Enabling Fair Pricing on HPC Systems with Node Sharing

Alex D. Breslow* Ananta Tiwari† Martin Schulz‡
Laura Carrington† Lingjia Tang§ Jason Mars§

*University of California, San Diego, CA, USA, abreslow@cs.ucsd.edu
†San Diego Supercomputer Center, La Jolla, CA, USA, {tiwari, lcarring}@sdsc.edu
‡Lawrence Livermore National Laboratory, Livermore, CA, USA, schulzm@llnl.gov
§University of Michigan, Ann Arbor, MI, USA, {lingjia, profmars}@eecs.umich.edu

ABSTRACT

Co-location, where multiple jobs share compute nodes in large-scale HPC systems, has been shown to increase aggregate throughput and energy efficiency by 10 to 20%. However, system operators disallow co-location due to fair-pricing concerns, i.e., a pricing mechanism that considers performance interference from co-running jobs. In the current pricing model, application execution time determines the price, which results in unfair prices paid by the minority of users whose jobs suffer from co-location.

This paper presents POPPA, a runtime system that enables fair pricing by delivering precise online interference detection and facilitates the adoption of supercomputers with co-locations. POPPA leverages a novel shutter mechanism – a cyclic, fine-grained interference sampling mechanism to accurately deduce the interference between co-runners – to provide unbiased pricing of jobs that share nodes. POPPA is able to quantify inter-application interference within 4% mean absolute error on a variety of co-located benchmark and real scientific workloads.

Keywords

Online Pricing, Supercomputer Accounting, Resource Sharing, Chip Multiprocessor, Contention

1. INTRODUCTION

Supercomputers typically have hundreds to thousands of users and consist of tens to thousands of individual servers connected over a high-speed optical interconnect. At any one time, many users concurrently utilize the system. The current approach has been to give each user a non-overlapping set of compute nodes on which to run his or her application. While this approach prevents jobs from different users from clobbering one another, it leads to a missed performance opportunity. In fact, recent work has shown that co-location, where a set of jobs from different users runs on a shared set of compute nodes, can increase mean application performance and system energy efficiency by 20% by reducing contention for shared resources in the memory subsystem and inter-node network [38, 33, 20]. In addition, current architectural trends and exascale computing studies suggest that the benefit of co-location is likely to increase. The studies project that compute nodes will have hundreds to thousands of cores [16]. For some applications, it may not be possible to use all of these cores efficiently. In particular, 80% of all XSEDE jobs use less than 512 cores [11, 45], which means co-location will likely be necessary to utilize all of a node's cores.

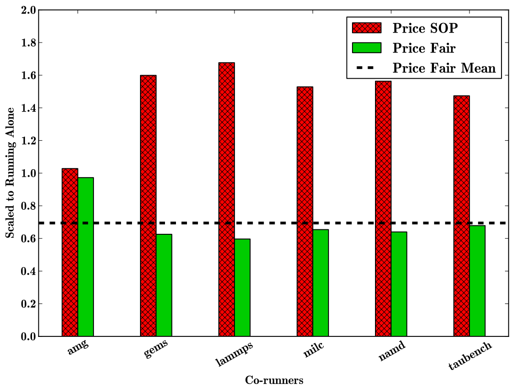


Figure 1: Performance of GTC, a plasma physics code, when co-located with the applications on the x-axis. The current pricing mechanism penalizes the user for co-locating their job by charging them more when their job degrades more.

Co-location seems inevitable for larger jobs as well. Projected scaling trends suggest an increase in the number of cores per node that outpaces increases in memory bandwidth and cache capacity, which will reduce the resources available per core [16]. To mitigate contention, resource-hungry jobs will have to be spread out over more compute nodes and paired with resource-light jobs to maintain high system utilization [20].

Although co-location is beneficial to performance and energy efficiency, it also creates a new set of challenges, one of which is *fair pricing*. Fair pricing is a concern because although there is a net benefit from co-location, some pairings can cause one of the applications to slow down. When this happens, we argue that the user should be discounted. However, if we apply the current state-of-practice (SOP) in HPC infrastructures, where users are billed proportionally to the

SC13 November 17 - 21, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11
http://dx.doi.org/10.1145/2503210.2503256 ...$15.00.

time to execute their job, we find there is gross inequity – users whose jobs benefit from co-location pay comparatively less while users whose jobs do not benefit pay more.

Figure 1 illustrates the challenge. Under the current state-of-practice, a user running GTC[41], a plasma physics code, pays 60% more when co-located with LAMMPS[3], a molecular dynamics code, versus AMG[15, 1], a parallel algebraic multigrid solver. To remedy this problem, we suggest discounting a user based on the interference caused by the other co-running applications. The greater the interference, the greater the discount. The green bars show one such scheme. Because co-location increases machine throughput per unit time, these discounts can be viewed as passing the efficiency savings from co-location back to the end user when their expectation of service is violated.

Although the concept of progressive discounts is simple, the realization of such a policy on real systems poses a number of practical challenges. In particular, a *fair* pricing model of this nature requires precisely quantifying the interference due to shared resource contention. While there has been significant research into predicting cross-core interference, many of the techniques make heavy use of static profiling or have been tailored to specific machines or applications [42, 25, 26]. Even though this work has yielded considerable insight into the problem of shared resource contention, we argue that in practice, it is not practical for precise pricing on a real HPC cluster. In this domain, static profiling and machine- or application-specific approaches are not suitable as jobs may run very shortly after submission and their characterizations may not be known a priori. Although application profiling may enrich the solution space, we note that altering even a single input parameter for an application can vastly change its characteristics. For example, doubling a single array dimension can often radically transform an application's sensitivity to and aggressiveness on the memory subsystem. Thus, an instantaneous and dynamic mechanism is needed to continuously monitor and quantify the interference jobs suffer to drive precise pricing.

In addition to being dynamic and precise, the fundamental pricing mechanism must also be lightweight. The underlying pricing agent has to be mostly invisible to the application and therefore must have a negligible overhead, below the system noise threshold. These objectives lead us to the two key insights of the work – only a software system that uses empirical, online tests is suitable for this problem domain, and such an approach must be agnostic to the underlying software and hardware.

In this paper, we present such a solution: the Persistent Online Precise Pricing Agent (POPPA). POPPA is a lightweight runtime system that utilizes a cyclic, fine-grain, interference sampling mechanism to accurately deduce the interference between co-runners. The key design feature of POPPA is a dynamic contention detection technique we call **shuttering**. For brief periods of execution, POPPA pauses all applications but one and measures how the selected application's performance changes versus running co-located. From the disparity between the application's rate of forward progress made while running co-located versus shuttered, POPPA is able to precisely determine the impact of interference resulting from co-location and use these measurements to drive fair pricing for all users' jobs.

The contributions of this work are as follows:

- We introduce POPPA, a lightweight, workload and machine agnostic runtime system that enables fair pricing for HPC clusters. POPPA functions entirely in software, requires no changes to the system stack in current HPC clusters, and is readily deployable.
- We present the design of *precise shuttering*, a mechanism for the precise online measurement of the performance impact of cross-core interference. Our precise shuttering approach functions dynamically and requires no a priori knowledge or profiling of the applications.
- We present a new pricing model for HPC clusters based on POPPA to provide fair pricing to users.
- We provide a thorough evaluation of POPPA's efficacy and robustness as the central accounting mechanism on HPC clusters with a mix of MPI benchmarks and real workloads.

POPPA predicts co-located application run time with 4% mean absolute error and incurs less than 1% overhead. Using POPPA, we are able to discount the average user by 7.4% and deliver a pricing distribution that closely resembles that of an omniscient oracle.

2. BACKGROUND AND MOTIVATION

In order to better understand why fair pricing is of such importance, we must first explore the current state-of-practice in accounting on supercomputers. We start by examining the accounting and allocation model found in the United States Department of Energy Office of Science INCITE program [12] and the National Science Foundation XSEDE program [11], two of the largest U.S. programs that provide resources to the general HPC research community. Each of these programs facilitates access to a number of large scale computing infrastructures. To successfully obtain an allocation, researchers submit grant proposals and, after reviews, are awarded time on those systems as a finite number of **service units (SUs)**. When a user runs a job on a system, they deplete their bank of SUs at a rate proportional to the length of their programs' execution and the number of compute nodes that they request.

In this model, users need strong guarantees that the value of an SU will not be negatively affected by other users' jobs running on the same computing resources. Similarly, supercomputer administrators care about user satisfaction and are incentivized to provide users with the best possible experience because individual supercomputing centers are awarded funds largely based on the success and popularity of their facilities. Consequently, we observe that **throughout all levels of the funding ladder, fair pricing and accounting are crucial concerns**. Regardless of what mechanisms are implemented to improve supercomputer performance, energy efficiency or fault tolerance, they must not pervert the fairness of the pricing scheme.

2.1 MPI Programming Model

Most large scale scientific applications utilize the Message Passing Interface (MPI) as the core abstraction to facilitate workload distribution across a cluster. Two main characteristics of MPI programs are as follows:

1) **Single Program Multiple Data (SPMD)**: MPI processes execute the same static program binary and use unique identifiers called ranks to dictate communication patterns as well as which blocks of code get executed by different processes. While this allows for a large amount of potential

diversity between processes, in practice most MPI programs are Single Program Multiple Data (SPMD): all processes execute the same core algorithm on different data. Thus within an MPI program, all the processes have high similarity, e.g., they all compete for the same resources.

2) **Tightly coupled communication synchronization**: The vast majority of MPI programs exhibit tightly coupled communication synchronization. Because of this tight synchronization, processes must execute in relative lock-step. If a process reaches an explicit or implicit barrier before the other necessary parties, it must wait until all others make similar progress before proceeding.

2.2 Co-location of MPI programs

When we reason about the nature of MPI programs, it quickly becomes evident that executing a single MPI program across a private set of compute nodes is an inefficient use of system resources. The *homogeneity between MPI processes* and the fact that *they are tightly coupled* mean that many processes will execute the same program regions with high concurrency. When this happens, there is high risk for resource contention and performance degradation – homogeneous processes have high propensity to evict one another's data in the shared last level cache (LLC), contend for the memory controller, saturate off-chip bandwidth to main memory, and cause a backlog of messages for internode communication.

Previous research shows that homogeneous MPI processes can degrade one another's performance by more than 2x [20, 38]. In addition, these works show that introducing heterogeneity in workloads by co-locating multiple MPI programs on disjoint cores can drastically improve performance and energy efficiency. In fact, both studies find that aggregate throughput increases by 12 to 23% on average over the current state of practice, and [20] shows that system energy efficiency increases by 11 to 22%.

In conclusion, given the high cost of large supercomputers and the great performance and efficiency benefit of co-location, it is essential that we provide fair pricing mechanisms to make co-location practical.

3. POPPA OVERVIEW

In this section, we present the overview of the Persistent Online Precise Pricing Agent (POPPA) framework. Our primary design objective for POPPA is to provide accurate performance interference estimates for parallel applications with negligible overhead. As shown in Figure 2, POPPA consists of a main monitoring agent called the Controller and a series of Execution Managers.

Execution Manager: Each Execution Manager is responsible for launching and overseeing the entire execution of a parallel application on a given machine. The Execution Managers read from the central job queue and select the next job to run according to the job priority and its resource needs. An Execution Manager launches the selected job and attaches a performance monitoring context (PMC) to the job. The PMC monitors the job performance by reading and evaluating appropriate hardware performance counters. During execution, the Execution Manager updates and reports the current status and performance data of the job to the Controller.

Controller: The Controller is the main component of POPPA. Its principle responsibility is to conduct **shuttering**, a mechanism to measure and quantify the performance interference among the co-running applications. In essence, the Controller periodically pauses each application but one for a very short period and monitors the performance impact on the lone running application. To measure this impact, the Controller probes the PMCs of each active job to acquire the performance data and logs it. We present more details of the shuttering mechanism including our algorithms and policies in Section 5 and evaluate its accuracy and overhead in Section 8.

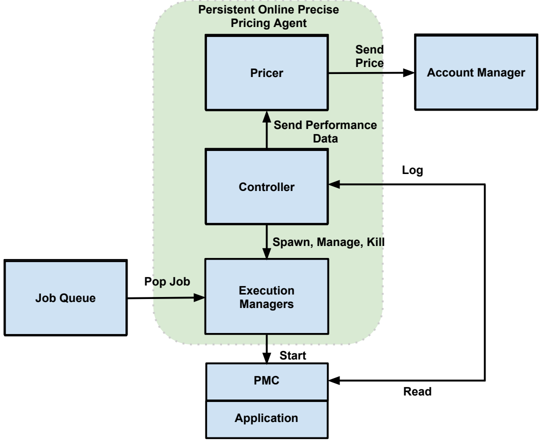


Figure 2: Interaction between POPPA components and other entities

Figure 2 presents how POPPA can be used for pricing. After execution of a job has completed, the Pricer thread analyzes the raw performance data logged by the Controller and quantifies the performance interference and degradation. More details of the analysis and pricing are presented in Sections 4 and 6. Based on the quantification, the Pricer produces the price to be charged and propagates it to the Account Manager, which then deducts the price from the user's bank of SUs.

4. PRICING MODEL

In this section, we discuss the key issues related to pricing and accounting on current supercomputers and extend those notions to a supercomputer with job co-locations.

4.1 Pricing Without Co-location

For purposes of this discussion, assume that a user wants to run a job i on a supercomputer and that P_i denotes the price that the user is charged for running i.

In present day systems, P_i is given by Equation 1, where L is a rate constant in terms of service units per core per time quanta, C_i is the number of cores that a job uses in whole compute node increments, and T_i is the run time of the program.

$$P_i = L * C_i * T_i \tag{1}$$

From this equation, we can see that the price variable P_i is linearly proportional to both the cores variable C_i and the time variable T_i.

4.2 Pricing With Co-location

In this section, we propose how one could modify the existing pricing model to more fairly price applications when co-locations are present. In particular, if we have a job i that is co-located with a set of jobs J, we want a formula that will produce a reasonable price $P_i^{co(J)}$, which takes into account the net interference from all applications in J. To this end, we replace L with a rate function F, yielding Equation 2, where $F : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. T_i^{solo} is the run time when the job i gets all compute nodes to itself and $T_i^{co(J)}$ is the run time of the job i when i is co-located with the set J of other jobs.

$$P_i^{co(J)} = F(T_i^{solo}, T_i^{co(J)}) * C_i * T_i^{solo} \qquad (2)$$

Ideally, F is monotonically non-increasing so that the more degradation an application suffers from co-location, the more the user is discounted. For the purposes of this paper, we assume utility is proportional to 1 minus the rational degradation. Therefore if we equate utility to fairness, then we select F such that users are discounted at a rate proportional to the degradation that each of their jobs experiences due to contention from co-runners. Thus if $D_i^{co(J)}$ is the degradation, then we want $P_i^{co(J)} = (1 - D_i^{co(J)}) * P_i^{solo}$. Consequently we define F as follows:

$$F(T_i^{solo}, T_i^{co(J)}) = L * \frac{T_i^{solo}}{T_i^{co(J)}} = L * (1 - D_i^{co(J)}) \qquad (3)$$

By substituting Equation 3 into Equation 2 we see that we achieve the specific pricing model shown in Equation 4.

$$P_i^{co(J)} = L * \frac{T_i^{solo}}{T_i^{co(J)}} * C_i * T_i^{solo} \qquad (4)$$

While Equation 4 is good for the user, we acknowledge that it is an idealistic model. Its simplicity makes it easy for end users to understand; however, we note other factors such as resource manager queue wait times, job priority, workload composition, the ratio of each shared resource a job consumes, machine architecture, and scheduling policy, i.e. capability versus capacity are also important factors when determining a fair price. Thus supercomputing facilities will have to decide what F makes sense for each of their systems.

5. PRECISE SHUTTER MECHANISM

As previously mentioned, POPPA's chief design objective is to produce fair prices with high precision, low overhead, and without the need for a priori knowledge. To achieve these goals we have designed *precise shuttering*, an online co-runner interference masking approach. Essentially, the precise shuttering mechanism functions by alternating an application's execution environment between one where co-runners are executing and another where they are effectively absent.

Figure 3 shows shuttering in action on two applications A and B that are co-located. The shuttering algorithm alternates between execution regions where A and B co-execute, A executes while B sleeps, A and B co-execute, and B executes while A sleeps. We repeat this pattern throughout the execution of the programs.

To gain insight from shuttering, we must measure the performance of each application before, during, and after shutter regions. During each shutter of duration $\mathcal{S}$, we leverage hardware performance monitors via `libpfm4` [7, 27] to measure the instructions per cycle of the sole non-sleeping application. To infer the degradation due to co-runners, we also measure the instructions per cycle (IPC) of all active applications $\mathcal{S}$ microseconds before the shutter and $\mathcal{S}$ microseconds directly after it.

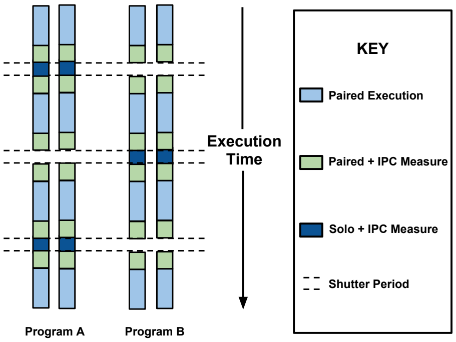


Figure 3: Shown here is shuttering in action on two separate jobs. During a shutter, one job executes while all others sleep.

Since we are primarily concerned by how performance changes with the presence or absence of contention, we only need to monitor the performance during small windows around shutters. We also perform each shutter infrequently to minimize the perturbation of application execution and parameterize the rate of shutter samples to control POPPA's overhead. As we show in this work, frequent shutters are not required to produce an accurate predictive model.

Algorithm 1 *Measure(i, S, K)*

```
Initialize array $perfValue$ of length $|A[i]|$
for $k = 0$ to $K - 1$ do
    for each thread $t$ that is part of $A[i]$ do
        $perfValue[t] = ReadCounters(t)$
    end for
    Sleep for $S$ $\mu s$
    for each thread $t$ that is part of $A[i]$ do
        $perfDict[t].append(ReadCounters(t)\text{-}perfValue[t])$
    end for
end for
```

Algorithm 2 *Shutter_Core(j, S, K)*

```
for $i = 0$ to $|A| - 1$, where $i \neq j$ do
    for each thread $t$ that is part of $A[i]$ do
        Pause $t$
    end for
end for
Measure(j, S, K)
for $i = 0$ to $|A| - 1$, where $i \neq j$ do
    for each thread $t$ that is part of $A[i]$ do
        Resume $t$
        perfDict[t].append(THREAD_ASLEEP)
    end for
end for
```

5.1 Algorithms

In this section, we present the logic of the shutter mechanism, whose core parts are shown in Algorithms 1, 2 and 3. Below we define a list of common data structures and constants used by the algorithms:

- A, an array of co-located applications
- *perfDict*, a lookup table that stores the measured IPC values of each application

Algorithm 3 *POPPA_Core*

```
1: $j = 0$
2: while true do
3:     for $i = 0$ to $|A| - 1$ in parallel do
4:         Measure(i, S, K)
5:     end for
6:     Shutter_Core(j, S, K)
7:     for $i = 0$ to $|A| - 1$ in parallel do
8:         Measure(i, S, K)
9:     end for
10:    $j = (j + 1) \mod |A|$
11:    Sleep $P\mu s$
12: end while
```

- K, the number of IPC measurements to make in a row in a specific region[1]
- S, the length of the each measurement in μs
- P, the length of time between groups of measurements, i.e. the normal execution period, in μs
- $\mathcal{S}$, the length of a shutter, approximately $K * S$

The core routine is Algorithm 3. At each iteration, we first measure the IPC of each application while co-located (lines 3-5). We then shutter application j by calling $Shutter\_Core$ (line 6), which subsequently calls $Measure$ to measure the IPC while j is running alone. After that, we measure the IPC of all applications and increment j (lines 7-10). Then the shutter component of POPPA goes to sleep for $P\mu s$ of normal execution (line 11). Since POPPA is persistent, this process repeats continually as applications end and new applications enter the application pool.

5.2 Tuning the Shutter Mechanism

The shutter implementation presents a number of challenges. In particular, selecting the correct granularity to shutter at is key to accurately quantifying interference without noticeably adding to it. The first parameter is the gap between shutters P. As P is decreased, the amount of time that POPPA is active increases, consequently also increasing overhead. Since utilization in supercomputers is often above 95%, we assume that each core has an application thread assigned to it. Due to this fact, POPPA must time slice with application threads. If POPPA is active for x% of a single core's execution time, then assuming POPPA threads do not migrate, one of the co-running applications is likely to suffer at least an x% hit to performance due to synchronization between processes.

Since the POPPA runtime inevitably has overhead, we experimented with conducting round-robin migration of the POPPA threads to distribute the performance impact of time slicing across all application threads; however, we determined that a better solution was to select values for K, P and S that make POPPA's CPU utilization very low, as migration is not guaranteed to be fine-grain enough to mitigate the effect of time slicing.

Another important parameter is $\mathcal{S}$ the duration of a shutter. In our implementation, this quantity is equal to the base cost of doing a shutter on 8 MPI processes, approximately 120 to 200μs (see Figure 4 in Section 8.1), plus $K * S$, where $K * S$ is the product of the number of consecutive measurements and the length of each such measurement. During a shutter, the paused application makes no progress, thus keeping shutter duration very short relative to P is a primary concern.

An unexpected find relating to the shutter mechanism is that in certain cases, POPPA actually slightly improves the performance of co-located applications. During shutters, applications that sleep sacrifice a small amount of forward progress and the lone runner receives a performance boost from reduced contention. When the net performance boost from running in isolation offsets the net performance loss from sleeping, applications speed up relative to the baseline co-schedule performance. For pairs of two applications, speedup occurs when a co-schedule increases one application's run time by more than 2x relative to running with half the cores idle per socket. This phenomenon is demonstrated empirically in Section 8.2.

6. ESTIMATING DEGRADATION

In this section, we present our method for linking the raw data that POPPA produces to the actual prices we charge.

6.1 Idealized Model for Degradation

Our pricing model assumes that for an application i, we know the degradation $D_i^{co(J)}$ that i suffers as a result of colocation with a set J of applications. In our pricing model discussion, we formulated $1 - D_i^{co(J)}$ as $\frac{T_i^{solo}}{T_i^{co(J)}}$. While this gives us a precise way to calculate degradation, POPPA cannot directly measure T_i^{solo}. Thus, we modify the formulation such that it is amenable to the IPC data that POPPA produces.

On modern chip multiprocessors, if we are given an execution time in seconds, we can convert this to a value in clock cycles. Thus if we know the clock ticks per second, we can write the performance of i normalized to running alone as the ratio of clock cycles C_i^{solo} and $C_i^{co(J)}$(see below).

$$Perf_i^{norm} = 1 - D_i^{co(J)} = \frac{C_i^{solo}}{C_i^{co(J)}} \tag{5}$$

Additionally, if we assume i to be a truly serial program, then it is the case that i's dynamic instructions I_i do not change. Thus $I_i^{solo} = I_i^{co(J)}$, and consequently we can transform Equation 5 into a ratio of IPCs by multiplying by $\frac{I_i^{co(J)}}{I_i^{solo}}$, yielding the following:

$$Perf_i^{norm} = \frac{IPC_i^{co(J)}}{IPC_i^{solo}} \tag{6}$$

6.2 Known Challenges with Parallel Programs

For parallel programs, however, it turns out that Equation 6 is often imprecise. Many parallel programs contain mutexes, semaphores, and other locking mechanisms to enforce program correctness by preventing data races. When a load imbalance occurs, that is, one parallel process advances faster than its siblings, these locking mechanisms can distort both dynamic instruction count and CPU clock cycles.

With MPI, this issue is quite prevalent. If a communication routine is implemented as blocking, then it is common practice to have the thread that initiated the routine to poll for a certain number of cycles and then sleep. During this polling period, the thread executes a while loop where it continually tests whether the communication operation has

[1] We fix $K = 1$ for experiments and analyses in Section 8.

completed. If the thread fails to finish the communication operation within a certain interval, it is put to sleep and signaled to wake up when the operation has completed. Because contention and background noise on the system can cause this polling period to change in duration, the number of dynamic instructions attributed to these communication regions is variable. With MVAPICH2, the MPI-2 implementation, the maximum polling period can be adjusted [52]. While we were tempted to disable polling, we knew that doing so would be disadvantageous. In particular, polling greatly increases individual application performance because the blocking thread avoids the performance hit associated with going to sleep and waking back up, as it can proceed as soon as communication has finished. Thus, we decided to keep the parameters that maximized performance even though it made precise prediction more challenging.

6.3 Filtering

Even though Equation 6 is imprecise in the presence of variable execution, we find that in practice, it is still sufficient for producing reasonable degradation estimates. We also assume that the average over the N IPC samples that we collect is roughly equivalent to the actual average IPC during shutters (IPC_i^{solo}) and during normal paired execution (IPC_i^{co}). These assumptions are presented below in Equations 7 and 8.

$$Perf_i^{norm} \approx \frac{IPC_i^{co(J)}}{IPC_i^{solo}} \tag{7}$$

$$IPC_i^{solo} \approx \frac{\sum_{j=0}^{N_i^{solo}} IPC_{i,j}^{solo}}{N_i^{solo}} \text{ and } IPC_i^{co} \approx \frac{\sum_{j=0}^{N_i^{co}} IPC_{i,j}^{co}}{N_i^{co}} \tag{8}$$

POPPA gives us data in the form of a stream of blocks of IPC measurements, each consisting of K IPC measurements just before a shutter, K measurements during a shutter, and K afterward. We denote this stream of blocks as B and the lth such block as B_l; within each block B_l, the K IPC values in B_l before the shutter are denoted as IPC_l^{before}, the K IPC values during a shutter as IPC_l^{during}, and the K IPC values after a shutter as IPC_l^{after}. Thus $B_l = (IPC_l^{before}, IPC_l^{during}, IPC_l^{after})$. We denote the arithmetic means of each of these values as $\overline{IPC_l^{before}}$, $\overline{IPC_l^{during}}$ and $\overline{IPC_l^{after}}$. Using this notation, we present the filtering algorithm (Algorithm 4) that allows us to increase the precision of the performance estimate.

Algorithm 4 *Filtered Prediction(IPC Tuples B)*

```
Initialize $IPC^{co}$ and $IPC^{solo}$ to 0
for each $(\overline{IPC_l^{before}}, \overline{IPC_l^{during}}, \overline{IPC_l^{after}})$ in $B$ do
    if $|\overline{IPC_l^{before}} - \overline{IPC_l^{after}}| < \delta$ and $\overline{IPC_l^{before}} < \overline{IPC_l^{during}}$ and $\overline{IPC_l^{after}} < \overline{IPC_l^{during}}$ then
        $IPC^{co} \overset{+}{=} 0.5(\overline{IPC_l^{before}} + \overline{IPC_l^{after}})$
        $IPC^{solo} \overset{+}{=} \overline{IPC_{during}}$
    end if
end for
Return $(\frac{IPC^{solo} - IPC^{co}}{IPC^{solo}})$
```

Algorithm 4 aims to reduce noise from sampling IPC. It removes groups of IPC values where the IPC during a shutter is not greater than the IPC directly before and after. Since a shutter can only relieve shared resource contention, the IPC during a shutter should always exceed the IPC before and after a shutter if all measurements occur during the same computational phase. The second mechanism, which states that the absolute difference in IPC before and after cannot exceed δ works to ensure that clusters that cross phase boundaries are removed. We empirically determined $\delta = 0.05$ to be a reasonable value.

7. EXPERIMENTAL SETUP

This section describes our methodology. We ran our experiments on the Gordon Supercomputer [32, 49]. Each node is dual-socket. For each socket, there is an 8-core Intel EM64T Xeon E5 (Sandy Bridge) processor. Simultaneous multithreading is disabled [61]. The CPU frequency is 2.6Ghz, and each core has private 32KB instruction and data L1 caches, a private 256KB L2 cache, and each socket has 20MB of L3. There are 64GB of DRAM. Compute nodes run CentOS linux with kernel version 2.6.32. The interconnect is QDR InfiniBand with 8GB/s of bidirectional bandwidth, and the topology is a 3D torus of switches [10, 57]. Our applications and benchmarks are shown in the table that follows. These benchmarks and applications encompass a wide variety of scientific domains such as subatomic particle physics [5], plasma physics [41], molecular dynamics [3], ocean modeling [2], computational fluid dynamics [6, 8], shock hydrodynamics [36], finite element methods [4] along with various other numerical methods that are of high interest to the HPC community. We also note that GTC and MILC, in particular, use a substantial number of dedicated allocation hours on many leadership class machines.

Benchmarks, *Miniapps* and **Applications**
Swim [9], ADVECT3D [51], pcubed [39]
NAS Parallel Benchmarks: CG, FT, LU, MG [14, 47]
Lulesh [36], *MiniGhost* [4], *MiniFE* [4], *NekBone* [6, 8]
GTC [41], **LAMMPS** [3], **MILC** [5], **POP** [2]

We compile GTC, LAMMPS, MILC, POP, CG, FT, LU and MG with GNU compilers version 4.7 and MVAPICH2 version 1.7. LULESH, MiniGhost, MiniFE, and NekBone are compiled with PGI compilers version 11.9 and OpenMPI version 1.6.

In our experiments, we co-locate two 8 process MPI applications together on the same set of sockets. Each socket has half its cores run one application and the other half run the other. Applications co-run together for a minimum of 5 iterations of both applications. As soon as one application ends, we immediately restart it. Data collection stops once both applications have completed 5 iterations. For the shutter mechanism, we fix $K = 1$ and $P = 200$ms.

8. EVALUATION

In this section, we evaluate the accuracy, overhead, and the pricing fairness of POPPA.

8.1 Quantifying POPPA's Base Overhead

In this section, we quantify the minimum time to execute components within the main loop of the POPPA daemon. The main loop consists of the three core operations of Algorithm 3 – measuring the IPC of the application just prior to the shutter, issuing the shutter and measuring the IPC of

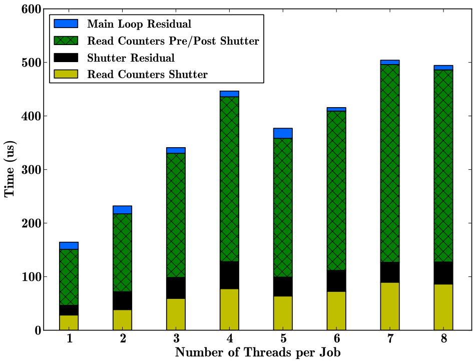


Figure 4: Breakdown of base overhead to execute a single iteration of POPPA's core algorithm, where reading PMC values dominates total time

the application during that window, and measuring the IPC of the application immediately following the shutter.

For these experiments, we co-locate two MPI benchmarks, an auto-generated loop from the `pcubed` benchmark suite and a busy loop, called the NULL co-runner, that runs for the duration of the `pcubed` loop. In POPPA, we set all of the sleep parameters to 0, so we can measure the minimum execution time for all subcomponents of the loop. During each iteration of the main loop, we measure its total execution time, the time to measure the IPC both before and after the shutter, the total execution time of the shutter, the time to send the `SIGSTOP` and `SIGCONT` signals, and the time to make the IPC measurements during the shutter.

Figure 4 presents the results. On the x-axis we vary the number of threads in each job. So 4 corresponds to four `pcubed` tasks bound to cores 0, 2, 4, and 6 and four `busy loop` tasks bound to cores 1, 3, 5, and 7. The y-axis shows the total time in μs to execute the main loop. When studying this figure, several interesting trends emerge. Not surprisingly, adding more threads increases the minimum loop execution time. Execution time is dominated by IPC measurement in the form of calls to `libpfm`, particularly those outside the shutter region. In fact, we spend about 4x as much time measuring the IPC outside of shutter regions compared to within them. This difference in overhead results from 1) we only measure active threads within a shutter, which is an optimization decision that we made, so the overhead to read the performance counters doubles outside of a shutter, and 2) we make two sets of IPC measurements outside of a shutter (before and after) versus a single set of measurements during one.

We see that the mean time to shutter does not exceed 130μs and the mean time to execute the main loop does not exceed 500μs. Thus, our mechanism is fine grained enough to measure the IPC at sub-millisecond intervals for thread counts that are representative of contemporary multi-socket systems.

In addition to the minimum delays incurred by shuttering, we quantify the effect of enlarging the amount of time spent in a shutter. For this experiment, we fix the sleep time at the end of the main loop, P (see Section 5.1), to 200,000μs and increase the shutter duration, $\mathcal{S}$ (see Section 5.1), multiplicatively by factors of 2 from 200μs to 409,600μs. We separately co-run each of the NAS Parallel Benchmarks (NPB) with the busy loop NULL. Since NULL generates no interference, any dilation in run time is a direct result of increasing the shutter window.

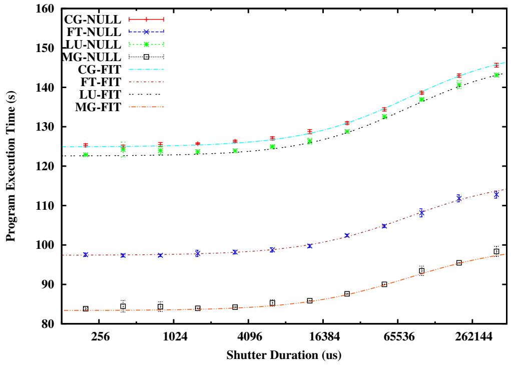


Figure 5: The relative overhead of expanding the duration of a shutter, where points correspond to measurements and lines correspond to instantiations of the model

Figure 5 presents the results. All four benchmarks exhibit a similar trend. When $\mathcal{S}$ is small relative to P, the overhead is small, but as the ratio $\mathcal{S} : P$ increases, so does the overhead. However, the overhead begins to flatten out as $\mathcal{S}$ approaches and exceeds the value of P.

We need to formulate an analytical model for the overhead that a pricing shutter creates for an arbitrary co-located pool of n jobs. To do so, we examine the overhead from n consecutive shutters. Over the course of n shutters, each job will run in isolation once and sleep $n-1$ times while a single other job enjoys the privilege. Each such shutter has duration $\mathcal{S}$. Thus each job will sleep for $(n-1)*\mathcal{S}$ seconds.

The total time for n iterations of the main loop of the daemon is also important for the analysis. Measuring the IPC before, during and after a shutter is $3\mathcal{S}$, as each takes $\mathcal{S}$ time. After this, the daemon sleeps P seconds. This pattern is cyclic, so the combined time is $n*(3\mathcal{S}+P)$. Equation 9 shows ratio of sleep time to total time.

$$Z(\mathcal{S},P) = \frac{\text{sleep time}}{\text{total time}} = \frac{(n-1)*\mathcal{S}}{n*(3\mathcal{S}+P)} \tag{9}$$

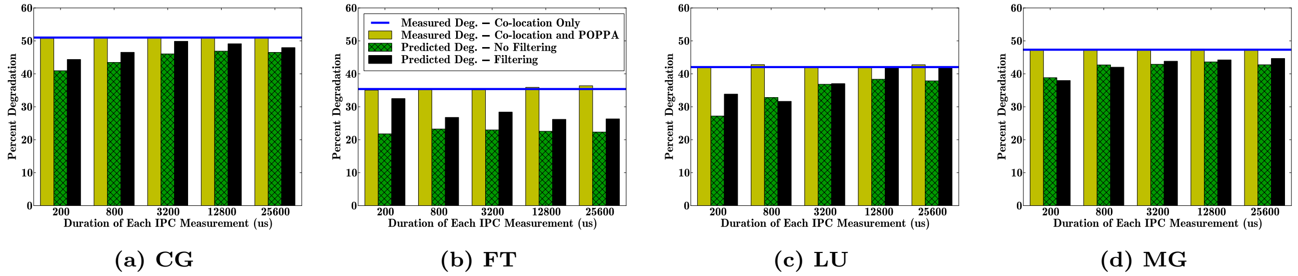
The model for the execution time of the jobs in Figure 5 is shown below:

$$T(\mathcal{S},P) = T_i * \frac{1}{1-Z(\mathcal{S},P)} = T_i * \frac{n*(3\mathcal{S}+P)}{2n\mathcal{S}+nP+\mathcal{S}} \tag{10}$$

Here T_i is the run time of application i when co-located with the NULL co-runner. When we examine the model fit to the data in Figure 5, we observe that `CG-FIT`, `FT-FIT`, `LU-FIT`, `MG-FIT` almost exactly predict the actual overhead of the shutter for all $\mathcal{S}$ in $\{100*2^k\mu s|1 <= k <= 12\}$ and a fixed P of 200ms. This model incorporates $\mathcal{S}$, P, and T; if we know any two of these quantities, we can solve for the third. Thus administrators can decide on a system by system basis what is exactly an acceptable amount of degradation due to the pricing shutter and choose values of $\mathcal{S}$ and P accordingly.

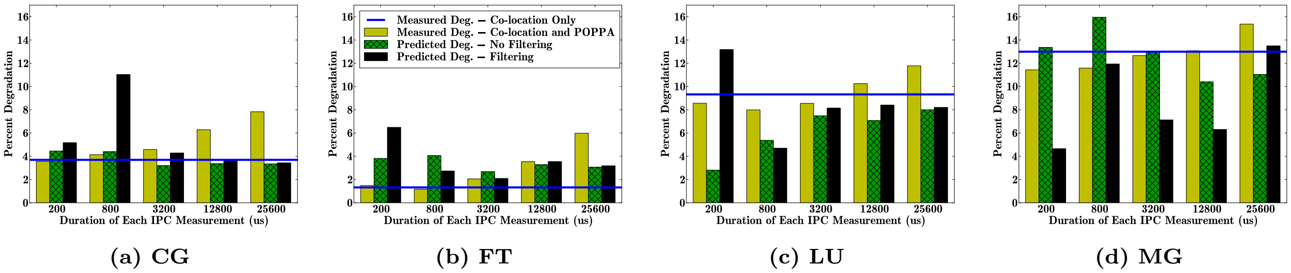
8.2 Determining the Sampling Rate

In this section, we evaluate the precision and overhead of the POPPA daemon for different shutter lengths ($\mathcal{S}$ values) while keeping P fixed to 200ms. We saw in the previous section, that the overhead due to the shuttering mechanism has an analytical upper bound given by Equation 10. Using this equation, we selected values of $\mathcal{S}$ with less than 5% overhead: 200, 400, 800, 1600, 3200, 6400, 12800, and 25600μs.



(a) CG (b) FT (c) LU (d) MG

Figure 6: Effect of shutter duration on accuracy and overhead for each NPB co-run with ADVECT3D-256



(a) CG (b) FT (c) LU (d) MG

Figure 7: Effect of shutter duration on accuracy and overhead for each NPB co-run with Swim-150

We ran two sets of pairwise experiments. In the first, we co-located the NPBs with a contentious co-runner (ADVECT3D with a grid size of 256^3), and in the other we co-scheduled the NPBs with a moderately contentious co-runner (Swim with a grid dimension of 150^3). Figures 6a, 6b, 6c, and 6d show the performance prediction accuracy of the POPPA daemon for CG, FT, LU, and MG when they are co-located with ADVECT3D. Both the accuracies of the unfiltered and filtered predictors are shown. For clarity, we opt not to present the results for 400, 1600 and 6400μs.

In this set of experiments, we are able to very accurately predict the contention with negligible overhead. Filtering improves prediction performance. Our predictors have the largest error for FT. $\mathcal{S} = 200\mu s$ gives the highest accuracy, but as $\mathcal{S}$ increases, so does the error. This error results from FT's very fine grain phases, which coarser granularity shutters have trouble capturing.

Figures 7a, 7b, 7c, and 7d show the prediction accuracy for the NPBs paired with Swim. Again, our prediction accuracy is very precise. In this case, we note that the filtered prediction is sometimes overly zealous when predicting contention. However, this result is unsurprising given that filtering removes clusters of IPC measurements where the IPC measured during a shutter does not exceed the IPC directly before and after.

A contrasting finding between the experiments with ADVECT3D and Swim concerns daemon overhead as a function of $\mathcal{S}$. In the experiments with ADVECT3D, overhead is flat regardless of $\mathcal{S}$ whereas it sharply increases with Swim. This divergence is caused by the fact that ADVECT3D is configured to be contentious whereas Swim is not. During a shutter, the lone running application receives a respite from the contention generated by the other application. In the case of the NPBs with ADVECT3D, this causes each NPB to speed up by approximately 2x, which offsets the lost throughput from sleeping during alternate shutters. By contrast, Swim degrades each NPB by at most 15%, so the time spent sleeping cannot be masked.

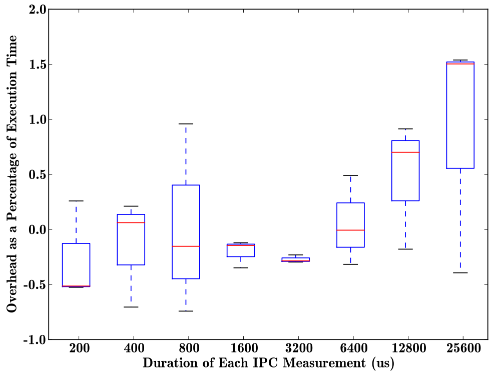


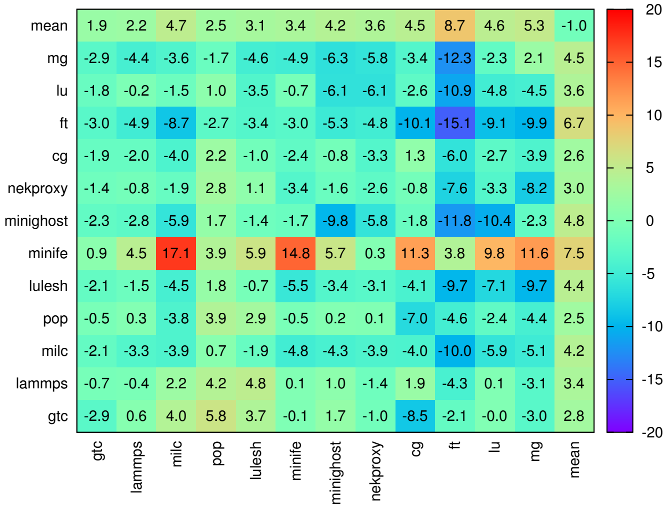
Figure 8: Overhead of POPPA on NAS benchmarks co-located with ADVECT-256

These experiments show that the shutter duration $\mathcal{S}$ is largely irrelevant for accuracy. Thus when selecting $\mathcal{S}$, it makes sense to select a value that induces minimal overhead and run time variation. Figures 8 and 9 present both the daemon's overhead and its distribution for the surveyed values of $\mathcal{S}$. In Figure 8, regardless of the value of $\mathcal{S}$, overhead due to the pricing shutter never exceeds 2%. However, in Figure 9, this value exceeds 4%, which is clearly too costly. $\mathcal{S} = 3200\mu s$ delivers an overhead of less than 1% and with the smallest variation. For this reason, we use $\mathcal{S} = 3200\mu s$ for the remainder of our experiments.

8.3 Pairwise Evaluation

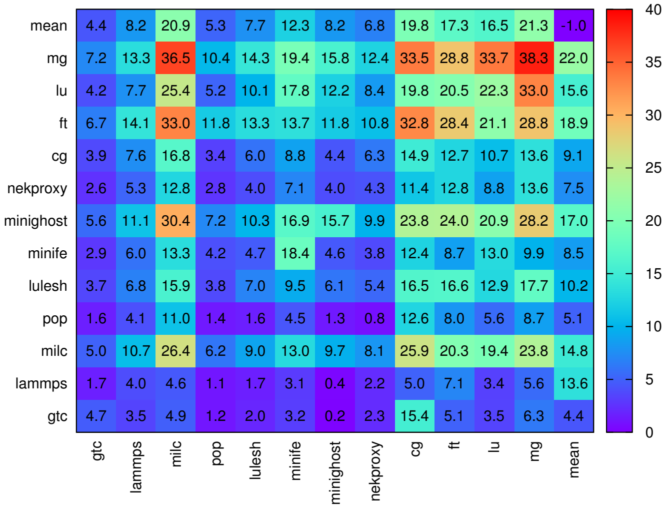
In this section, we evaluate the precision of POPPA on pairwise co-locations. Since our filtered prediction was better in aggregate in our previous experiments, we apply that prediction mechanism rather than the simple one. We run co-schedules of all possible combinations of our 12 benchmarks and real applications.

Figure 10 shows the accuracy of our filtered predictor at quantifying degradation. The x-axis lists the names of the benchmarks, and the y-axis lists the co-runners. Individ-



	gtc	lammps	milc	pop	lulesh	minife	minighost	nekproxy	cg	ft	lu	mg	mean
mean	1.9	2.2	4.7	2.5	3.1	3.4	4.2	3.6	4.5	8.7	4.6	5.3	-1.0
mg	-2.9	-4.4	-3.6	-1.7	-4.6	-4.9	-6.3	-5.8	-3.4	-12.3	-2.3	2.1	4.5
lu	-1.8	-0.2	-1.5	1.0	-3.5	-0.7	-6.1	-6.1	-2.6	-10.9	-4.8	-4.5	3.6
ft	-3.0	-4.9	-8.7	-2.7	-3.4	-3.0	-5.3	-4.8	-10.1	-15.1	-9.1	-9.9	6.7
cg	-1.9	-2.0	-4.0	2.2	-1.0	-2.4	-0.8	-3.3	1.3	-6.0	-2.7	-3.9	2.6
nekproxy	-1.4	-0.8	-1.9	2.8	1.1	-3.4	-1.6	-2.6	-0.8	-7.6	-3.3	-8.2	3.0
minighost	-2.3	-2.8	-5.9	1.7	-1.4	-1.7	-9.8	-5.8	-1.8	-11.8	-10.4	-2.3	4.8
minife	0.9	4.5	17.1	3.9	5.9	14.8	5.7	0.3	11.3	3.8	9.8	11.6	7.5
lulesh	-2.1	-1.5	-4.5	1.8	-0.7	-5.5	-3.4	-3.1	-4.1	-9.7	-7.1	-9.7	4.4
pop	-0.5	0.3	-3.8	3.9	2.9	-0.5	0.2	0.1	-7.0	-4.6	-2.4	-4.4	2.5
milc	-2.1	-3.3	-3.9	0.7	-1.9	-4.8	-4.3	-3.9	-4.0	-10.0	-5.9	-5.1	4.2
lammps	-0.7	-0.4	2.2	4.2	4.8	0.1	1.0	-1.4	1.9	-4.3	0.1	-3.1	3.4
gtc	-2.9	0.6	4.0	5.8	3.7	-0.1	1.7	-1.0	-8.5	-2.1	-0.0	-3.0	2.8

Figure 10: Run time prediction accuracy (%) for jobs on the x-axis co-located with jobs on the y-axis



	gtc	lammps	milc	pop	lulesh	minife	minighost	nekproxy	cg	ft	lu	mg	mean
mean	4.4	8.2	20.9	5.3	7.7	12.3	8.2	6.8	19.8	17.3	16.5	21.3	-1.0
mg	7.2	13.3	36.5	10.4	14.3	19.4	15.8	12.4	33.5	28.8	33.7	38.3	22.0
lu	4.2	7.7	25.4	5.2	10.1	17.8	12.2	8.4	19.8	20.5	22.3	33.0	15.6
ft	6.7	14.1	33.0	11.8	13.3	13.7	11.8	10.8	32.8	28.4	21.1	28.8	18.9
cg	3.9	7.6	16.8	3.4	6.0	8.8	4.4	6.3	14.9	12.7	10.7	13.6	9.1
nekproxy	2.6	5.3	12.8	2.8	4.0	7.1	4.0	4.3	11.4	12.8	8.8	13.6	7.5
minighost	5.6	11.1	30.4	7.2	10.3	16.9	15.7	9.9	23.8	24.0	20.9	28.2	17.0
minife	2.9	6.0	13.3	4.2	4.7	18.4	4.6	3.8	12.4	8.7	13.0	9.9	8.5
lulesh	3.7	6.8	15.9	3.8	7.0	9.5	6.1	5.4	16.5	16.6	12.9	17.7	10.2
pop	1.6	4.1	11.0	1.4	1.6	4.5	1.3	0.8	12.6	8.0	5.6	8.7	5.1
milc	5.0	10.7	26.4	6.2	9.0	13.0	9.7	8.1	25.9	20.3	19.4	23.8	14.8
lammps	1.7	4.0	4.6	1.1	1.7	3.1	0.4	2.2	5.0	7.1	3.4	5.6	13.6
gtc	4.7	3.5	4.9	1.2	2.0	3.2	0.2	2.3	15.4	5.1	3.5	6.3	4.4

Figure 11: Performance degradation (%) for jobs on the x-axis co-located with jobs on the y-axis

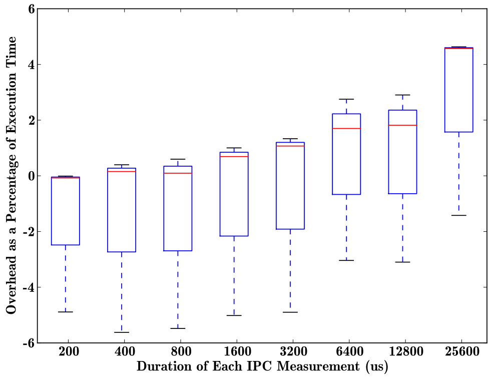


Figure 9: Overhead of POPPA on NAS benchmarks co-located with Swim-150

ual cells present the percentage difference in predicted run time versus actual, where negative values represent underprediction and positive values represent overprediction. The top row "mean" presents the mean absolute error across the apps, and the right most column "mean" presents the mean absolute error that an application creates in the prediction accuracy for the other codes.

Figure 11 presents the degradation of each application as a percentage of run time relative to running with the NULL co-runner, i.e half the cores vacant on each socket. The top row presents the mean degradation of each scientific code on the x-axis and the right most column presents the mean degradation each application on the y-axis causes to its co-runners.

If we study Figures 10 and 11 in concert, a number of interesting trends emerge. POPPA does well at quantifying degradation for all pairings consisting exclusively of our real applications, GTC, LAMMPS, MILC, and POP. Our mean absolute error is 2.5% and absolute error never exceeds 5.8%. We accurately characterize both ends of the spectrum. We predict high degradation for MILC paired with itself and we neither significantly underpredict or overpredict for pairings with low mutual contention such as GTC-LAMMPS and LAMMPS-POP. For pairings of real apps with benchmarks, the prediction accuracy is generally quite good except for when MILC is co-located with MiniFE and FT.

For our proxy apps LULESH, MiniFE, MiniGhost and NekProxy (NekBone), the results are more mixed. We are able to predict their performance with a mean absolute error of 3.8%. MiniFE is a particularly interesting because in each case we overpredict the degradation for its co-runner (mean of 7.5%). This overprediction is an artifact of the filtering algorithm. When we use our unfiltered predictor, we overpredict by at most 1.5% for MiniFE's co-runners. MiniGhost, by contrast causes us to underpredict contention for some of its co-runners.

On the NPBs, our prediction error is slightly higher. If we exclude FT, our mean absolute prediction error is within 5.3%. FT however, poses challenges both for its prediction and applications it is co-located with. In both cases, we underpredict the actual degradation. This underprediction is due to the duration $\mathcal{S}$ of the shutter. If we reexamine Figure 6b, we observe that $\mathcal{S} = 200\mu s$ yields the highest accuracy when FT is co-located with a contentious co-runner. We also observe in Figure 7b that out of the possible values for $\mathcal{S}$, $\mathcal{S} = 3200\mu s$ prognosticates the lowest contention. On the whole, our system is generous and tends towards modestly underpredicting contention. Our mean absolute error across all pairings is 4.0%.

8.4 Pricing Fairness

In this section, we show POPPA's pricing fairness versus the state-of-practice and the oracle. Figure 12 shows the distribution of relative SUs charged for each application using the different pricing schemes. On average, the state-of-practice would charge users 14% more as result of co-locating their jobs. Jobs that degrade more, pay more. POPPA on the other hand discounts users by an average of 7.4%, which is close to the 11.5% discount that the oracle would offer.

When we examine the minimum and maximum relative SUs charged, we also see favorable results for POPPA. The maximum discount given by POPPA is 40.8%, which is close to the oracle's 38.3%. The max normalized price paid by a user using POPPA's counsel is 103.8% of the spread baseline versus the oracle's 99.8%. In the minority of cases where

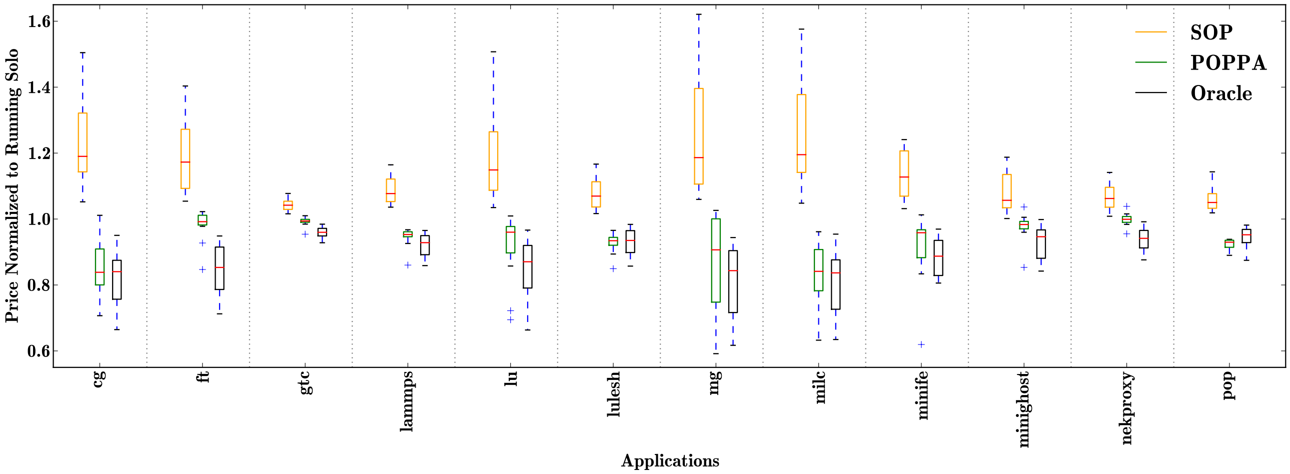


Figure 12: The distribution of prices a user would pay for a given application when using either the state-of-practice (SOP), POPPA, or the maximally fair Oracle

POPPA charges more than the spread baseline (23/144), it is usually smaller than run-to-run variation, with a mean surcharge of 1.3%. In addition, the mean price paid for each application never exceeds 99.2% of the baseline, and thus over time, all users will receive a discount. Contrast this with the state-of-practice, where a user running MILC in the worst case can pay up to 62.1% more and on average would expect to pay 24.9% more as a result of cross-application interference.

If we consider the impact of POPPA's discounts, we find they are entirely tenable. Recall that the job striping study [20] found that co-locating MPI benchmarks and full-scale applications at scale increased mean system throughput by 12 to 23%. Thus discounting users by a mean 7.4% does not inflate the purchasing power of SUs, and so SU allocation need not be changed.

9. RELATED WORK

There are a number of works that investigate pricing or identify pricing as a key issue for large scale grid and cloud infrastructures [13, 63, 48, 54]. Our work differs from these works in that we address the pricing issue in supercomputers with co-locations. To the best of our knowledge, our work is the first to explore this problem space.

Although this work addresses challenges related to fair pricing, it shares similarities with research that addresses identifying and mitigating contention in multicore systems. Early work on simultaneous multi-threading processors investigated co-scheduling of heterogeneous threads [55, 56, 21] as a way to increase throughput by reducing contention.

Cross core contention has also been extensively studied [22, 65, 44, 43]. A mechanism similar to the pricing shutter is explored in [44] but differs in that it is in the commercial data center space and in that it focuses on L3 miss rates with and without the presence of contention.

Another solution to mitigating contention has been cache partitioning both in software and in hardware [46, 58, 53, 23]. Core fusion is an architectural design that helps reduce the cross core contention problem by dynamically combining simpler cores into larger cores [34, 59]. Others have examined using scheduling to mitigate contention [64, 29, 28, 18, 17] and [50, 60] investigate scheduling considerations in mapreduce environments.

There are also studies that evaluate the effectiveness of analytical and statistical models to solve problems related to contention [40, 62, 24, 31]. The computational complexity, heuristics and approximation algorithms for optimal multiprocessor scheduling are explored in [30, 19, 37, 35].

10. CONCLUSION

We have provided a mechanism to enable fair pricing on HPC systems, one of the fundamental roadblocks to enable node sharing on HPC systems. By employing POPPA, we can accurately measure performance degradation across a range of MPI applications. Using this data, we price users in a fashion that approaches the optimal fairness provided by the oracle, and our mean absolute prediction error is 4% across all combinations of 12 application codes.

POPPA is not a definitive solution to the pricing problem but a key part of a more holistic solution. Going forward, the development of additional, light-weight techniques for application introspection will become essential. By harnessing this dynamic information, further optimization opportunities will arise. Through combining these solutions, the road to exascale supercomputers looks bright.

11. ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers for their time and feedback. In addition, they thank Professor Mike Norman of UCSD, Professor Leo Porter of Skidmore College, and Terri Quinn of LLNL. Some of the ideas in this paper were inspired by discussions with the late Allan Snavely. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-635977). This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under award number 62855 "Beyond the Standard Model – Towards an Integrated Modeling Methodology for the Performance and Power"; PNNL lead institution; Program Manager Sonia Sachs. The authors acknowledge National Science Foundation support under CCF-1302682.

12. REFERENCES

[1] Asc sequoia benchmark codes. https://asc.llnl.gov/sequoia/benchmarks/.

[2] Cesm1.0: Parallel ocean program (pop2). http://www.cesm.ucar.edu/models/cesm1.0/pop2/.

[3] Large-scale Atomic/Molecular Massively Parallel Simulator. http://lammps.sandia.gov/.

[4] Mantevo suite. http://www.mantevo.org/.

[5] MIMD Lattice Computation (MILC) Collaboration. http://www.physics.indiana.edu/~sg/milc.html.

[6] Nek5000 project. https://nek5000.mcs.anl.gov/index.php/Main_Page.

[7] perfmon 2: improving performance monitoring on linux. http://perfmon2.sourceforge.net/.

[8] Proxy-apps for thermal hydraulics. https://cesar.mcs.anl.gov/content/software/thermal_hydraulics.

[9] Spec cpu 2000 benchmark suite. www.spec.org/cpu2000.

[10] Gordon user guide. http://www.sdsc.edu/us/resources/gordon/, 2012.

[11] Extreme Science and Engineering Discovery Environment. www.xsede.org, 2013.

[12] Innovative and Novel Computational Impact on Theory and Experiment. http://www.doeleadershipcomputing.org/incite-program/, 2013.

[13] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4), 2010.

[14] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks–summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, New York, NY, USA, 1991. ACM.

[15] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang. Challenges of scaling algebraic multigrid across modern multicore architectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011.

[16] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. www.cse.nd.edu/Reports/2008TR-2008-13.pdf, 2008.

[17] S. Blagodurov and A. Fedorova. Towards the contention aware scheduling in hpc cluster environment. In *Journal of Physics: Conference Series*, volume 385. IOP Publishing, 2012.

[18] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems*, 28, 2010.

[19] J. Blazewicz, J. K. Lenstra, and A. Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1), 1983.

[20] A. D. Breslow, L. Porter, A. Tiwari, M. Laurenzano, L. Carrington, D. M. Tullsen, and A. E. Snavely. The case for colocation of hpc workloads. *Concurrency and Computation: Practice and Experience*, 2013.

[21] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero. Predictable performance in SMT processors. In *1st Conference on Computing Frontiers*, 2004.

[22] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *11th International Symposium on High-Performance Computer Architecture*, 2005.

[23] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st annual international conference on Supercomputing*. ACM, 2007.

[24] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012.

[25] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011.

[26] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Understanding memory contention. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*. IEEE, 2012.

[27] S. Eranian. Perfmon: Linux performance monitoring for ia-64. *Downloadable software with documentation, http://www. hpl. hp. com/research/linux/perfmon*, 2003.

[28] A. Fedorova, M. Seltzer, and M. D. Smith. Cache-fair thread scheduling for multicore processors. *Division of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-17-06*, 2006.

[29] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[30] M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4), 1975.

[31] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011.

[32] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely. Dash: a recipe for a flash-based data intensive supercomputer. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[33] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng. Oversubscription on multicore processors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010.

[34] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2), 2007.

[35] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.

[36] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, et al. Exploring traditional and emerging parallel programming models using a proxy application. *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013), Boston, USA*, 2013.

[37] H. Kasahara and S. Narita. Practical multiprocessor

scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, 33(11), 1984.

[38] M. J. Koop, M. Luo, and D. K. Panda. Reducing network contention with mixed workloads on modern multicore, clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on.* IEEE, 2009.

[39] M. A. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. M. Tikir, and S. Poole. Reducing Energy Usage with Memory and Computation-Aware Dynamic Frequency Scaling. In *Proceedings of the 17th international Euro-Par conference on Parallel processing*, EuroPar'11, Bordeaux, France, 2011.

[40] S.-H. Lim, J.-S. Huh, Y. Kim, G. M. Shipman, and C. R. Das. D-factor: a quantitative model of application slow-down in multi-resource shared systems. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems.* ACM, 2012.

[41] Z. Lin, G. Rewoldt, S. Ethier, T. S. Hahm, W. W. Lee, J. L. V. Lewandowski, Y. Nishimura, and W. X. Wang. Particle-in-cell simulations of electron transport from plasma turbulence: recent progress in gyrokinetic particle simulations of turbulent plasmas. *Journal of Physics: Conference Series*, 16(1), 2005.

[42] 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 *MICRO '11: Proceedings of The 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2011. ACM.

[43] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers.* ACM, 2011.

[44] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization.* ACM, 2010.

[45] R. L. Moore, D. L. Hart, W. Pfeiffer, M. Tatineni, K. Yoshimoto, and W. S. Young. Trestles: a high-productivity hpc system targeted to modest-scale and gateway users. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, New York, NY, USA, 2011. ACM.

[46] F. Mueller. Compiler support for software-based cache partitioning. In *ACM Sigplan Notices*, volume 30. ACM, 1995.

[47] NAS. Nas parallel benchmarks website, http://www.nas.nasa.gov/Resources/Software/npb.html.

[48] D. Niu, C. Feng, and B. Li. Pricing cloud bandwidth reservations under demand uncertainty. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems.* ACM, 2012.

[49] M. Norman and A. Snavely. Accelerating data-intensive science with Gordon and Dash. In *2010 TeraGrid Conference*, 2010.

[50] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley. Performance-driven task co-scheduling for mapreduce environments. In *Network Operations and Management Symposium (NOMS), 2010 IEEE.* IEEE, 2010.

[51] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for.* IEEE, 2010.

[52] MVAPICH Team. Mvapich2 1.8 user guide. `http://mvapich.cse.ohio-state.edu/support/mvapich2-1.8_user_guide.pdf`, 2012.

[53] M. K. Qureshi and Y. 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] B. Sharma, R. K. Thulasiram, P. Thulasiraman, S. K. Garg, and R. Buyya. Pricing cloud compute commodities: a novel financial economic model. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012).* IEEE Computer Society, 2012.

[55] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[56] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30. ACM, 2002.

[57] S. M. Strande, P. Cicotti, R. S. Sinkovits, W. S. Young, R. Wagner, M. Tatineni, E. Hocks, A. Snavely, and M. Norman. Gordon: design, performance, and experiences deploying and supporting a data intensive supercomputer. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond.* ACM, 2012.

[58] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1), 2004.

[59] M. A. Suleman, M. Hashemi, C. Wilkerson, Y. N. Patt, et al. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on.* IEEE, 2012.

[60] C. Tian, H. Zhou, Y. He, and L. Zha. A dynamic mapreduce scheduler for heterogeneous workloads. In *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on.* IEEE, 2009.

[61] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[62] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 2012.

[63] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou. Distributed systems meet economics: pricing in the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing.* USENIX Association, 2010.

[64] Y. Wiseman and D. Feitelson. Paired gang scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 14(6), June 2003.

[65] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on.* IEEE, 2010.