Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems

Haishan Zhu
The University of Texas at Austin
haishanz@utexas.edu

Mattan Erez
The University of Texas at Austin
mattan.erez@utexas.edu

Abstract

Latency-critical applications suffer from both average performance degradation and reduced completion time predictability when collocated with batch tasks. Such variation forces the system to overprovision resources to ensure Quality of Service (QoS) for latency-critical tasks, degrading overall system throughput. We explore the causes of this variation and exploit the opportunities of mitigating variation directly to simultaneously improve both QoS and utilization. We develop, implement, and evaluate Dirigent, a lightweight performance-management runtime system that accurately controls the QoS of latency-critical applications at fine time scales, leveraging existing architecture mechanisms. We evaluate Dirigent on a real machine and show that it is significantly more effective than configurations representative of prior schemes.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: Hardware/software interfaces

Keywords QoS; Colocation; Latency Distribution; Latency Tail; Run-Time Prediction; DVFS; Cache Partitioning

1. Introduction

There is an growing interest in cloud computing as it can significantly reduce the cost of computation and storage. However, while consolidating the workloads of many clients reduces average cost, the datacenters that house these cloud services require tremendous up-front investment; for example, Facebook recently invested $1 billion on its new datacenter [57]. A major challenge datacenters face today is that this large investment is often underutilized because current individual nodes cannot always meet the processing requirements of customers unless they are not fully utilized [2, 11]. Studies on large data centers have reported average utilization to be between 10% and 50% [2, 11, 41, 62].

The main reason for this underutilization is that many user-facing applications have strict latency requirements such that collocating them with other throughput-oriented batch applications is challenging. The contention from background batch jobs increases the average latency, worst-case execution time (WCET), and the fraction of tasks that exceed the requirements of user-facing latency-critical jobs. However, without "backfilling" nodes with batch jobs, it is practically impossible to fully load them.

Although the problem of meeting latency goals while improving system utilization has been addressed in previous work (e.g., [9, 12, 14, 15, 19, 26–30, 32, 36, 39, 42, 43, 45, 49–51, 58, 60–63]), intensive resource contention also leads to large variation in the execution time of latency-critical tasks even when their average and percentile performance matches the *quality of service* (QoS) goals. In this paper we focus on this little-explored variation problem. We show how variation significantly reduces efficiency. We then propose and evaluate techniques to reduce variation and take advantage of the resulting predictable task completion times.

An interesting characteristic of many user-facing workloads is that there is limited utility in completing tasks faster than the critical latency goal [2, 11, 35]. This is true both for very low-latency tasks, such as those from search, content caching, and delivery workloads [13, 42, 62], and for the somewhat longer latency associated with tasks of live video processing, online stream data analysis, and recognition tasks [6, 17, 38]. In this work, we focus on the latter kind of workloads. Specifically, we study workloads that are offloaded to the cloud and that are characterized by being both computationally intensive and latency sensitive. Because very fast execution is not beneficial but latency targets are very strict, performance variation leads to poor system utilization—resources are reserved and allocated such that most tasks can finish on time, but are wasted for those tasks that finish quickly. Importantly, the variation in completion time has both slowly- and rapidly-varying components such that managing resources at coarse time scales is not enough to fully utilize a system.

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.

DOI: http://dx.doi.org/10.1145/2872362.2872394

To address the challenges and opportunities of rapid variation, we study applications with execution-time constraints, which we call *foreground* (FG) applications by measuring their performance when running with different batch-oriented *background* (BG) applications on a real machine. We focus on variation caused by contention in the memory system because it is the most commonly shared resource and because other resource conflicts (e.g., I/O) can be handled at a coarser granularity by task schedulers [12, 13]. We show that FG applications suffer from significant performance variance when running under contention. We design and implement *Dirigent*, a lightweight resource management runtime that directly reduces the variation of FG jobs at very fine timescales to simultaneously meet their latency targets and provide more resources to BG jobs. Dirigent is based on a novel technique for dynamically predicting the task-to-task completion time within the execution of an FG task and adapting frequency and cache partitioning. This fine time scale management is fundamentally different from prior approaches and is enabled by the lightweight runtime and completion-time predictor.

Our work on Dirigent makes the following contributions:

- We expose the problem of performance variation for latency-critical (foreground) tasks when running under contention. Such variation prevents schedulers and resource managers from maximizing utilization.
- We develop a very lightweight technique for accurately predicting the execution time of foreground tasks at very fine time scales and with very high accuracy; we evaluate this predictor on a real system.
- We further develop the Dirigent lightweight runtime controller that exploits the new trade-off space between reducing the execution time variation of foreground tasks and the throughput of background tasks. Dirigent coordinates all contending processes with the knowledge of expected completion times and deadlines. Dirigent is thus able to improve background-task performance by enabling foreground tasks to yield resources when they are expected to finish faster than their required latency.
- We evaluate Dirigent on a real machine using a range of benchmarks. Results show Dirigent significantly reduces the performance variation of foreground applications while minimizing the background-task throughput degradation when compared to allowing the background jobs to freely contend for resources. On average, Dirigent can achieve 85% reduction in the standard deviation of execution time for foreground applications at the cost of just 9% background task performance loss; this translates into 30% better background throughput than schemes that operate at coarse time scales while also achieving higher completion success rates for the foreground latency-critical tasks.

The rest of this paper is organized as follows. Section 2 provides background on applications of interest and the challenges related to performance variation. Section 3 explains how reducing variation in foreground applications benefits system operation, and discusses the underlying architectural and system mechanisms to control resource use. Section 4 describes the design and implementation of Dirigent. Section 5 presents evaluation results. Section 6 reviews prior QoS management techniques related to Dirigent. Finally, Section 7 concludes the paper and discusses future work.

2. Background

In this section we briefly summarize background most pertinent to Dirigent, including a description of our target workloads, and the goals of performance and QoS management techniques in the context of the workloads.

2.1 Target Workloads and Metrics

Based on prior publications, we classify cloud workloads into three categories [2, 6, 8, 17, 35, 38]. The first includes tasks that are not user-facing, for which throughput is the primary concern, and that can be freely scheduled in the background when resources are available. The second class includes short latency-critical user-facing tasks, such as responding to web search requests and content caching. This class of workloads is characterized by short deadlines in the order of tens of milliseconds. The third is an emerging class of workloads that correspond to offloading of work from user devices to the cloud. Examples include online video processing, online stream data analysis, and recognition tasks. Such tasks are user facing and latency critical, yet are also computationally intensive and have relatively long execution times. These tasks can take hundreds of milliseconds or more to finish and therefore can benefit from being offloaded to the cloud [6, 17, 38].

Both the second and third classes are typically user-generated tasks, of which those arising from sophisticated data and sensor processing applications often belong to the third class. Both classes pose challenges for efficient cloud resource usage. In particular, their performance is expressed both in terms of throughput (number of tasks processed per unit time) and in terms of strict latency constraints where only a small fraction of tasks can violate the constraints without severe penalties. Two common and useful measures corresponding to these performance goals are *average execution time* for throughput and *95-percentile execution time* (or other percentile goals) for latency constraints. Note that in evaluating Dirigent in this paper, we use workloads comprised of tasks from the first and the third categories.

2.2 Challenges

The strict performance goals of user-facing tasks can lead to poor system utilization, where nodes do not operate near their peak processing capability. There are two primary reasons for that. First, FG tasks on their own cannot be used

to fully utilize a node because it is often undesirable to collocate FG tasks because of aspects relating to how data is sharded across cloud servers and sizes of application states [2, 11, 42]. Second, "backfilling" BG tasks to better utilize nodes is also not always possible because of the detrimental impact this has on FG performance.

The problem with collocating BG and FG tasks is the performance degradation because the contention between FG and BG tasks for limited processor resources frequently violates FG performance goals. In some collocation scenarios, compromised average latency can be easily identified. The bigger challenge is variations in performance that lead to latency violations. Variations in FG completion times happen when long-running BG tasks vary in their characteristics (phases) or when new tasks are scheduled for collocation, altering the interference and contention impact [30, 37, 62].

Prior work that addresses the challenge of collocation has focused on managing interference at relatively coarse granularity by scheduling only tasks that do not interfere much onto the same node [45, 60], by statically throttling BG tasks such that sufficient resources are given to collocated FG tasks [62], by slowly and dynamically adjusting BG task throttling to respond to diurnal load variation [42], or by changing scheduling policies within the OS or across the cloud server to ensure QoS [12, 13, 30, 37, 51, 52, 63].

In contrast to these prior works, Dirigent focuses on very fine time scale to directly minimize task-to-task variation in FG task completion times, which allows Dirigent to free up significantly more resources that can be used by BG tasks without compromising either the throughput or latency targets of the FG tasks. We explain these benefits as well as the mechanisms Dirigent relies on in the next section.

3. Dirigent Principles

In this section we discuss how focusing on reducing the variation in FG task execution benefits system operation and mechanisms to manage variation.

3.1 Benefits of Minimizing Variation

An important insight that Dirigent leverages is that minimizing variation can simultaneously meet the performance targets of latency-critical tasks and improve system utilization. Large variations in the completion time of FG tasks cause inefficiencies in the shared system because they lead to over-provisioned resources for FG tasks and system underutilization for latency targets to be met. To understand these inefficiencies, consider an FG task that can meet its throughput and latency targets when running alone, but suffers from large variation in execution time when under contention.

Figure 1 shows an example of the execution time probability density function of such a program, where the blue curve represents standalone execution, the red curve shows the behavior under contention, and an ideal curve is shown in green. In standalone execution, FG tasks often complete significantly ahead of the deadline and the throughput achieved (average execution time) is higher than required. As a result hardware is not fully utilized because there is large performance headroom in most FG executions that could be used for other tasks (shown as the highlighted region). However, when a BG task is introduced, too many deadlines are missed despite the average throughput target being maintained.

Figure 1: Example FG completion time probability density functions when run alone, under contention, and in an ideal scenario; the shaded region points to underutilized resources when run alone.

Previous work focused on keeping core occupancy high while meeting latency goals [42, 45, 62]. However, by explicitly addressing task-to-task variation at fine time scales within a single task, the ideal (green curve) can be achieved. In this ideal case, both throughput and latency targets are met precisely and core and frequency resources are maximally utilized. Dirigent achieves this ideal curve through unique fine time scale monitoring and control mechanisms.

Furthermore, high variation reduces resource utilization because of task scheduling policies designed for meeting FG task latency constraints. Consider a common reservation-based scheduling policy that ensures on-time completion of FG tasks by reserving sufficient resources to guarantee a 95% latency target [1]. Figure 2 shows how this scheduler operates on two different sets of tasks: tasks of type A, which have high execution time variance, and type B which have low variance. With high variance, the scheduler reserves too much time for the first task (shown in green) because allocation is forced to expand due to the long tail of execution time probability distribution curve, leading to poor system utilization. This does not happen with the low-variance tasks. Significant prior work on scheduling has shown that scheduling tasks with deadlines can be done more aggressively and with higher system resource utilization when variance is low [56].

Dirigent controls fine-grained scheduling and frequency to maximize utilization while meeting QoS goals. In this way overall utility per unit energy is maximized. This is in contrast to a line of prior work that reduces the impact of variation by rapidly adjusting processor clock frequency [5, 21, 31, 40, 41, 59]. Matching frequency to FG compute needs reduces processor energy consumption, but falls short of maximizing efficiency because the processor itself consumes just $25\% - 35\%$ of total system power [16, 41, 46].

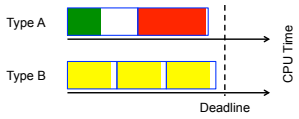


Figure 2: Reservation-based scheduler efficiency with two different task types: type A with high execution time variance and type B with low variance.

3.2 Contention Control Mechanisms

As we explain in Section 4, Dirigent relies on existing mechanisms to manage interference and provide good QoS for the FG jobs. Dirigent uses per-core DVFS and cache partitioning, which we summarize below. We also discuss additional hardware mechanisms that have recently been proposed.

Per-Core dynamic voltage and frequency scaling (DVFS) is a common mechanism for controlling performance and improving processor energy efficiency [23, 33, 47]. Per-core frequency management enables performance adjustments at fine time scales [21, 54], and it can also be used as a throttling mechanism to manage contention and resource usage [19, 22].

Cache partitioning is another well-studied mechanism that provides performance isolation of processes that are collocated and that is used to limit variation and contention [7, 9, 24, 32, 39, 53, 55]. Prior work established the effectiveness of cache partitioning as a QoS mechanism and it has recently been implemented in commercial processors. However, because of the large capacity of the last level cache, changes to cache partitions take significant time to have an impact on execution, an effect termed *cache inertia* [32]. Thus, cache partitioning is effective at relatively long time scales.

While we do not currently use these mechanisms in Dirigent as they are not yet available in commercial processors, there is also a large body of work on QoS mechanisms for managing memory bandwidth and latency resources. Yun et al. studied the performance benefits of memory bandwidth reservation for latency-sensitive applications [61]. Mutlu et al. show different levels of QoS goals and performance benefits that can be achieved by making memory scheduling QoS aware [34, 48–50]. Usui et al. presented QoS aware memory scheduler that handles different priority levels in heterogeneous systems [58]. In other related work, Ebrahimi et al. proposed source throttling, which is a hardware-based mechanism controls the rate at which cores generate requests to shared memory system [14]. Jeong et al. studied the row-buffer locality interference in multicore processors [29]. Zhou et al. proposed an architecture that allows fine-grain micro-architecture resource partitioning among threads [65]. Ma et al. designed a mechanism to control queueing delay of requests in different architecture structures [44]. While other contention sources exist in warehouse scale datacenters, there is a large body of related work that show how intelligent task schedulers can identify and avoid such resource conflicts [12, 13, 30, 36, 37].

4. Dirigent Design and Implementation

Dirigent is composed of three main components: profiler, predictor, and controller. The profiler examines and records the execution of an FG applications offline and in isolation. The profiling information is then used online to predict the expected execution time of the FG application. The controller then partitions resources and throttles tasks to minimize the execution time variation of FG tasks while providing as much resources as possible to BG tasks, thus achieving the goal of simultaneously meeting latency-critical requirements and allowing high processor utilization. The rest of this section discusses in detail the design and initial implementation of Dirigent. Again, we point out that Dirigent is unique in the fine time scales on which it operates.

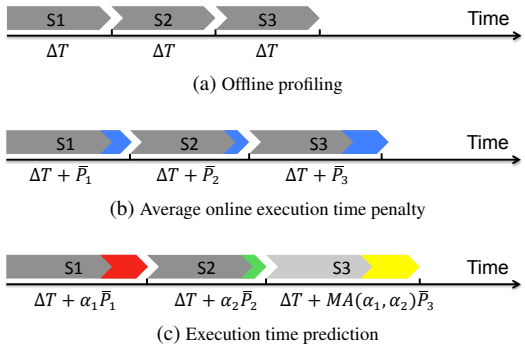
We implement Dirigent in C++ and evaluate it using a 6-core Intel Xeon E5 2618L-v3 processor, which supports per-core frequency settings and cache partitioning [24].

4.1 Offline Execution Profiler

Dirigent profiles the execution of an FG application when running alone offline. The profiler records progress information that is then used to make accurate online predictions of completion time under contention while a task is still executing. The Dirigent profiler periodically samples the execution progress of the FG task being profiled by measuring and recording a series of (*time*, *progress*) pairs when the FG is running alone without any contention (see Figure 3a). The FG program in the example of Figure 3a has 3 profiled segments and takes $3\Delta T$ time units to execute, where ΔT is the sampling period. We measure *progress* by counting the number of retired instructions using the processor's model-specific performance counter monitors [25], but more abstract metrics can also be used. Note that progress can significantly differ between segments even though the sampling frequency is constant. This is because progress depends on the instruction mix, data access pattern, data set size, and other factors.

Dirigent requires minimal profiling. In our current implementation, the profiler requires no extra hardware as both execution time and instruction count are readily available in most architectures today. Dirigent uses performance counters and `sleep` method for periodic sampling with negligible impact on the running application. Although performance overhead is not a great concern in offline profiling, this low overhead ensures the accuracy of these measurements. Furthermore, note that while our current implementation of Dirigent relies on offline profiling, it is possible to perform online profiling instead, which requires the system to run the FG task a few times while all other tasks are paused to record a stable profiling record.

4.2 Execution Time Predictor



(a) Offline profiling

(b) Average online execution time penalty

(c) Execution time prediction

Figure 3: Execution time predictor example.

To predict FG completion time and control resources, Dirigent periodically samples progress during contended online execution. Dirigent predicts FG completion time by tracking actual progress, comparing this progress to the profiled data, and computing a *time penalty* experienced by the FG process, which is then projected forward to completion as explained below. We pin the Dirigent runtime thread to a core that runs a BG task and again use the `sleep` method to periodically interrupt the BG task and execute the Dirigent runtime task.

The time penalty of a specific segment is computed assuming a fixed rate of progress within each segment. The penalty is the difference between the expected time to make the amount of progress within the profiled segment at the rate of progress experienced in the online segment vs. the profiled time for this segment. This is summarized in Equation 1 where P_i is the penalty for the i^{th} segment and ΔT_i is the duration of that segment. Note that ΔT_i can be slightly different than ΔT in the real implementation because of factors such as errors in timers. We account for that difference in Dirigent to ensure the accuracy of the prediction. Also, we introduce the symbol α_i shown in Equation 1 as shorthand for the ratio of measured vs. expected progress rates for the i^{th} segment.

$$P_i = \frac{Profiled\_progress_i}{Measured\_progress_i}\Delta T_i - \Delta T_i = (\alpha_i - 1)\,\Delta T_i \tag{1}$$

Frequent samples increase the execution time prediction accuracy and the opportunities for performance management, but each prediction and control segment has overhead. We measured this overhead using a subset of our workload mixes. Results show the runtime overhead is minimal and each Dirigent invocation requires on average less than $100\mu s$ (including predictor and throttler). We therefore chose a sampling period $\Delta T = 5ms$ to balance the overhead and effectiveness of online prediction and control. This sampling period provides 100 or more segments in all the FG applications we test with only negligible runtime overhead.

To increase prediction accuracy, Dirigent maintains an exponential moving average (with weight 0.2) of the penalty within each segment across multiple executions of the FG task, which we denote $\overline{P_i} = 0.2P_i + 0.8\overline{P_i}$. Figure 3b shows the average penalty in blue. Note that just like progress, the average penalty can differ significantly across segments. The moving average smooths occasional outlier executions. At a given point in time during a single FG task's execution, Dirigent's predictor uses the penalties observed so far for each segment, the total elapsed time from the start of the process, and the average penalties of the segments yet to execute to compute the expected execution time of the task. The formula is shown in Equation 2, where k is the segment corresponding to current time T, N are the total number of segments in the profiled execution, and $MA\left(\{\alpha_i\}_{i=1}^{k}\right)$ denotes an exponential moving average over the rate factors measured so far in the current execution. This moving average is used as the expected penalty scaling factor for the remainder of the current execution.

$$T_{est,k} = T + \sum_{i=k+1}^{N}\left(MA\left(\{\alpha_i\}_{i=1}^{k}\right)\overline{P_i} + \Delta T_i\right) \tag{2}$$

Figure 3c illustrates how the prediction calculation of Equation 2 is performed. When the example program finishes executing the second segment, the penalty scaling factor α_2 is much smaller than the factor α_1 of the first segment. This difference arises from factors such as OS noise, phase changes of the BG applications, and context switches. The moving average across executions and of executions within the segment smooths out the difference of the scaling, which is then used as the predictive factor for the remaining segment.

In our experiments with Dirigent, we arbitrarily chose a weight of 0.2 for the exponential moving averages and a 5ms sampling interval. As we show in Section 5.2, the predictor is highly accurate with these parameters and is able to predict the expected execution time to within 2%, typically, across multiple applications and different levels of contention. We tested the sensitivity of Dirigent to weight factors in the range of $0.1 - 0.3$ and conclude that Dirigent is robust. We also evaluate Dirigent's sensitivity to sampling periods. We conclude that even 40 samples per execution of the FG task tested provide for accurate completion-time predictions. However, the low performance overhead ($< 100\mu s$ per invocation) enables high sampling frequency for accurate prediction and tight QoS control for FG tasks that widely differ in execution time.

4.3 Performance Controller

Dirigent monitors the performance of FG applications online and uses the predictor to determine whether these applications are progressing faster or slower than necessary to meet their latency goals. Recall that Dirigent does not strive

to minimize the execution time of FG tasks, but rather to minimize their execution time variation while meeting their latency targets. As a result, resources for the FG tasks are not over-provisioned. If a FG task is expected to complete before its target time, it is deprioritized and BG tasks can achieve higher throughput. On the other hand, if a FG task is lagging, Dirigent prioritizes resources toward that FG task and away from BG tasks.

In our current implementation of Dirigent, we allocate resource by controlling the frequency at which each core operates, by partitioning the last-level cache (LLC), and by pausing BG tasks when necessary. We chose these mechanisms from the possible ones described in Section 3.2 because they are both effective and available in current systems. We use frequency and task-pausing to control FG progress at fine time scales and cache partitioning at a coarser ones; with large caches, *cache inertia* means significant time passes before the impact of adjusting partitions takes effect [32].

Fine time scale control: The goal of the fine time scale controller is to quickly respond to changes in contention and FG task progress to ensure deadlines are met and minimize performance variance. The controller observes the execution time predictions and decides whether FG tasks can yield resources or whether BG tasks must be throttled and to what extent. A simplified version of the fine time scale controller policy for a single FG task is designed as follows.

At each decision point, the controller determines if the FG task is ahead or behind. If ahead, the controller will check the following three options in order. First, if any background jobs are paused, the control decision is to continue them. Second, if no tasks are paused but some are throttled, the decision is to speed up any throttled BG processes by one speed grade (using existing per-core DVFS mechanisms). Third, if all BG tasks are already running at their maximum frequency, the decision is to throttle the FG task frequency. Similarly, if the FG task is behind schedule, the decision is to speed up to maximum frequency. If it is already at maximum frequency, the decision is to immediately throttle the frequency of the BG tasks. If the BG tasks are already at the minimum frequency, the most intrusive active BG is paused; we define intrusiveness as the number of LLC load misses a task generates, which we obtain from existing performance counters. The throttler controls each core using the `CPUFreq Governor` of Linux [4].

While the Dirigent runtime is very lightweight, the impact of control decisions is not instantaneous. We therefore only make control decisions every some small number of prediction segments (5 in our experiments, arbitrarily). Furthermore, we only take control actions if the expected FG execution time is more than 2% ahead the target deadline and only pause BG tasks if the FG task is expected to complete more than 10% behind its deadline. We chose 2% because it corresponds to the typical error of the predictor and is thus a good safety margin that prevents prematurely slowing down or interfering with a FG task. We chose a larger threshold for pausing because its overhead is greater (again, the value of 10% was arbitrarily chosen within a reasonable value range; sensitivity studies reveal that Dirigent is not sensitive to this choice).

The decision making process is slightly more complicated when there are multiple collocated FG processes along with BG tasks. Each FG task may exhibit different levels of performance degradation even when the interference level is the same for all of them. Furthermore, any action taken on BG tasks will impact all collocated FG tasks. As a result, when all FG tasks show the same performance tendency, we use the same policy described before for a single FG process. Otherwise, BG tasks are throttled based on the performance of the slowest concurrent FG task, and any other FG tasks that are expected to finish sooner than the deadline are throttled down individually.

Coarse time scale control: Dirigent uses cache partitioning for coarse-grain control over the expected execution time of FG tasks, specifically the Cache Allocation Technology recently introduced by Intel [24], which can be used to specify which cache ways may be used by each processor. Because of cache inertia the system's response time to partition changes is fairly slow when compared to the typical short durations of FG tasks. We therefore use statistics collected over multiple executions of a FG task to guide adjustments to cache partitioning. Specifically, our current implementation of Dirigent tracks three measures: (1) the correlation between a FG tasks' execution time and the LLC misses it generates (over multiple executions); (2) a history of the absolute number of LLC misses over executions; and (3) a history of the Dirigent decisions states over time. In our current implementation, we use the history of 10 last executions to compute the measures above. We construct three heuristics to determine whether FG tasks benefit from greater isolation and more dedicated cache ways or whether BG tasks are allowed to utilize a greater portion of the LLC.

First, if there is strong correlation between the execution time of FG tasks and their LLC misses, it indicates that growing the FG partition is likely to improve FG performance. Therefore, if correlation is strong *and* FG tasks have recently missed deadlines, we increase isolation and add one LLC way to the FG partition (removing it from the list of ways utilized by BG tasks). We somewhat arbitrarily chose a correlation coefficient of 0.75 as the threshold determining strong correlation.

Second, Dirigent observes the LLC hit-rate history and if growing the FG partition does not lower FG tasks LLC misses, Dirigent shrinks the foreground partition. This heuristic coupled with the coarse time scale and averaging performed prevents the FG partitions from continuously growing due to anomalous executions.

It is also possible for the correlation between misses and performance to not be strong yet for partitioning to still help:

Table 1: FG and BG Benchmarks

Type	Name	Description
FG	bodytrack	Body tracking of a person
	ferret	Content similarity search
	fluidanimate	Fluid dynamic for animation
	raytrace	Real-time raytracing
	streamcluster	Online clustering of an input stream
Single BG	bwaves	Simulation of blast waves in 3D
	PCA	Principal Component Analysis
	RS	Range Search
Rotate BG	namd	Biomolecular system simulation
	soplex	Linear program solver
	libquantum	Simulation of quantum computer
	lbm	Simulation of fluids with free surfaces

when FG performance is bottlenecked by high memory latency caused by contention from BG tasks. Because Dirigent's fine time scale controller throttles BG tasks when they heavily contend for resources, correlation would not detect the need for stricter partitioning of FG and BG tasks. Therefore, our *third heuristic* grows the FG partition when the controller history indicates that BG tasks are heavily throttled and their utilization of core resources is low. The second heuristic then differentiates between scenarios where BG tasks should be throttled from those where partitioning is more beneficial by shrinking the FG partition back if hit-rate does not improve. We show the effectiveness of our method in Section 5.3.

5. Evaluation

We evaluate Dirigent on a real machine with a range of workloads representative of a wide range of FG and BG behaviors. We first introduce the evaluation infrastructure and workloads. We then discuss a set of experiments that demonstrate the accuracy of Dirigent's completion-time predictor, the effectiveness of the coarse time scale partitioning heuristic, Dirigent's performance benefits compared to a baseline configuration and configurations that roughly correspond to prior work, and the new tradeoff between BG task throughput and FG deadline target that Dirigent enables.

5.1 Workloads and Evaluation Infrastructure

System: We evaluate Dirigent on a 6-core Intel Xeon E5-2618L v3 server processor. The nominal per-core maximum frequency is 2GHz and 9 frequency steps are available for throttling ($1.2 - 2.0$GHz, though Dirigent uses just 5 equispaced frequencies). Turbo Boost is enabled in all experiments. The processor has a 15MB L3 LLC and supports Intel's Cache Allocation Technology, which enables us to partition the cache between FG and BG tasks to provide additional isolation. The system is configured with 4 2133MHz DDR4 channels and has a total of 16GiB. We run a Linux 3.13.0 kernel at runlevel S, which provides an environment with little OS interference and good facilities for implementing both the Dirigent predictor and controller, with an efficient `sleep` implementation and the built-in `CPUFreq Governer`, respectively. We pin all tasks to individual cores and Dirigent is pinned to a core that is shared with a BG task. We set the BG processes to have higher process niceness than the Dirigent runtime and FG processes to have the lowest niceness. We use the configuration above with no explicit resource management as the *baseline configuration*.

Workloads: Table 1 lists the benchmarks we use in the evaluation. We select the subset of PARSEC applications that represent latency-sensitive applications as FG tasks [3]. We chose PARSEC as it is designed to represent emerging and user-facing workloads of the type that are being offloaded to cloud systems. We use a single run of each benchmark with sim-medium inputs as a single FG task. As shown in Figure 4, these tasks span a range of completion times ($0.5 - 1.6$s) and LLC miss rates. The figure shows the behavior of the FG benchmarks both when running alone and under contention. For this figure, we use 1 FG for all FG tasks and 5 BG cores all running *bwaves*, which falls in the middle of contention range. The FG workloads are all fairly compute-intensive, making them good offload candidates. While they are compute-intensive, they still offer a range of sensitivity to interference from BG tasks both because of LLC and memory access contention. This can be seen by the different correlation levels between LLC miss rate increase and execution time degradation between the different benchmarks. To ensure accurate measurements of these tasks, execution time is measured inside the FG processes using PARSEC's Region of Interests (ROI) interface.

We use two kinds of BG workloads to represent different interference types: BG phase changes and context switches. We use three standalone BG workloads that exhibit strong phase change behavior: the scientific simulation *bwaves* from SPEC 2006 [18] and the machine learning applications Principal Component Analysis (*PCA*) and Range Search (*RS*) from MLPack [10]. All other benchmarks we examined did not provide strong phase behavior, at least with respect to impact on interference, and we omit them from the evaluation—such workloads do not pose significant challenges to the Dirigent predictor.

To mimic varying interference caused by context switches, we select four applications from SPEC 2006 that exhibit a range of memory intensiveness [18], though we arbitrarily chose between all benchmarks that exhibit similar intensity. We then form two-benchmark workloads and randomly switch between the two paired benchmarks each time a FG task completes. The pairs we use are (*lbm+namd*), (*lib+namd*), (*lbm+soplex*), and (*lib+soplex*). We refer these BG workloads as Rotate BG workloads. Figure 5 summarizes the different behaviors of the BG workloads while using a single core running *ferret* as a representative FG workload. The blue bars show the total number of L3 load misses per thousand FG instructions generated by all 6 cores.

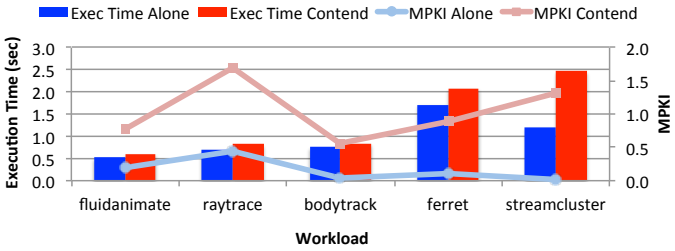


Figure 4: Overview of FG Workloads.

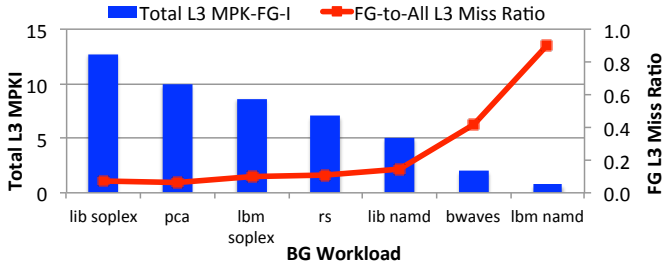


Figure 5: Overview of BG Workloads.

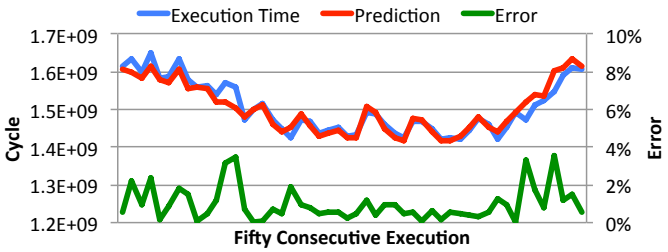


Figure 6: Prediction Trace for *Raytrace* with *RS*.

The red curve shows the fraction of misses generated by FG tasks, which can be interpreted as the ratio between FG task and total memory bandwidth consumption. As can be seen, the BG workloads cover a wide spectrum of behaviors and contention pressure. Similar trends are observed when mixing these BG tasks with other FG workloads.

5.2 Predictor Accuracy

Since the throttling actions of Dirigent are guided by the execution time predictor, it is important to validate its accuracy. Figure 6 shows the execution time, prediction results, and prediction error for 50 consecutive executions of *raytrace* and *RS* workload in the baseline configuration (no explicit resource management). The results shown correspond to a completion-time prediction that is made about half-way through a FG task's execution. Predicted completion closely tracks the actual completion time. Figure 7 shows the average predictor accuracy and the completion time standard deviation normalized to the mean of each workload for all

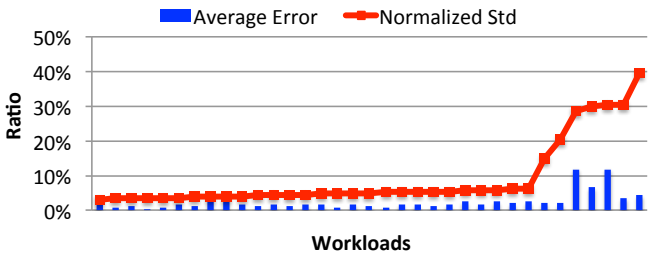


Figure 7: Prediction Accuracy for all FG-BG mixes.

35 workload combinations we use (combinations of one of the 5 FG benchmarks with each of the 7 BG workloads). Average error (ϵ) is computed as shown in Equation 3 at the midpoint of each task over 100 consecutive task executions. The predictor is highly accurate across all these workload combinations with an overall average error of just 2.4%. As expected, higher execution time variation poses greater challenges and predictions tend to be less accurate in such cases; the 5 points with average error of $> 4\%$ all use streamcluster as the FG, with each of the BG we included in the workload. Among those, *RS* gives the highest error rate (12.5%) and the mix between *libquantum* and *namd* gives the lowest error rate (4.4%). Note that the standard deviation of execution time in these cases is significantly larger than predictor errors.

$$\epsilon = \frac{1}{N}\sum_{i=1}^{N}\frac{|predict_i - measure_i|}{measure_i} \quad (3)$$

5.3 Coarse Time Scale QoS Control

To verify the effectiveness of the heuristics used in the coarse time scale QoS controller in Dirigent, we conducted an exhaustive search on cache partitions for 5 arbitrarily chosen workload mixes and show one of the results, for *streamcluster* as the FG task and *PCA* as the BG task in Figure 8. We chose this combination to present because it is one of the few workloads in which FG tasks require a larger partition and therefore stresses the heuristic. As the partition dedicated to the FG tasks grows, the performance of the FG task improves. The knee of the curve representing this exhaustive search is at 5 ways. Dirigent's coarse time scale controller converges to this same partition after just 32 FG task executions (5 coarse time scale controller invocations).

5.4 Dirigent Performance

We use five configurations to evaluate the effectiveness and benefits of Dirigent. In the *Baseline* configuration all cores run at the highest frequency and freely contend for resources. *StaticFreq* sets the FG cores to run at the highest allowed frequencies (2GHz) and BG cores to run at the slowest speed (1.2GHz), giving more resources to FG tasks. *StaticBoth* sets the best static cache partitioning (corresponding

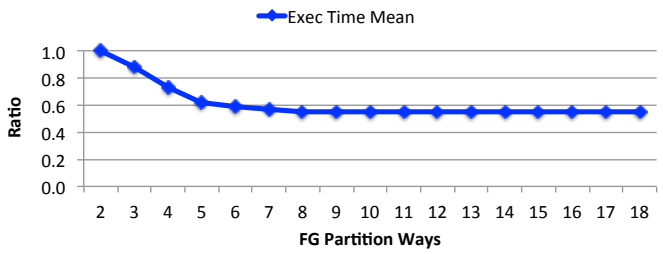


Figure 8: Exhaustive Search on Partition Size.

to Dirigent's heuristic, which we verified is near-optimal) as well as the best frequency for the BG cores. *DirigentFreq* uses Dirigent's fine time scale control only and does not use cache partitioning. *Dirigent* is the full Dirigent implementation that combines coarse time scale cache partitioning with fine time scale frequency control. Note that we omit the coarse time scale-only configuration of Dirigent because it performs just slightly worse than *StaticBoth* because both use the same partition. Our understanding is that the *StaticBoth* configuration is very similar to the behavior of Heracles [42] in our scenario because the execution time of FG tasks in our experiments are much shorter than polling intervals and controller's optimization convergence time in [42]; our workloads also do not exercise the network.

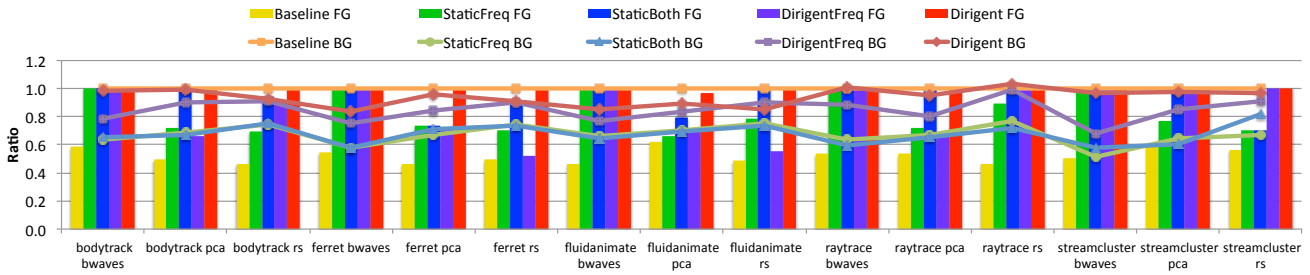
To quantify the benefits of reduced variation in FG task execution time, we define the deadline for each FG task to be $\mu_{Baseline}+0.3\sigma_{Baseline}$, where $\mu_{Baseline}$ and $\sigma_{Baseline}$ are the average and standard deviation of FG completion time in the *Baseline* configuration; PARSEC does not define latency goals even though the applications do represent potential user-facing tasks. We set the deadline for each benchmark to be slightly larger than the uncontended run time of a task but still far smaller than when contention is unmanaged. In this way Dirigent has the FG run time slack to allow BG jobs to run. The tradeoff between deadline tightness and system throughput is demonstrated later in Section 5.5. For FG tasks, we focus on the *FG success ratio*, which is computed as the fraction of FG task executions that complete within the deadline defined above. For BG tasks, we report *BG performance*, which is the total number of instructions executed during the experiment (across all cores) normalized to *Baseline*. We normalize BG throughput to *Baseline* because running with no constraints results in the highest BG throughput.

Single FG process: Figure 9a and Figure 9b report the performance of both FG and BG tasks in all the workload mixes with one FG process and five BG processes, respectively. The results are summarized in Figure 10 with arithmetic mean of FG success rate and harmonic mean of relative BG throughput.

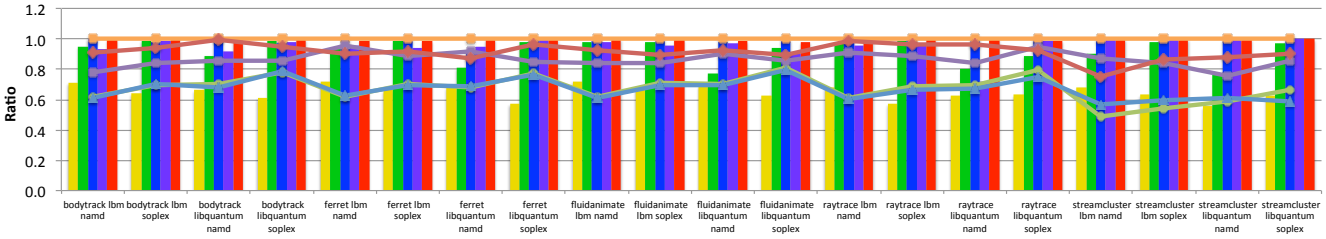
We make three key observations about these results. First, while BG performance is high with *Basline*, the FG success rate is very poor, averaging just under 60%. Second, while the (semi-)static mechanisms significantly improve FG completion rate (to nearly 100% in the case of *StaticBoth*, BG performance is severely degraded. On average, BG throughput is reduced to $\sim 60\%$ that of *Baseline*. Not shown in the graphs is that the FG throughput is improved by 7.5% on average with *StaticFreq*; this is because more BG tasks are throttled more than with the other configurations. Third, the importance of fine time scale control is clearly demonstrated by both *DirigentFreq* and *Dirigent*. Even without cache partitioning, *DirigentFreq* is able to meet the 95% completion target for all but a few workloads and is able to consistently deliver better BG performance than the static schemes (85% of *Baseline* on average). Finally, *Dirigent*, which combines both fine and coarse time scale control, is able to consistently match or exceed both the FG success rate ($> 99\%$ on average and 97% in worst success) and BG throughput of all other managed schemes simultaneously; the BG throughput of *Dirigent* comes very close to the throughput of unconstrained execution, averaging 92% and never dropping below 75% of *Baseline*.

We further look at the execution of one of the workload mixes, a *ferret* FG task collocated with five *RS* BG tasks. Figure 11 shows the execution time probability density function curves for the five configurations. Results show that the curves for *Baseline* and *StaticFreq* stretch wide horizontally. Comparing to the *StaticBoth*, the *DirigentFreq* is able to significantly reduce execution time variation by moving the two peaks in *StaticBoth*'s curve closer. *Dirigent* is able to further reshape the curve and merge the two peaks together, achieving even more predictable performance and better BG job throughput. Figure 12 demonstrates the distribution of frequencies that *DirigenttFreq* and *Dirigent* use for cores running BG tasks, and show that partitioning the cache significantly reduces the performance contention on FG performance, allowing BG process to run safely at much higher frequency on average. Overall, *Dirigent* can achieve 85% reduction in the standard deviation of execution time of FG tasks at the cost of only 9% of BG performance loss across all the workload mixes we tested. *DirigentFreq* captures part of the benefits in reducing performance variation, achieving 70% reduction in standard deviation of execution time, but suffers from higher BG performance loss at 15%.

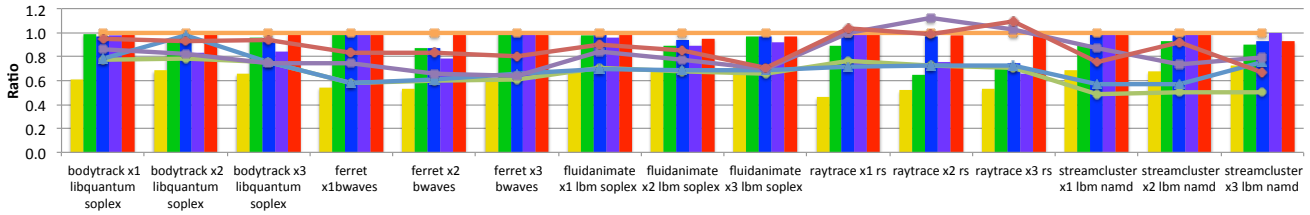
Concurrent FG Processes: Figure 9c uses the same metrics but with workload mixes that have multiple FG processes. These detailed per-workload results are summarized in Figure 13. Due to the large number of possible combinations, it is impractical to exhaustively experiment with all of them. We therefore select five combinations that cover a low to high performance variation range in *Baseline* and measure the performance of these mixes using a varying number of concurrent FG tasks. The workloads in Figure 9c are sorted in ascending order of number of concurrent FG processes



(a) Single BG Workload Mixes



(b) Rotate BG Workload Mixes



(c) Multiple FGs Workload Mixes

Figure 9: Comparison of FG and BG Performance.

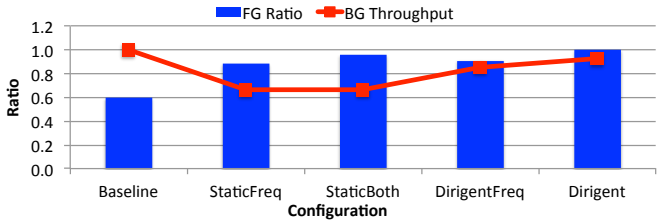


Figure 10: Summary of All Single FG Workload Mixes.

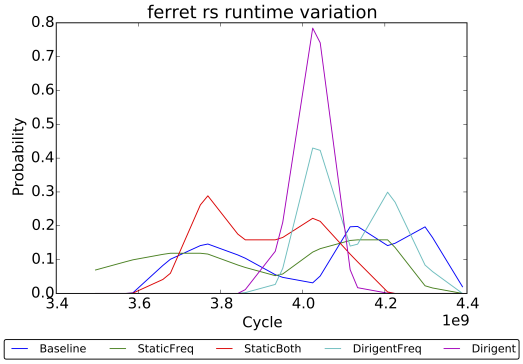


Figure 11: Execution Time Probability Density Function Curve.

within each pair; the total number of FG and BG processes is always 6 (the number of cores).

Overall, the results show similar trends and observations to those exhibited by single-FG process workloads. We discuss two additional insights. First, each FG task may experience different levels of contention due to different FG-BG phase interleavings. This forces the fine-grain controller to use conservative BG performance settings to try and allow even the slowest FG task to complete on target. As seen in Figure 9c, within each FG-BG workload mix with *Dirigent-Freq*, the general trend is that BG throughput decreases with each additional FG task. This problem is alleviated by the introduction of cache partitioning, as it effectively isolates most of the performance interference between FG and BG tasks. Second, since all FG tasks share the same partition, increasing the number of FG tasks also increases their performance variation. This is shown in Figure 14, where standard

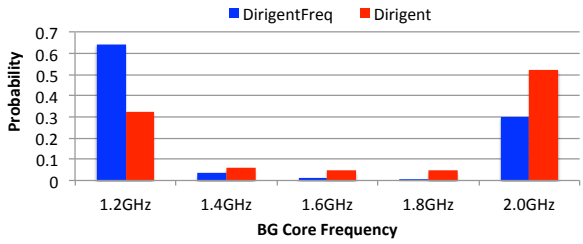


Figure 12: BG Core Frequency Distribution.

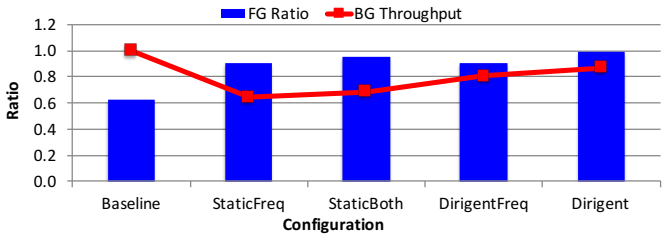


Figure 13: Summary of All Multiple FG Workload Mixes.

deviation for each configuration is normalized to the value in the baseline. While the results show increased variance with more FG processes within each workload mix, *Dirigent* is still able to effectively reduce the performance variation with low BG task performance overhead.

5.5 FG Throughput and BG Performance Tradeoffs

So far we evaluated Dirigent with a fixed FG target execution time that reflects the baseline FG throughput. We now show the tradeoff that Dirigent enables between FG task throughput and BG task performance with precise QoS control. Figure 15 shows one arbitrarily chosen workload: a single *raytrace* FG process and 5 BG *bwaves* processes. We gradually increase the target completion time from the average completion time in standalone execution until it is larger than the average *Baseline* execution time. The blue bars show the average execution time normalized to the standalone execution time, the red bars show the standard deviation at each target normalized to that of *Baseline*, and the green curves show the BG task throughput normalized to *Baseline*.

Dirigent is able to accurately control the execution time across the entire range of target deadlines; the only exception is when the target is set at the standalone execution time because there is no opportunity for collocation without violating QoS. When the deadline is set higher, Dirigent exploits opportunities when FG tasks are running faster than necessary and converts them into BG performance. Importantly, Dirigent effectively enables the tradeoff between FG throughput and BG performance at high system utilization by reducing the variance of FG task execution time. Note that we do not plot the success rates because Dirigent consistently achieves them at the desired $> 99\%$ rate.

6. Related Work

Online QoS Management: Mars et al. proposed an interference characterization methodology and QoS management schemes that target data center applications [45, 60]. Heracles is a performance management runtime that uses multiple control modules leveraging software and hardware mechanisms to enforce QoS for latency sensitive tasks that are collocated with batch jobs [42]. Zhang et al. proposed to identify interference using cycle-per-instruction data, and the results can be used to enforce QoS by static and manual throttling [62]. The above work is most closely related to Dirgient, however, these mechanisms use only coarse time scale statistics and performance variation is not discussed or addressed. As a result the system lacks the ability to adjust contention at fine time granularity, which we demonstrate is crucial to maximizing utilization.

Other related work studies fine-granularity QoS with a focus on saving the energy of executing only FG tasks. PEGASUS improves the energy efficiency of datacenters by dynamically adjusting the power limit of processors [41]. Adrenaline categorizes queries for target applications and only speeds up ones that are likely to fail QoS goals [21]. TimeTrader and Rubik exploit request queueing latency variation and apply any available slack from queueing delay to FG computation to reduce energy consumption [31, 59]. Suh et al. propose to use various execution time prediction mechanisms to guide frequency scaling to save energy [5, 40]. In contrast, Dirigent converts variation in the run time of FG tasks into improved system throughput.

To the best of our knowledge, Dirigent is the first to trade off the performance of latency-critical jobs that finish sooner than required with higher system throughput for BG tasks. A related mechanism proposed by Min et al. tackles fine time granularity QoS problems for GPUs in heterogeneous platforms [28]. However, the progress heuristics used for the GPU were not general and the mechanism proposed is limited to managing main memory bandwidth contention between the CPU and GPU.

Interference Analysis: Interference is a well studied problem and many models and heuristics have been proposed. A good example is the sophisticated model for predicting multicore interference proposed by Zhao et al. [64]. However, this and other prior models do not address deadline-oriented applications because the prediction is made on coarse time scales. Further, these analysis techniques require complex computation that are not suitable for a dynamic lightweight runtime such as Dirigent [62, 64]. Application Heartbeats is a general progress report framework [20], our profiler uses similar concept but on a millisecond scale.

QoS-Aware Scheduling: In addition to the QoS control mechanisms discussed in Section 3.2, task scheduling across multiple nodes in a cloud server or cluster can also be used to manage contention and performance. Kambadur et al. proposed a sample-based interference prediction methodology

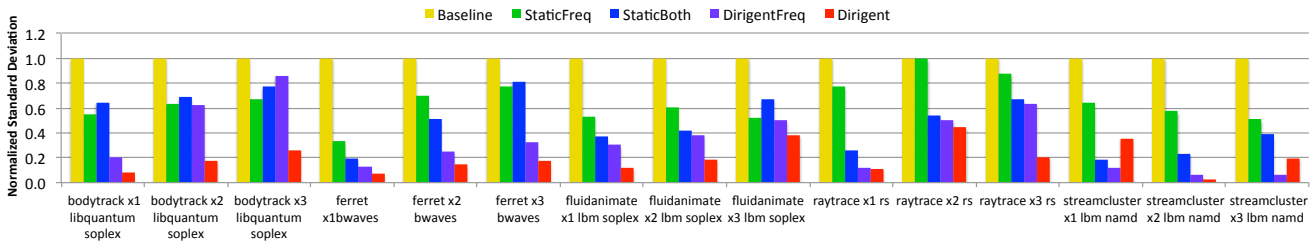


Figure 14: Normalized Standard Variation of Multiple FG Workload Mixes.

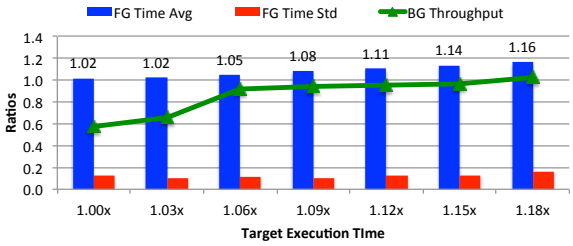


Figure 15: Tradeoff Between FG throughput and BG performance

for identifying application pairs whose collocation should be avoided [30]. SMiTe predicts the SMT-level interference by profiling interference and sensitivity on each kind of shared resources to guide cluster scheduler [63]. Lee et al. designed and implemented a scheduler in the virtual machine hypervisor for soft realtime applications [36]. Leverich et al. analyzed the source of QoS degradations of latency-critical workloads, and devised a new scheduler to handle these issues [37]. Paragon and Quasar are two task schedulers that classify applications and schedule them into data centers by performing resource allocation and assignment to minimizing interference [12, 13]. Q-Clouds and DeepDive are two QoS-aware scheduler that handle contentions and interference in virtual environment [51, 52]. These works are orthogonal to Dirigent and Dirigent can be integrated with these schemes to manage performance on each node.

7. Conclusion

In this work, we expose the problem of performance variation for latency-critical tasks when collocated with batch jobs and the associated challenges and opportunities. We explain how such variation leads to low hardware utilization and resource over-provisioning. Our main insight is that minimizing task-to-task variation offers significant opportunities for improving system utilization without compromising QoS goals. We present the design, implementation, and evaluation of the Dirigent lightweight contention management runtime to exploit these opportunities. Dirigent is effective because it can accurately predict the completion time of a running task at very fine time scales. It can thus control resources during a task's execution to meet deadlines and maximize batch throughput.

We show that Dirigent is particularly well suited for cloud-oriented applications where it is common to run just one latency-critical process per node and use batch-oriented tasks to improve utilization. In this case Dirigent exerts precise control over completion time and resources to boost utilization by $\sim 30\%$ compared to configurations that are similar to previously proposed schemes while providing higher QoS for the latency-critical tasks. We further show that even with multiple concurrent latency-critical tasks, Dirigent is still effective, much more so than alternative techniques, and always achieves very high deadline success rates ($> 98\%$).

We opted to implement Dirigent using existing hardware mechanisms and evaluate it on a real machine. Even without new hardware techniques, Dirigent has low runtime overhead ($< 100\mu s$ per invocation) and we are able to demonstrate the effectiveness and benefits of our techniques. However, for tasks with very tight latency constraints ($\lesssim$ 10ms) additional hardware support may be required. One limitation of the current Dirigent implementation is its dependency on profiling. In this paper we assumed offline profiling but because of the short profiling duration it can be performed online, though it will require pausing all BG tasks while profiling. In future work, we plan to improve the Dirigent prediction and control algorithms to allow concurrent profiling by adding interference offsets into the baseline execution time. A second limitation is that in this first effort with Dirigent, we limited our evaluation to performance variation caused by external interference. Accurate predictions of execution times in the presence of strong input dependence may require interfaces that extend Application Heartbeats [20] or program slicing [40]. Finally, we intend to evaluate Dirigent for parallel applications, where precise control over completion time reduces the overheads associated with synchronization and load imbalance.

Acknowledgment

We thank Huawei, which partially funded this research, the anonymous reviewers who provided valuable feedback and suggestions, and members of the Intel developers forum community who helped answer questions about use of processor features.

References

[1] Alia Atlas and Azer Bestavros. Statistical rate monotonic scheduling. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*. IEEE, 1998.

[2] Luiz Andres Barroso and Urs Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan Claypool, 2009.

[3] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] Dominik Brodowski and Nico Golde. CPU Frequency and Voltage Scaling Code in the Linux kernel.

[5] Tao Chen, Alexander Rucker, and G Edward Suh. Execution time prediction for energy-efficient hardware accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[6] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007.

[7] Derek Chiou, Prabhat Jain, Srinivas Devadas, and Larry Rudolph. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference*. Citeseer, 2000.

[8] Jason Clemons, Haishan Zhu, Silvio Savarese, and Todd Austin. Mevbench: A mobile computer vision benchmarking suite. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011.

[9] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ACM SIGARCH Computer Architecture News*. ACM, 2013.

[10] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 2013.

[11] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 2013.

[12] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 2013.

[13] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 2014.

[14] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ACM Sigplan Notices*. ACM, 2010.

[15] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Prefetch-aware shared resource management for multi-core systems. *ACM SIGARCH Computer Architecture News*, 2011.

[16] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*. ACM, 2007.

[17] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[18] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[19] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009.

[20] Henry Hoffmann, Jonathan Eastep, Marco D Santambrogio, Jason E Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th international conference on Autonomic computing*. ACM, 2010.

[21] Chang-Hong Hsu, Yunqi Zhang, Michael Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, Ronald G Dreslinski, et al. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015.

[22] Ramesh Illikkal, Vineet Chadha, Andrew Herdrich, Ravi Iyer, and Donald Newell. PIRATE: QoS and performance management in CMP architectures. *ACM SIGMETRICS Performance Evaluation Review*, 2010.

[23] Intel. Intel Product Information.

[24] Intel. Cache Monitoring Technology and Cache Allocation Technology.

[25] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals.

[26] Ravi Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 2004.

[27] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[28] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012.

[29] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Mike Sullivan, Ikhwan Lee, and Mattan Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012.

[30] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. Measuring interference between live datacenter applica-

tions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[31] Harshad Kasture, Davide B Bartolini Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[32] Harshad Kasture and Daniel Sanchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. *ACM SIGARCH Computer Architecture News*, 2014.

[33] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE, 2008.

[34] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010.

[35] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 2013.

[36] Min Lee, AS Krishnakumar, Parameshwaran Krishnan, Navjot Singh, and Shalini Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *ACM Sigplan Notices*. ACM, 2010.

[37] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[38] Chit-Kwan Lin and H. T. Kung. Mobile app acceleration via fine-grain offloading to the cloud. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. USENIX Association, 2014.

[39] Fang Liu, Xiaowei Jiang, and Yan Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010.

[40] Daniel Lo, Taejoon Song, and G Edward Suh. Prediction-guided performance-energy trade-off for interactive applications. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[41] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st annual international symposium on Computer architecuture*. IEEE Press, 2014.

[42] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015.

[43] Ying Lu, Tarek Abdelzaher, Chenyang Lu, and Gang Tao. An adaptive control framework for qos guarantees and its application to differentiated caching. In *Quality of Service, 2002. Tenth IEEE International Workshop on*. IEEE, 2002.

[44] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, et al. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (pard). In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[45] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou 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*. ACM, 2011.

[46] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV. ACM, 2009.

[47] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N Patt. Predicting performance impact of dvfs for realistic memory systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012.

[48] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011.

[49] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007.

[50] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2008.

[51] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*. ACM, 2010.

[52] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. Technical Report 183449, EPFL, 2013.

[53] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006.

[54] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. Computational sprinting. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012.

[55] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*. ACM, 2011.

[56] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 2004.

[57] Techspot. Facebook to build a $1 billion wind-powered data center in Fort Worth.

[58] Hiroyuki Usui, Lavanya Subramanian, Kevin Chang, and Onur Mutlu. Squash: Simple qos-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *arXiv preprint arXiv:1505.07502*, 2015.

[59] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and TN Vijaykumar. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[60] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*. ACM, 2013.

[61] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013.

[62] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, New York, NY, USA, 2013. ACM.

[63] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[64] Jiacheng Zhao, Huimin Cui, Jingling Xue, Xiaobing Feng, Youliang Yan, and Wensen Yang. An empirical model for predicting cross-core performance interference on multicore processors. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, 2013.

[65] Yanqi Zhou and David Wentzlaff. The sharing architecture: sub-core configurability for iaas clouds. In *ACM SIGARCH Computer Architecture News*. ACM, 2014.