo latency and below 100 milliseconds [25, 40]. We use Prophet to predict how many instances of each batch application can be co-located with each LS application without violating the QoS. Our evaluation baseline disallows co-locations on accelerator, which is the state-of-the-art approach to bound QoS in modern WSCs without a precise QoS prediction mechanism. To enable concurrent task execution, we use MPS that supports up to 16 applications submitting tasks to GPU simultaneously. We could co-locate 0 to 15 instances of batch applications on each GPU. From WSC perspective, compared to the actual hardware utilization, the achieved throughput for batch applications when they are safely co-located with latency-sensitive service is the ultimate performance optimization metric. Therefore, when Prophet predicts that k batch applications can be co-located with an LS application on the same accelerator without violating QoS target, the utilization improvement is calculated as $k/15$.

Improving Utilization - Figure 16 shows the utilization improvement when co-locating batch applications with each LS application guide by Prophet, and the actual measured oracle utilization improvement. There is a spectrum

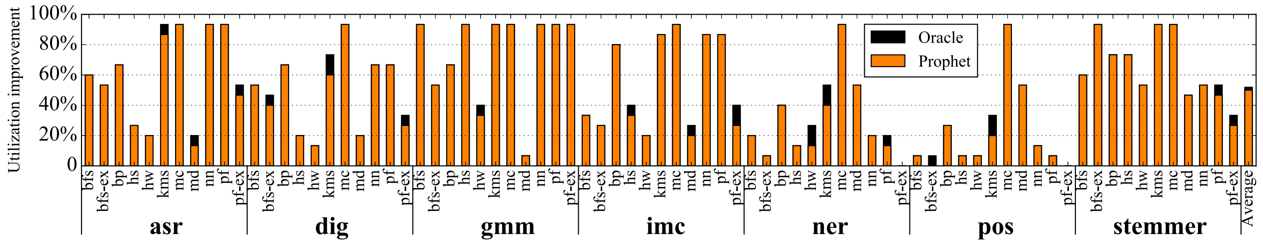


Figure 16: Utilization improvement achieved by Prophet. It improves the accelerator utilization by 49.9% on average.

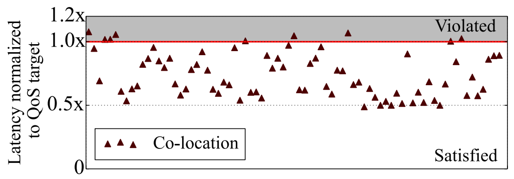


Figure 17: QoS of LS applications in all the 77 co-locations based on Prophet's prediction.

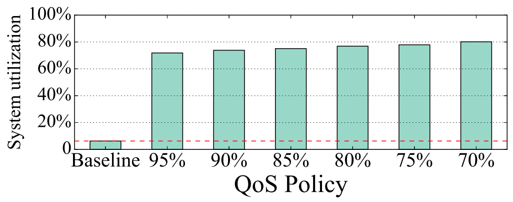


Figure 18: Accelerator utilization when using Prophet to select "safe" co-locations under each QoS policy.

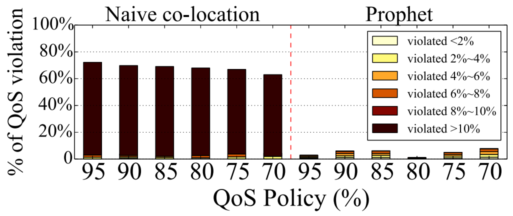


Figure 19: Percentage of QoS violation in all scheduled co-locations under each QoS policy.

of achieved utilization improvement. The improvement is high especially when compute-intensive LS application is co-located with PCIe-intensive batch applications (e.g., co-locating *asr* with *pf*) and vice versa. Prophet achieves up to 93.3% of the maximum utilization, because these applications use different shared resources and the contention is low. On the contrary, when the co-located applications contend for the same shared resources (e.g., *pos* with *bfs-ex*, and *asr* with *hw*), only a small number of batch applications can be co-located with LS applications due to the QoS requirement. The utilization improvement drops to 0 in the worst case, which means the co-location is not allowed. Benefiting from the precise interference prediction, Prophet increases the accelerator utilization by 49.9% on average, while the measured oracle utilization improvement is 51.8%.

Meeting QoS Target - To demonstrate the capability of Prophet in steering the application co-location while maintaining the QoS target, we co-locate applications on real GPU systems based on the co-location decisions predicted by Prophet. For each co-location, Figure 17 shows the average performance of the LS application normalized against its QoS target. We can find that only 7 out of the 77 co-location decisions (less than 9.1%) result in slight QoS violation, while the QoS violation ranges from 0.4% to 7.8%. For other co-locations, the QoS of LS applications is maintained with increased utilization.

7.4 Scale-out Study

Instead of improving the utilization of a single accelerator as in previous sections, this section shows how to choose co-locations to maximize the utilization of accelerators in a datacenter. In this experiment, we model a cluster composed of 700 Nvidia K40 GPUs and each 100 of them are running one of the seven LS applications in Sirius suite and Tonic suite. The batch workloads are composed of 11,000 application instances evenly selected from the 11 batch benchmarks shown in Table 2.

Figure 18 shows the accelerator utilization achieved by Prophet under various QoS policies. In the figure, the baseline does not allow accelerator co-location and the resulting utilization is low. Prophet significantly improves the accelerator utilization by co-locating batch applications with LS applications. For example, under 70% QoS policy (the QoS target's performance is 70% of solo performance), the accelerator utilization is increased to 80.15%.

Figure 19 presents the percentage of co-locations that suffer from QoS violation when all PEs are fully utilized (left) vs. when co-locations are suggested by Prophet (right) under different QoS policies. The percentage of QoS violations is defined as the number of co-locations that suffer from QoS violation divided by the overall number of co-locations. For all QoS policies, the percentage of QoS violation is smaller than 7.8%. Note that for Prophet 95% and 80% QoS polices, 50% and 40% of the violations is under 2% QoS degradation, respectively. Even in the worst cases, the co-locations suggested by Prophet violate the QoS target less than 10%, compared to the naive full co-location case where more than 70% of the co-locations suffer more than 10% degradation.

Table 3: Comparison between Prophet and prior work

	GPU sharing				Interference prediction			Prophet
	TimeGraph [26]	Baymax [13]	GPU-EvR [29]	SMK [48]	I-Torque [41]	Bubble-Up [33, 50, 53]	Quasar [16]	
QoS Prediction						✔	✔	✔
Work on Accelerator	✔	✔	✔	✔	✔			✔
Improved Utilization		✔		✔	✔	✔	✔	✔
Precise PCIe Modeling								✔
Global Memory Modeling								✔
Concurrent Kernel Exec.		✔	✔					✔

8. Related Work

Precise prediction for performance interference on CPUs has been identified as a key challenge in datacenters and is widely studied in prior work [28, 33, 46, 50, 53]. However, none of the prior work is applicable for non-preemptive accelerators such as GPUs because the key factors that cause performance interference among co-located applications are fundamentally different on GPUs from on CPUs. More specifically, the prior work focus on the resource contention on CPUs including contention for shared caches, memory bandwidth and function units for SMT cores. For instance, Bubble-Up [33] and Bubble-Flux [50] quantify an application's sensitivity to contention by co-locating the application with a set of cache-intensive micro-benchmarks with varying working set size. The measured sensitivity curve is then used to predict the potential performance degradation when the application is co-located with a co-runner. The only shared resource this work addresses is the shared cache and memory bandwidth. SMiTe [53] focuses on micro-architectural shared resources on SMT servers including memory ports and functional units, and relies on regression to achieve precise performance prediction.

These techniques would fail on accelerators because

- they neglect key performance factors on non-preemptive accelerators: queuing delay for PEs, global memory bandwidth contention and PCIe bandwidth contention.
- they neglect the complex interaction of interference on different shared resources.
- PCIe bandwidth contention is significantly different from well-studied main memory bandwidth contention.

Another related research direction aims at increasing accelerator utilization through hardware sharing [3, 13, 38, 39, 45, 48]. Prior work in this direction mainly focuses on designing novel kernel/warp schedulers, and they are not able to predict performance interference caused by sharing accelerators. Several techniques [4, 18, 29] including TimeGraph [26] and GPUSync [17], are proposed to improve the performance of realtime tasks (e.g., frame rate for video processing) using fine-grained kernel scheduling when they are co-located with low priority applications. They make sure that sufficient resources are assigned to realtime tasks in the long term but cannot guarantee QoS [29]. Baymax [13] does not perform QoS prediction and cannot be used in selecting co-runners. As such, this technique can only be applied after co-schedules are decided. The only work that predicts interference on GPUs, I-Torque [41], assumes sequential kernel execution and considers neither global memory bandwidth contention nor PCIe bandwidth contention. We implemented and evaluated the sequential model, and experimental results show that its prediction error is much worse than Prophet for emerging commodity GPUs.

Beyond interference prediction and kernel scheduling, a large amount of techniques are proposed to improve application performance on accelerators in general [5, 6, 11, 30, 42, 44, 47], and to model GPU performance/energy [21, 22, 52]. However, most of the existing GPU modeling work focuses on modeling the performance of a single program on a specific GPU and is not able to be used to identify the "safe" co-locations. Prophet instead focuses on providing satisfactory QoS for LS applications while improving accelerator utilization.

As a summary, Table 3 compares Prophet with prior GPU sharing techniques and interference prediction techniques.

9. Conclusion

We present Prophet, a methodology to precisely predict the performance interference for application co-location on non-preemptive accelerators. With the precise interference prediction from Prophet, "safe" co-location pairs are identified to improve the utilization of accelerators without violating the QoS of latency-sensitive applications. Through evaluating Prophet with emerging latency-sensitive applications, we demonstrate the accuracy of Prophet in predicting the performance interference due to resource contention on processing elements, global memory bandwidth and PCIe bandwidth. The average prediction error of Prophet is 5.47% across pairwise co-locations. Based on the precise QoS prediction, Prophet improves the non-preemptive accelerator utilization by 49.9% on average through co-locations while maintaining the QoS of latency-sensitive applications.

Acknowledgments

We thank our anonymous reviewers for their feedback and suggestions. This work was partially sponsored by the National Basic Research 973 Program of China under grant 2015CB352403, the National Natural Science Foundation of China (NSFC) (61602301, 61502019), and by National Science Foundation (NSF) under grants CNS-CSR-1419243, CCF-SHF-1302682, and SHF-1553485.

References

[1] Nvidia Multi-Process Service. `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[2] Profiler User's Guide. `http://docs.nvidia.com/cuda/profiler-users-guide`.

[3] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte. The Case for GPGPU Spatial Multitasking. In *the 18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2012.

[4] P. Aguilera, K. Morrow, and N. S. Kim. QoS-aware Dynamic Resource Allocation for Spatial-Multitasking GPUs. In *the 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 726–731. IEEE, 2014.

[5] J. Anantpur and R. Govindarajan. PRO: Progress Aware GPU Warp Scheduling Algorithm. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 979–988. IEEE, 2015.

[6] R. Ausavarungnirun, S. Ghose, O. Kayıran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 25–38. ACM, 2015.

[7] L. A. Barroso and U. Hölzle. The Case for Energy-proportional Computing. *Computer*, (12):33–37, 2007.

[8] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *Micro*, 23(2):22–28, 2003.

[9] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann. Cooperative Multitasking for Heterogeneous Accelerators in the Linux Completely Fair Scheduler. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 223–226. IEEE, 2011.

[10] R. Bittner, E. Ruf, and A. Forin. Direct GPU/FPGA Communication via PCI Express. *Cluster Computing*, 17(2):339–348, 2014.

[11] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. Hwu. Automatic Execution of Single-GPU Computations across Multiple GPUs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 467–468. ACM, 2014.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE, 2009.

[13] Q. Chen, H. Yang, J. Mars, and L. Tang. Baymax: QoS Awareness and Increased Utilization of Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 681–696, New York, NY, USA, 2016. ACM.

[14] C. Delimitrou and C. Kozyrakis. iBench: Quantifying Interference for Datacenter Applications. In *International Symposium on Workload Characterization (IISWC)*, pages 23–33. IEEE, 2013.

[15] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. *ACM SIGARCH Computer Architecture News*, 41(1):77–88, 2013.

[16] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–144, New York, NY, USA, 2014. ACM.

[17] G. Elliott, B. C. Ward, J. H. Anderson, et al. GPUSync: A Framework for Real-time GPU Management. In *the 34th Real-Time Systems Symposium (RTSS)*, pages 33–44. IEEE, 2013.

[18] G. A. Elliott and J. H. Anderson. Globally Scheduled Real-time Multiprocessor Systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[19] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, R. Dreslinski, T. Mudge, J. Mars, and L. Tang. DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40, New York, NY, USA, 2015. ACM.

[20] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–238, New York, NY, USA, 2015. ACM.

[21] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 152–163, New York, NY, USA, 2009. ACM.

[22] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 280–289, New York, NY, USA, 2010. ACM.

[23] N. Jones. The Learning Machines, 2014.

[24] W. Joo and D. Shin. Resource-Constrained Spatial Multitasking for Embedded GPU. In *International Conference on Consumer Electronics (ICCE)*, pages 339–340. IEEE, 2014.

[25] H. Kasture and D. Sanchez. Ubik: Efficient Cache Sharing with Strict QoS for Latency-critical Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 729–742, New York, NY, USA, 2014. ACM.

[26] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments. In *USENIX Annual Technical Conference (ATC)*, pages 17–30, 2011.

[27] D. Kirk et al. NVIDIA CUDA Software and GPU Parallel Computing Architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[28] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean code: Achieving Near-free Online Code Transformations for Warehouse Scale Computers. In *Proceedings of the 47th An-*

nual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 558–570. IEEE, 2014.

[29] H. Lee, A. Faruque, and M. Abdullah. GPU-EvR: Run-time Event-based Real-time Scheduling Framework on GPGPU Platform. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[30] S.-Y. Lee, A. Arunkumar, and C.-J. Wu. CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 515–527. ACM, 2015.

[31] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proceedings of the 9th European Conference on Computer Systems*, page 4. ACM, 2014.

[32] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 450–462. ACM, 2015.

[33] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259, New York, NY, USA, 2011. ACM.

[34] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, New York, NY, USA, 2009. ACM.

[35] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 208–222. IEEE, 2006.

[36] C. Nvidia. Compute Unified Device Architecture Programming Guide. 2007.

[37] C. NVIDIA. GPU Occupancy Calculator. *CUDA SDK*, 2010.

[38] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 407–418, New York, NY, USA, 2013. ACM.

[39] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 593–606. ACM, 2015.

[40] V. Petrucci, M. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, L. Tang, et al. Octopus-Man: QoS-Driven Task Management for Heterogeneous Multicores in Warehouse-Scale Computers. In *the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–258. IEEE, 2015.

[41] R. Phull, C.-H. Li, K. Rao, H. Cadambi, and S. Chakradhar. Interference-Driven Resource Management for GPU-based Heterogeneous Clusters. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 109–120. ACM, 2012.

[42] B. Pichai, L. Hsu, and A. Bhattacharjee. Address Translation for Throughput-Oriented Accelerators. *Micro, IEEE*, 35(3): 102–113, May 2015.

[43] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *the 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[44] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-Aware Warp Scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 99–110. ACM, 2013.

[45] K. Sajjapongse, X. Wang, and M. Becchi. A Preemption-based Runtime to Efficiently Schedule Multi-process Applications on Heterogeneous Clusters with GPUs. In *Proceedings of the 22nd international symposium on High-performance Parallel and Distributed Computing (HPDC)*, pages 179–190. ACM, 2013.

[46] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 89–100, New York, NY, USA, 2013. ACM.

[47] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu. A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 41–53. ACM, 2015.

[48] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing. In *the 22th International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369. IEEE, 2016.

[49] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. GPGPU Performance and Power Estimation Using Machine Learning. In *the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576. IEEE, 2015.

[50] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 607–618, New York, NY, USA, 2013. ACM.

[51] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 34–44. ACM, 2009.

[52] Y. Zhang and J. D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. In *Proceedings of the 17th*

International Symposium on High Performance Computer Architecture (HPCA), pages 382–393. IEEE, 2011.

[53] Y. Zhang, M. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS Prediction on Real System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the 47th Annual International Symposium on Microarchitecture (MICRO)*, pages 406–418, New York, NY, USA, 2014. ACM.

[54] J. Zhong and B. He. Kernelet: High-throughput GPU Kernel Executions with Dynamic Slicing and Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6): 1522–1532, 2014.