More for Your Money: Exploiting Performance Heterogeneity in Public Clouds

Benjamin Farley
University of Wisconsin

Venkatanathan Varadarajan
University of Wisconsin

Kevin Bowers
RSA Laboratories

Ari Juels
RSA Laboratories

Thomas Ristenpart
University of Wisconsin

Michael M. Swift
University of Wisconsin

ABSTRACT

Infrastructure-as-a-system compute clouds such as Amazon's EC2 allow users to pay a flat hourly rate to run their virtual machine (VM) on a server providing some combination of CPU access, storage, and network. But not all VM instances are created equal: distinct underlying hardware differences, contention, and other phenomena can result in vastly differing performance across supposedly equivalent instances. The result is striking variability in the resources received for the same price.

We initiate the study of customer-controlled *placement gaming*: strategies by which a customer exploits performance heterogeneity to lower their costs. We start with a measurement study of Amazon EC2. It confirms the (oft-reported) performance differences between supposedly identical instances, and leads us to identify fruitful targets for placement gaming, such as CPU, network, and storage performance. We then explore simple heterogeneity-aware placement strategies that seek out better-performing instances. Our strategies require no assistance from the cloud provider and are therefore immediately deployable. We develop a formal model for placement strategies and evaluate potential strategies via simulation. Finally, we verify the efficacy of our strategies by implementing them on EC2; our experiments show performance improvements of 5% for a real-world CPU-bound job and 34% for a bandwidth-intensive job.

1. INTRODUCTION

Cloud computing providers use a simple billing model in which customers pay a flat hourly fee for a bundle of virtualized resources. For example, Amazon's Elastic Compute Cloud (EC2) provides a variety of instance types [2] that each offer a level of CPU power measured by abstract units (EC2 Compute Units, or ECUs) together with storage, memory, and either "low", "moderate", or "high" I/O performance. Rackspace, Microsoft Azure, and other providers are similar [24, 21].

However, not all instances of a given type are created equal. Data centers grow to contain multiple generations of hardware (*e.g.*, network switches, disks, and CPU architectures) as old components are replaced or new capacity is added. Network topology may vary, with some routes having lower latency or supporting higher bandwidth than others. Multiplexing systems across customers with different workloads can also lead to uneven resource contention across the cloud. While a provider can try to render performance uniform across all users of the same abstract instance type, in practice this is untenable due the costs of maintaining homogeneous infrastructure and enforcing rigid isolation policies.

Prior work reports that performance variability is, indeed, the common case in clouds today [7, 29, 28, 20, 34, 16, 18, 22]: a customer that runs the same application on two instances, of the same abstract type, will often end up paying the *same* amount for measurably *different* performance.

We therefore initiate the study of *placement gaming*: customer-controlled strategies for selecting instances in order to exploit performance heterogeneity. The result of successful gaming is improved efficiency, such as lowered cost for the same amount of computation, or increased performance for the same cost.

We start by addressing the following question: Do there exist performance differences, in practice, large enough to motivate placement gaming? To answer this, we perform a preliminary evaluation of EC2 by measuring the performance of a variety of resource-intensive benchmarks on a pool of machines over the course of a week. We observe variation due to different generations of processors, but, perhaps more surprisingly, differences in performance even when two instances use the same processor type. Our results, described in detail in Section 2, show moreover that variation can be large, with performance varying by more than a factor of three across different instances.

These findings motivate the development of strategies for placement gaming. We focus on the practically relevant setting in which clouds provide only coarse-grained control over placement, i.e., one can start new instances or shutdown

existing ones but cannot exert direct control over what physical machines these instances are assigned to. It may seem that this provides little ability to game placement, but in fact even these rudimentary options give rise to a classic reinforcement learning [23] setting in which one seeks to find a balance between exploitation (continued use of an instance) and exploration (launching new instances).

We provide a preliminary formal model within which strategies can be expressed and analyzed. We detail two simple mechanisms useful in building strategies. The first is *up-front exploration*, in which a customer initially launches more instances than needed and retains only ones predicted to be high-performers. The second mechanism is *opportunistic replacement*, in which a customer can choose to migrate an instance (by shutting down it and launching a fresh copy) based on projections of future performance for the instance.

Strategies require estimation of an instance's future performance. The simplest strategies we explore just look at historical performance of the customer's job, and will do well when temporal variability on instances is relatively low. They have the added advantage of being *black box*, meaning their implementation requires no understanding of the cloud infrastructure. We also explore *gray-box* strategies that leverage partial knowledge of the provider's infrastructure, such as known distributions of processor types, network topology, or scheduling mechanisms.

We build a simulator to quickly explore gray-box and black-box strategies using both synthetic and measured performance models. This allows us to quickly compare a wide range of strategies, and suggests that even easily-implemented strategies can result in large speedups. To demonstrate the deployability of such strategies in EC2, we implement controllers for a black-box, opportunistic-replacement strategy for two parallelizable applications: NER [17, 32], a compute-intensive natural language recognizer; and the Apache web server [5] driven by a bandwidth-limited workload. We compare the efficiency—average throughput in records per second or MB/s per instance hour—of this strategy against a null strategy that simply uses the instances provided by EC2. Overall, our experiments demonstrate efficiency improvements of 5% for NER and 34% for Apache, which is in line with our simulation results.

FIXME: A better placement for this paragraph? Our results are not the final word on placement gaming, indeed our simple strategies are likely to be improved upon. Moreover, we expect that our work will spark future research not only on improved strategies, but also: developing new models that take into account more cost structures or customer goals (*e.g.*, improving stability instead of maximizing efficiency), building controllers to support further applications; and reinvestigating the provider-defined abstractions and pricing structures in light of gaming.

2. PERFORMANCE ON EC2

Cloud computing environments can exhibit substantial performance heterogeneity. Past research has demonstrated variations in CPU [28], memory [28], network [28, 20, 34], disk [20, 28], and application [22] performance. In order to understand this heterogeneity, we perform a longitudinal study of workloads to determine the range and nature of variation.

The goals of our study are to quantify three types of heterogeneity:

1. *Inter-architecture:* differences due to processor architecture or system configuration.
2. *Intra-architecture:* differences within a processor architecture or system configuration.
3. *Temporal:* differences within a single machine over time.

The relative extent of these differences can motivate different placement-gaming strategies.

2.1 Methodology

We performed our experiments in the US-East region of Amazon's Elastic Compute Cloud (EC2). This region consists of four availability zones[1], labeled as us-east-1a, us-east-1b, us-east-1c, and us-east-1d (the implicit ordering arising from the letters is arbitrary). For simplicity, we focused on *m1.small* instances, which give a guest virtual machine approximately 40% of a CPU based on our measurements. Published results demonstrate similar variations on other instance types [22].

We launched 20 EC2 m1.small instances in us-east-1a at 2pm UTC-6 on 3/13/12 and 20 similar instances in us-east-1c at 6pm on 4/9/12. Every instance ran each microbenchmark once per hour for 1 week. Part way through the us-east-1c run, we were notified that one instance was on a machine scheduled for replacement; thus, we report on a total of 39 instances. Unless otherwise indicated, we analyze performance across all 39 instances and do not distinguish between zones.

Figure 1 details a set of benchmarks we use to evaluate instance performance. These benchmarks focus on three main resources: CPU, network, and storage. For CPU, we used a mix of micro- and macro-level benchmarks. Slurp and NQueens are custom benchmarks designed to stress purely the CPU. Slurp measures CPU time given to an instance, while NQueens solves the classic n-queens problem for a $n = 14$. We also used two memory-intensive benchmarks from the SPEC CPU 2006 [12] benchmark suite (mcf and sphinx3) that stress the last level cache [14], which varies widely in size across architectures and is shared across guests. For network, we used iperf [33] to measure the outbound bandwidth of instances to another EC2 instance in the same zone. For storage, we used the Bonnie++ benchmark

[1]"Availability Zones are distinct locations that are engineered to be insulated from failures in other Availability Zones and provide inexpensive, low latency network connectivity to other Availability Zones in the same Region" [1]

Benchmark	Measured resources
Slurp	CPU percentage
NQueens	CPU
mcf	CPU+Memory
sphinx3	CPU+Memory
iperf	Network bandwidth
Bonnie++	Local Disk / EBS

Figure 1: Microbenchmarks for EC2 measurement study.

suite [8] to measure the performance of two storage options, Elastic Block Store (EBS) [3] volumes and local disk, using block reads (BRD) and block writes (BWR).

In the next three sections we will report on aggregate statistics derived from all 39 launches.

2.2 Heterogeneity Across Processor Architectures

A major source of heterogeneity in cloud environments is physically different processor architectures. This can occur either when a mix of machines is purchased initially or when a data center adds machines of a different type. We identified four processor types in use by our EC2 instances, shown in Table 1. Differences between processors directly affect performance, as newer generations typically incorporate more advanced pipelines and memory systems. Furthermore, in our experiments EC2 does not penalize virtual machines using newer processors with reduced CPU time. Charts in Figure 2 shows the variation in performance for CPU-intensive workloads, network bandwidth, disk bandwidth, and network storage bandwidth across the four architectures.

Across all four resources, these results show significant differences in performance across the four processor architectures Furthermore, the magnitude of performance difference varies widely by workload. For the CPU workloads, programs that stress the memory system, such as mcf and sphinx3 show much wider performance ranges (280% improvement of E5645 over AMD for mcf). The larger cache on the Intel CPU (4-12 MB vs. 1 MB for AMD) greatly improves its performance. Disk performance is similarly variable. As with the CPU tests, the performance varies by workload: for read, the E5645 performed best (67% faster than the worst type) while for write, these machines were 3% slower than the E5430s. EBS write follows network performance, as they both stress sending data.

Our measurements also reveal that each EC2 availability zone has a different mix of CPU types. Table 2 gives the distribution of CPU architectures across zones. The data in this table is an aggregation of all EC2 instances our group has launched over the past 9 months. Overall, the mix of machines varies within each availability zone: us-east-1b appears to have predominantly E5645's, one of the best-performing architectures, while us-east-1d has no or few AMD machines.

CPU	Count	Freq.	Cache	Introduced
AMD 2218 HE	4	2.6 GHz	1 MB	Q1 2007
Intel E5430	9	2.66 GHz	6 MB	Q4 2007
Intel E5507	7	2.26 GHz	4 MB	Q1 2010
Intel E5645	19	2.4 GHz	12 MB	Q1 2010

Table 1: Frequency of observed CPUs (as reported in /proc/cpuinfo) in the measurement study and their processor frequency, last-level cache capacity, and when they were introduced.

	1a	1b	1c	1d	All
AMD	134	0	343	0	477
E5430	521	0	469	33	1024
E5507	191	4	660	113	968
E5645	224	195	418	90	928

Table 2: Counts of architectures seen in the four us-east zones.

Overall, these results demonstrate substantial performance variation of across system architectures, and that performance varies by workload. Thus, many applications may benefit substantially if they can run on the best architecture for their needs.

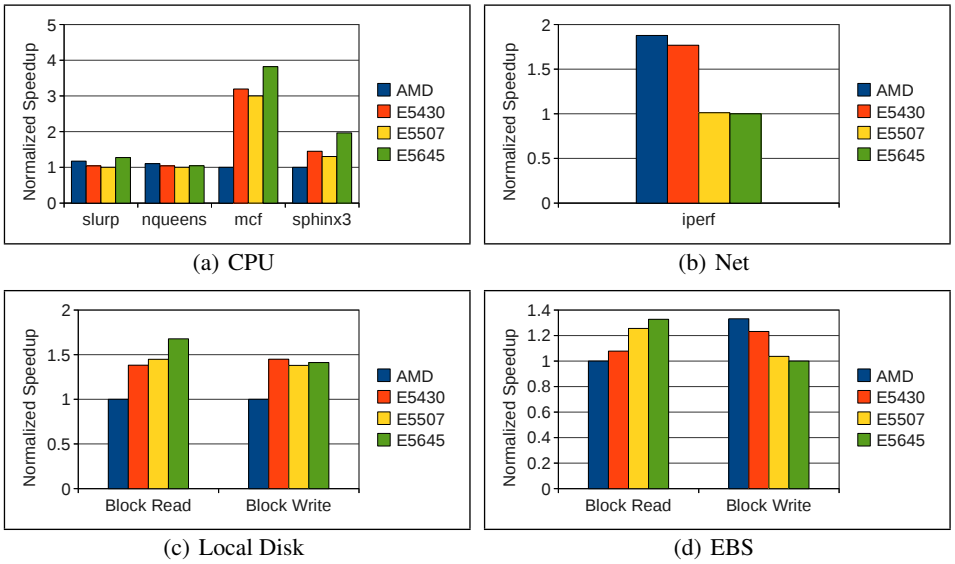
2.3 Heterogeneity Within Processor Architectures

There is also substantial variation across systems with the same architecture. This can arise from different system-level components, such as memory and peripherals, and from long-term interference from other workloads on the same node. In our experiments, we found such intra-architecture variation to be quite common. Figure 3 shows some examples of variation for CPU, network, and EBS.

Table 3 shows the speedup of the best performing instance of a given architecture type over the worst performing. Overall, CPU performance shows the least variability (0.5–15%), and the NQueens workload shows the least variability within the CPU tests. This program has a very small working set, and thus its performance is dominated by the processor pipeline and frequency, which vary less across machines. For CPU-sensitive applications the variation within a processor type tends to be smaller than that between different processor types, while for disk and network the opposite is true. These results demonstrate that selecting desirable purely by CPU architecture may be insufficient, as there are other sources of heterogeneity.

2.4 Temporal Heterogeneity

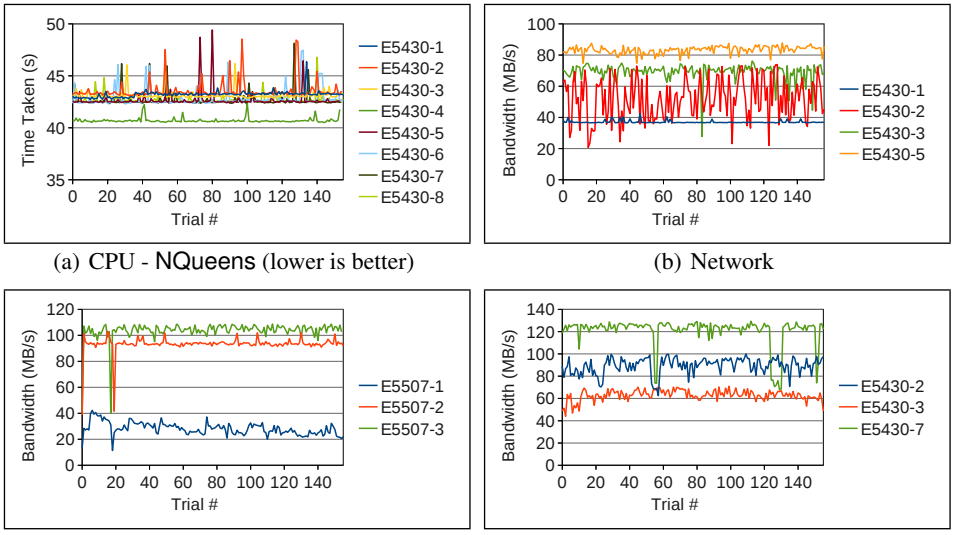
The performance of a single instance can vary over time due to competing workloads. We report the minimum, maximum, and average coefficient of variation (CoV) for all workloads in Table 4. As shown in the examples from Figure 3, the performance of a single machine can be very flat, *e.g.*, the network bandwidth of machine E5430-1 in (b). For comparison, machine E5430-2 in the same plot exhibits bandwidth variations between 20s to 75MB/s.



(a) CPU (b) Net

(c) Local Disk (d) EBS

Figure 2: Inter-architecture performance variation of benchmarks. Speedups are relative to the worst-performing architecture.



(a) CPU - NQueens (lower is better) (b) Network

(c) Local Disk - block write (d) EBS - block read

Figure 3: Examples of intra-architecture performance variation.

Res.	Benchmark	AMD	E5430	E5507	E5645
CPU	Slurp	3.7%	7%	2.2%	2.7%
	NQueens	3.5%	7%	1%	0.5%
	mcf	8.2%	13.6%	4.9%	10.6%
	sphinx3	9.6%	8.5%	6%	14.8%
Net	iperf	12.1%	125%	11.2%	11.5%
Disk	Local-BRD	12.6%	15.6%	33.9%	37.7%
	Local-BWR	66.5%	32.7%	269%	109%
	EBS-BRD	23.4%	106%	29.3%	45.4%
	EBS-BWR	18.7%	87.2%	11%	40.5%

Table 3: Speedup of the fastest instance's average performance over the slowest instance's average performance for each architecture.

Thus, the stability of a workload depends highly on the machine on which an instance is placed, as well as the resources it uses. On average, CPU performance is the most stable with an average CoV of 1.7%, while storage performance (both local disk and EBS) shows the highest average CoV at 9.8%. These results demonstrate that the hypervisor's ability to isolate competing workloads varies by the resources used; for example, we can conclude from this that Xen is much better at fairly sharing CPU time than disk access. This fits well with previous studies of isolation in Xen [11] .

2.5 Summary

Res.	Benchmark	Min	Max	Average
CPU	Slurp	0.12%	3.3%	1.1%
	NQueens	0.13%	3.6%	0.92%
	mcf	0.75%	13.3%	2.4%
	sphinx3	0.6%	13/0%	2.8%
Net	iperf	0.35%	25.4%	4.4%
Disk	Local-BRD	0.5%	10.2%	4.0%
	Local-BWR	4.9%	20.3%	10.7%
	EBS-BRD	4.0%	40.9%	12.1%
	EBS-BWR	2.7%	24.9%	12.5%

Table 4: The minimum, maximum, and average coefficient of variation of 39 EC2 instances for each resource.

Overall, these experiments demonstrate significant variation between architectures, individual machines, and over time. Table 5 summarizes the result. The variation between architectures demonstrates that cloud customers may want to select specific processor types to optimize performance. However, the performance variation within a processor type further shows that this may be insufficient for I/O performance, as there is similarly high variation within a single processor type. Furthermore, the variation of individual nodes over time indicates that workloads prefer nodes with more stable performance and benefit migrating away from congestion.

Resource	Inter-arch.	Intra-arch.	Intra-node
CPU	282%	15%	13%
Network	88%	125%	25%
Local Disk	67%	269%	20%
EBS	33%	106%	41%

Table 5: Summary of inter- and intra-architectural differences. Each row shows the maximum speedup in percent, for both types of variation for each resource.

3. PLACEMENT GAMING

Cloud providers give customers little control over the actual placement of their tasks. As shown in the preceding section, certain machines may give much better performance than others. The goal of our work is to allow customers to control the placement of their tasks on machines with good performance, which can lower the cost of a task or reduce its completion time.

We exploit several features of cloud computing to make this possible. First, cloud providers often bill for usage on fine time scales; for example, EC2 charges by the hour. Thus, it is possible to use a machine for an hour and give it up if it is too slow. Second, cloud providers allow programmatic control over launching virtual machines. This allows control software to dynamically launch additional instances in the hope of improving performance. Third, cloud workloads are often cost-associative and can use varying numbers of machines; this allows the productive use of extra machines for short periods. Finally, many cloud workloads store their persistent state in networked storage, such as Amazon's EBS network blockstore. Thus, moving a workload from one machine to another need not entail migration of the entire data set. In addition, applications run in distributed systems like EC2 are often intentionally optimized to be able to shutdown and restart quickly.

We begin by creating a formal model of the placement problem that can be used to develop specific strategies for improving performance. Our setting is that of a customer, the *tenant*, running a job denoted by J on a public cloud such as Amazon's EC2. The objective of placement gaming is to schedule J, using only the provider's legitimate API, in a manner that optimizes performance relative to cost. As mentioned previously, cloud providers give tenants limited control over instance placement; typically, a tenant has the ability to start and stop instances of various types in a particular portion of the cloud (availability zone or region).

We note that for some workloads, selecting a specific availability zone with a higher percentage of high-performing machines can trivially improve performance. However, performance variation between machines of the same architecture may still exist in all zones. Furthermore, not all workloads benefit from faster CPUs and instead prefer higher network bandwidth. In addition, customers may prefer to use multiple availability zones to improve reliability, or if the number of available instances in a given zone is limited. Similarly, a customer may achieve different performance from other instances types, such as small (shared core) or large (dedicated multiple cores) instances. However, we focus only on strategies that exploit the heterogeneity within a particular zone, which applies to all instance types.

3.1 A Placement Model

We describe a general model for the placement problem that abstracts the problem away from specific applications or cloud providers. For simplicity, we assume that the tenant launches servers synchronously, and that servers are scheduled with a minimum granularity that we refer to as a *quantum*. In EC2, for example, the most natural job quantum is one hour, since this is the minimum billing unit. Job time is organized into integral quanta, denoted by $t = 1, 2, \ldots$.

The provider draws servers from some pool. There is variation across servers in the amount of job-specific work they perform within a time quantum (cross-instance heterogeneity). There is also variation across the amount of work performed by a single instance across time quanta (temporal heterogeneity). Thus, we denote the *rate* of a given server S during a time quantum t as $r_t(S) \in [0, \infty]$.

We emphasize that a server's rate is job-specific, as different jobs leverage different blends of resources. Looking ahead, our case studies use rates representing the number of records processed per unit time for a natural-language processing task and the throughput of a cluster of web servers.

We assume for simplicity that the performance of each server is dictated according to a probability distribution $\mathbb{S}$. Thus the sequence of rates $r_1(S), r_2(S), \ldots$ is drawn independently from $\mathbb{S}$ for each S.

A *placement schedule* P is a sequence of sets of servers

denoting job placement across time. Let $P[t]$ be the set of servers assigned to a job in time quantum t. For a placement schedule P of duration T, the *cost* is

$$c(P) = \sum_{t=1}^{T} |P[t]| \ ,$$

i.e., the total number of server-quanta units it consumes. We model overheads associated with launching an instance by a *time penalty* m, which is the fraction of the first quantum consumed by overheads. For example, our simulations will use a very conservative estimate of $m = .05$ which is 180 seconds if the quantum is an hour. We do not model any other costs of migration, such as data transfer fees. Let $first(S) \in \{1, \ldots, T\}$ denote the quantum at which the job was launched on S. The *achieved rate* of S is

$$\tilde{r}_t(S) = \begin{cases} r_t(S) \cdot (1-m) & \text{if } t = first(S) \\ r_t(S) & \text{otherwise} \end{cases} \ .$$

Then the *work* yielded by a placement schedule P is

$$w(P) = \sum_{t=1}^{T} \sum_{S \in P[t]} \tilde{r}_t(S) \ .$$

In addition, let $w(P[t]) = \sum_{S \in P[t]} \tilde{r}(S)$ denote the work output by a placement schedule in quantum t, and $c(P[t]) = |P[t]|$, similarly, denote the cost.

It is also useful to define the *support* of a placement schedule P as $supp(P) = min_{t=1}^{T} |P[t]|$. This is the minimum number of servers executing J in any quantum.

3.2 Placement Strategies

A tenant employs a *placement strategy* σ, which is a scheme that guides the construction of a placement schedule as a job executes. Tenants can control placement only indirectly, by starting and stopping instances. We therefore focus on strategies σ that, at the end of any quantum $t > 0$, determines: (1) which servers in $P[t]$ to terminate and thus which servers in $P[t]$ to *retain*, *i.e.*, continue running in quantum $t+1$; and (2) how many fresh servers to request from the provider in quantum $t+1$. More formally, at the end of quantum t, a placement strategy determines:

1. A set of servers $K[t+1] \subseteq P[t]$ to *keep*, *i.e.*, to include in $P[t+1]$. All other servers are terminated. Let $k[t] = |K[t]|$.
2. A number $f[t]$ of *fresh* servers to invoke in the next quantum, i.e., to add to $P[t+1]$. We denote the resulting set of fresh servers $F[t+1]$.

Thus, $P[t+1] = K[t+1] \cup F[t+1]$. Note that for $t = 1$, all servers are fresh instances, *i.e.*, $P[1] = F[1]$. Before job execution, at the end of quantum $t = 0$, σ determines the number of initial servers to spin up by setting $f[1]$. As there are no servers to retain at the beginning of a job, $K[1] = 0$.

Formally, then, a strategy σ takes as input the current time t, the so-far placement schedule $P[1 \ldots t]$, and the so-far observed rates. It outputs $(K[t+1], f(t+1))$.

There are several natural performance objectives for a placement strategy. A tenant may wish to minimize the cost or the latency of executing a job J. Or the tenant may wish to bound the cost of its execution of job J, but maximize the amount of achieved work. Additionally, as a placement strategy σ generates schedules P probabilistically based on the distribution $\mathbb{S}$ of available machines, "maximization" may be defined in any of a number of ways: In terms of expected work $w(P)$, in terms of the probability of $w(P)$ exceeding a certain threshold, etc.

For simplicity, we focus on the objective of *maximizing the efficiency* $e(P)$ of a job J. This is the work per unit cost averaged across all the entire execution: $e(P) = \frac{w(P)}{c(P)}$ where $c(P)$ incorporates both the time spent executing the workload as well as time for testing the performance of instances and for migrating an application between instances.

3.3 Types of Strategies

A placement strategy embodies a tradeoff between exploration and exploitation. It may be beneficial to retain and repeatedly use a "good" server S, i.e., one with a high rates. Conversely, though, launching additional servers offers the possibility of discovering new servers with higher rates than retained ones.

The problem of job placement may be viewed as a Markov decision process (MDP) [36], in which the set of servers $P[t]$ at time t is the system state, and specification of $(K[t+1], f(t+1))$ is an action. However, for complex distributions $\mathbb{S}$ of performance and large number of servers, the state space is quite large. Solution approaches such as dynamic programming may be computationally costly, and also have the drawback of yielding complicated placement strategies and cost structures.

Instead, we consider a restricted space of placement strategies $\Sigma_{(A,B)}$ that we call (A,B)-strategies. These run at least A servers in every quantum and launch an additional B "exploratory" instances for one quantum each at some point during the execution. This model covers a wide variety of strategies that either launch additional instances (B) solely for the purpose of evaluating their performance as well as strategies that replace executing instances (A).

More formally we have:

DEFINITION 3.1. *For fixed number T of quanta, an (A,B)-strategy $\sigma \in \Sigma_{(A,B)}$ is one that always yields a placement schedule P in which $supp(P) \geq A$ and $\sum_{i=1}^{t} f[i] = A+B$. An (A,B)-strategy has fixed cost $c(P) = TA + B$.*

An example is the $(A,0)$-strategy σ_{null}, the *null strategy*, that launches A instances in the first quantum and uses them for the remainder of the schedule. In fact this strategy is optimal should all servers offer precisely the same performance. When heterogeneous performance is the norm, though, the class of $\Sigma_{(A,B)}$ allows more refined strategies. We will explore two general mechanisms useful in building (A,B)-strategies:

- **Up-front exploration:** We would like to find high-performing instances early so that we can use them for longer. An (A, B)-strategy that uses *up-front exploration* launches all B "exploratory" instances at the start of a job, i.e. at time $t = 1$. At $t = 2$ the highest performing A instances are retained and the other B instances are shut down.
- **Opportunistic replacement:** By migrating an instance—shutting it down and replacing it by a fresh one to continue its work—we can seek out better performing instances or adapt to slow-downs in performance. An (A, B)-strategy that uses *opportunistic replacement* will retain from time t any instance that is deemed a high performer and migrate any instance that is a low performer.

These mechanisms rely upon accurate judgments about instance performance: up-front exploration must rank server performance and opportunistic replacement must distinguish between low and high performance. Here we find a natural dichotomy exists between strategies that do so by exploiting partial knowledge about the infrastructure and those that do not:

- **Gray-box (GB) strategies** make decisions based in part on partial knowledge of the provider's infrastructure, such as hardware architectures, network topology, provider scheduler, or historical performance of instances. For example, we will explore strategies that leverage the distribution of CPU architectures historically observed in EC2.
- **Black-box (BB) strategies** use only the measured performance (rate) of the tenant's instances.

While gray-box strategies are potentially more powerful, black-box strategies can be simpler and more portable. For example, they can work for more causes of heterogeneity and for unknown machine distributions.

We will explore several (A, B)-strategies later in the paper. Here we make all the above concrete by detailing our simplest black-box strategy, called PERF-M, which combines up-front exploration (when $B > 0$) with opportunistic replacement. At a high level, the strategy uses the up-front exploration stage to estimate average performance of the job, and then in the remainder of the run the strategy attempts to beat that average. PERF-M ranks instances at the end of $t = 1$ based solely on their performance during $t = 1$.

For opportunistic replacement, PERF-M migrates an instance if its recent performance drops sufficiently below the average performance of all instances that ran in the first quantum. To define "sufficiently", we set a heuristic threshold that estimates the expected cost of the migrations needed to achieve above-average performance, amortized across the remaining time of the run. Formally, the replacement rule is that a server S will be migrated should

$$\overline{avg}_1 - r_t(S) > \delta = \frac{2m}{T - t} \quad (1)$$

where $\overline{avg}_1 = \sum_{i=1}^{A+B} r_1(S)/(A + B)$ (recall m is the time penalty of migration and T is the total duration of computation).

In detail the strategy works as follows.

(1) Launch $A + B$ instances at $t = 0$. The set of instances is $P[1]$.

(2) At the end of the first time quantum, measure the rate $r_1(S)$ for each instance $S \in P[1]$. Compute the mean performance as $\overline{avg}_1 = \sum_{i=1}^{A+B} r_1(S)/(A + B)$. Let the retained set $K[2] \subseteq P[1]$ include instances S such that both: (i) S is one of the top A performers within $P[1]$ and (ii) $\overline{avg}_1 - r_1(S) \leq \delta$. Shutdown all other instances, i.e. $P[1] \backslash K[2]$, and launch $A - |k[2]|$ fresh instances to maintain a minimum of A instances in every period.

(3) At the end of each quantum $2 \leq t < T$ and for each instance $S \in P[t]$, put S in $K[t+1]$ if $\overline{avg}_1 - r_t(S) \leq \delta$. Shutdown all other instances $P[t] \backslash K[t+1]$ and launch $A - |k[t + 1]|$ fresh instances.

Thus, the strategy starts more than the necessary number of instances and shuts down the slowest B of them at the end of the first period. In addition, in every period it shuts down all machines with below-average performance and replaces them.

We note that an embellishment on PERF-M would be to use more moving estimates of both the average performance and each server's performance, for example by using exponentially weighted moving averages.

4. STRATEGY SIMULATIONS

We evaluate various configurations of the two basic placement strategies, up-front exploration and opportunistic replacement, using a simulator. This allows us to quickly evaluate the impact of different cloud provider configurations, such as the mix of fast and slow machines and of performance variability (due to performance isolation or lack thereof), and of various workloads.

These simulations are helpful both to predict the expected behavior of various strategies, but also to understand the effect of workload or configuration on the possible speedup from placement gaming.

4.1 Methodology

We construct a simulator that takes as input a distribution of machines and of machine performance. The distribution can be purely synthetic, to explore the design space, or taken from measurements, and it is specified by indicating a set of instance types. Each type has associated to it a distribution fraction in $[0, 1]$ representing the prevalence of this type, as well as a mean rate and a standard deviation for rates. The distribution fractions of all types should sum to one. To select a new instance, then, the simulator randomly chooses an instance using weights indicated by the distribution fractions. Then, the instance's per-quantum performances are selected as independent normal random variables with the

mean and standard deviation of the type.

The simulator also takes as inputs what strategy to run and the necessary related parameters, i.e., A, B, T, and m.

The simulation uses the indicated strategy and parameters. Any time an instance is launched, its rates are selected as described above. The rates this selects are used for the strategy's scheduling decisions and also to calculate total work accomplished. In every run we also simulate the null strategy σ_{null} using the first A instances launched. This allows us to directly compute the speedup offered by a placement schedule P arising from a strategy and the placement schedule P_{null} for the null strategy by

$$spd(P, P_{null}) = \frac{e(P)}{e(P_{null})} = \frac{w(P) \cdot T \cdot A}{w(P_{null}) \cdot (T \cdot A + B)}$$

where we have substituted in for the costs of the two placement schedules, $c(P) = T \cdot A + B$ and $c(P_{null}) = TA$. Work completed is calculated using the achieved rate (which subtracts time lost due to migration) times the total number of instance hours.

4.2 Synthetic Simulations

Three major features of the cloud environment affect the achievable speedup from placement gaming: the magnitude of the difference in performance between machines, the performance variability observed by applications, and the distribution of differently performing machines. The size of the difference between machine performance essentially determines the cost of running on a slow machine (or conversely, the benefit of being on a fast machine). Performance variability affects the ability of a placement strategy to accurately predict future performance from a small number of measurements, while the distribution of machines affects the likelihood of improving performance by launching to a new instance.

To limit the scope of our parameter search, we fixed $T = 24$ and $A = 10$ to indicate a workload that ran for 24 hours with 10 instances computing at all times. We explored a range of values for B and evaluated both up-front exploration and opportunistic replacement strategies. While holding other variables constant we varied (in turn), the difference between mean architecture performance, variability within an architecture, and the distribution of machines between the two architectures. For simplicity, we here look at a distribution $\mathcal{S}$ of only two types of machines, "good" machines and "bad" machines with an even likelihood of each. As these are only synthetic results, we present a qualitative description of the results rather data.

Architecture variation. First, we perform an experiment in which we hold the performance variability (standard deviation) steady at 5% of the average performance (a number chosen to match what we observed in our testing of EC2) and evenly spread machines between the two distributions. We then varied the gap between "good" machines and "bad" machines from 10% to 50%.

Up-front exploration strategies do significantly better as the separation between "good" and "bad" instances increases. When the separation is small, the two distributions overlap so much that intra-instance variation is nearly as large as inter-instance variation. An upfront strategy measuring performance during the first time quantum cannot determine which instance will perform best in the long run. Replacement strategies do better than the upfront strategies in all cases, and the effect of larger separation is seen most as B increases. When the separation is small, replacement strategies perform less well at determining when to migrate and increasing B only raises cost without helping migration performance, lowering the overall performance.

Instance variability. Next we looked at the impact of variability within a machine's performance by holding steady the average performance difference between "good" and "bad" instances at 30% (again, a number we derived from results we've seen in our testing of EC2) and varying instead the standard deviation of performance within that instance type. We vary the standard deviation from 2.5% up to 10%.

What we found, not surprisingly, is that performance variability does not have much impact on the upfront strategies, but does impact the gains seen by our replacement strategies. The upfront strategies merely sample from the combined distribution, whose average remains constant. Widening of the distributions allows for a few slightly better instances, which are canceled out by a few slightly worse instances available and the larger number of average instances. If we were able to raise B significantly, we could sample more of the space and potentially do better with wider distributions, but this would require a much longer time horizon to recoup the cost of a larger initial search.

In a replacement strategy, however, the tighter the distribution, the better the performance achieved. This is particularly visible when $B = 0$. In this case, we are not launching any extra instances at the start and relying solely on migrations to improve performance. When the distributions are tight, all of the "bad" instances will be significantly less than the average, causing an attempted migration. When this migration is successful in moving that instance to the "good" distribution, the performance improvement will be significant. Two things happen as the distributions widen: 1) some of the "bad" instances move close enough to the average to avoid migration, 2) successful migrations may be only a slight improvement.

Architecture mix. Lastly we look at how the distribution of machines affects each strategy. So far we have been analyzing a 50/50 split, but it is important to understand how each strategy fairs when the cards are stacked in its favor, or against it. Again, we test a number of distributions.

Upfront strategies naturally perform best when the fraction of "good" machines is high. For a low fraction of "good" machines, the performance of upfront strategies increases linearly as B increases, and increases faster the higher the fraction of "good" machines, eventually topping out once

the total number of instances started ($A + B$) is enough to ensure that at least A of them are in the "good" distribution. As the fraction of "good" machines goes up, a smaller B is needed to achieve the same performance, which lowers total cost.

For replacement strategies, the analysis is a little more complex. If the number of "good" machines is too small, the initial instances may all land in the "bad" distribution and the existence of better machines may be missed. If the number of "good" machines is small, the difficulty migrating to a "good" machine may outweigh the benefit of getting there, especially if the length of the workload is short. Performance of the replacement strategies increases as the fraction of good machines increases until roughly 1/3rd of the machines are "good". Above that point, replacement strategies continue to outperform both the null and upfront strategies, but the percentage improvement over a null strategy starts to go down as the null strategy begins to improve. The opportunity for improvement over a null placement strategy seems optimal around the point where 1/3 of the machines perform better than the rest.

5. APPLICATIONS

We next demonstrate the real-world efficiency benefits of placement gaming on EC2. We focus on two workloads, one CPU-bound and one a bandwidth-bound.

To stress CPU performance, we use the Stanford Named Entity Recognizer (NER) tool [32]. This tool performs perfectly parallelizable natural language processing tasks used by several research groups at our university [41, 38, 30]. NER operates on short strings of text called records, which are batched together into chunks. For our experiments, we obtained a data set, used locally [41], consisting of approximately 12 million records. We split this data set up into 4 MB chunks; each invocation of NER processes one of these chunks and produces a 20 MB file of annotated text.

To stress network bandwidth, we use the Apache web server. Each instance runs Apache version 2.4.2 hosting a single 100 MB file. For each web server, we launch two separate client instances that repeatedly request the file with a concurrency level of ten using ApacheBench version 2.3 [4]. The rate for a server is the sum of the bandwidth measured by both clients. By using two clients, we create a bottleneck at the server and ensure that we are not unintentionally measuring other network characteristics, such as client bandwidth.

In the preceding sections we reported per-quantum rates. Here, we instead use application-specific metrics (records/sec and MB/s), and convert between per-quantum rates only when needed.

5.1 Performance Measurements

In order to verify the possibility of improving performance through placement gaming, we performed an initial round of performance heterogeneity measurements. For both NER and Apache, we ran 40 m1.small EC2 instances for 24 hours.

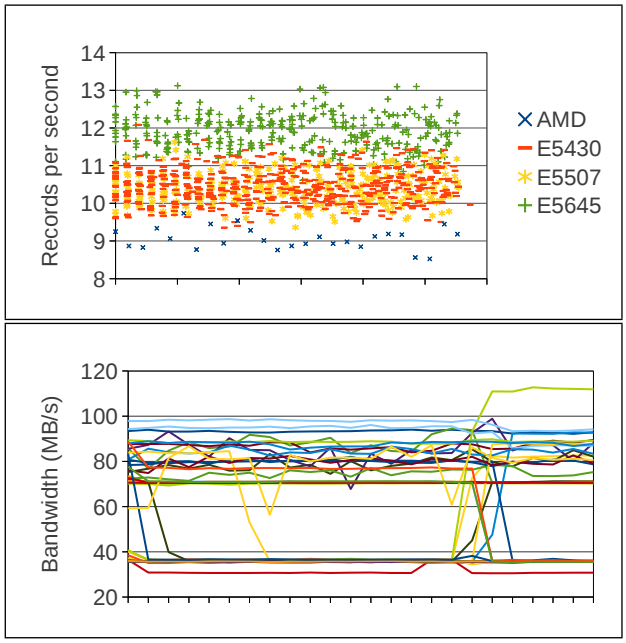


Figure 4: Performance over a 24 hour period for NER (top) and Apache (bottom).

CPU	Fraction	Mean	StDev
AMD	0.18	9.07	0.29
E5430	0.25	10.47	0.44
E5507	0.35	10.38	0.62
E5645	0.22	11.94	0.47

Table 6: Performance statistics for NER broken down by architecture. The fraction column indicates the percentage of such CPUs in us-east-1c. The means and standard deviations are in records per second.

Figure 4 (top) depicts a time series of performance while Table 6 gives a breakdown, by CPU architecture, of the performance in records per second of NER. As expected, certain architectures perform better than others: the best-performing architecture, E5645, performs 31% faster than the slowest, AMD. The temporal variation for NER is quite low, meaning the performance over time is stable and predictable.

For Apache, the measurement study revealed more complex behavior. Figure 4 (bottom) depicts a time series, drawn as connected lines, of the instances' performance. Compared to NER, there is a wider range of performances, with over a 270% difference between the best and worst performers. Significantly, many instances underwent large, rapid shifts in throughput. Unlike NER, performance is only loosely correlated with architecture, with AMDs and E5430s performing better in general (but not always) than E5507s and E5645s. There are also clear performance bands: on the low end near 35 MB/s and a slightly larger band between 70 MB/s and 90 MB/s. We do not know the exact reasons for these bands, but clearly EC2 is shaping traffic, perhaps due

Strategy	Up-front Exploration	Opportunistic replacement rule
CPU	CPU	Never
CPU-M	CPU	$\overline{C} - cpu(S) \geq \delta$
CPU-MAX	CPU	$cpu(S) \neq \max_i\{cpu(S_i)\}$
PERF	Performance	Never
PERF-M	Performance	$\overline{avg}_1 - r_t(S) \geq \delta$

Table 7: Gray-box (first three rows) and black-box (last two rows) strategies.

to contention or topology. Past research has shown that up to eight m1.small instances may reside on the same system, which could lead to contention for bandwidth [26]. In further experiments with a client external to EC2 we observed similar behavior, suggesting that this phenomenon is not entirely topology driven.

5.2 Concrete Strategies

We begin with several gray-box (A, B)-strategies for NER. These rely on a one-time measurement phase that correlates performance of the NER task with different architectures (as reported by CPUID) and, implicitly, on the distribution of architectures available in a zone. In our experiments, we will use the just-reported measurement study as this one-time measurement.

The CPU strategy performs up-front exploration using $A + B$ instances, retains the A instances with fastest the CPUs for NER, and subsequently performs no migrations. The CPU-M strategy performs the same up-front exploration as CPU and performs a migration using a thresholding equation similar to Equation 1. Particularly, let $cpu(S)$ be the rate of the architecture of S, as measured in the one-time measurement phase. Let $\overline{C}$ be the average performance across all instances Then, a server S will be migrated if

$$\overline{C} - cpu(S) > \delta \ ,$$

where δ is as defined in Section 3.

We also fix an opportunistic (A, B)-strategy that simply seeks out the best architecture as reported by the measurement phase. The CPU-MAX strategy performs the same up-front exploration as CPU and then performs migration whenever an instance's CPU is not the best performing. Table 7 summarizes these three CPU strategies. These strategies are not useful for the Apache workload, which our measurements show does not have strong correlation with architecture.

We also explore two black-box strategies that apply both to NER and Apache. The PERF strategy executes an up-front exploration: run $A + B$ instances for the first quantum and then shut down the B worst-performing ones. The last strategy is PERF-M (detailed in Section 3), which uses the same up-front exploration as PERF but also performs opportunistic replacement by migrating any instance whose recent performance was below the first quantum's average.

5.3 Simulation of the Strategies

We first use simulation as educated by the initial NER and Apache measurements to evaluate the strategies. This requires fixing suitable job-specific approximations of $\mathbb{S}$, the distribution from which new instances are drawn. We let $\mathbb{S}_{\text{ner}}$ be defined by the following sampling procedure: (1) select a CPU type uniformly according to the fractions of their occurrence in our measurements of us-east-1c and (2) choose their per-quantum performances $r_t(S)$ as independent, normally distributed random variable with mean and standard deviation as measured for that architecture in the above experiments. Table 6 therefore provides the data defining $\mathbb{S}$. Our simulator can be used as-is with $\mathbb{S}_{\text{ner}}$.

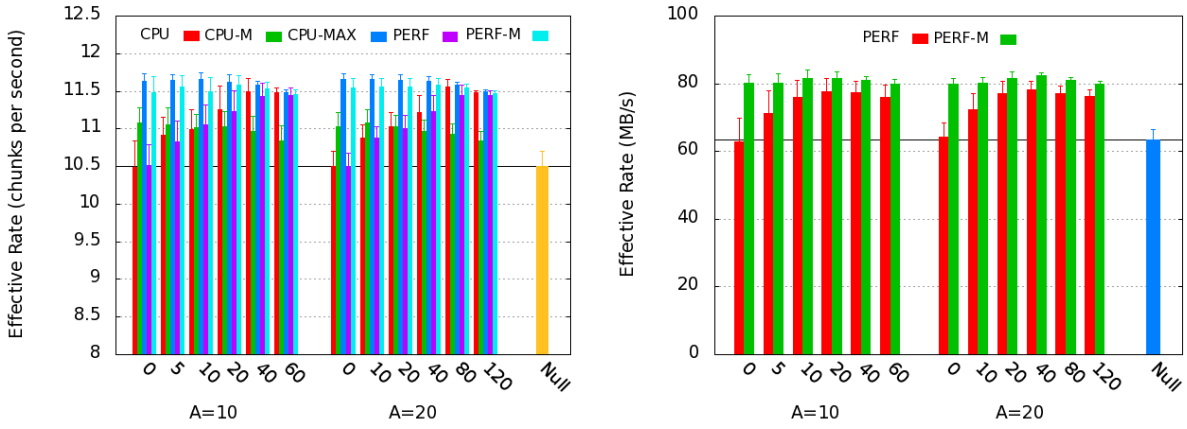
The sampling procedure for $\mathbb{S}_{\text{apache}}$ is more involved, because we would like to capture the observed large shifts in performance. Rather than clustering by CPU type, we visually identified four performance ranges and then calculate the fraction of machines within those ranges. In addition, to include the effect of performance shifts, we assign probabilities of a one-time shift pairwise between those ranges. Overall, there is a 25% chance of a shift for a given instance. We then pick a random shift time for those instances selected for a shift.

We modify our simulator to support $\mathbb{S}_{\text{apache}}$ in addition to $\mathbb{S}_{\text{ner}}$. This allows us to explore (potential) speedups across a wide range of settings. We conservatively set migration cost to $m = .05$ (3 minutes/hour) in all the following experiments.

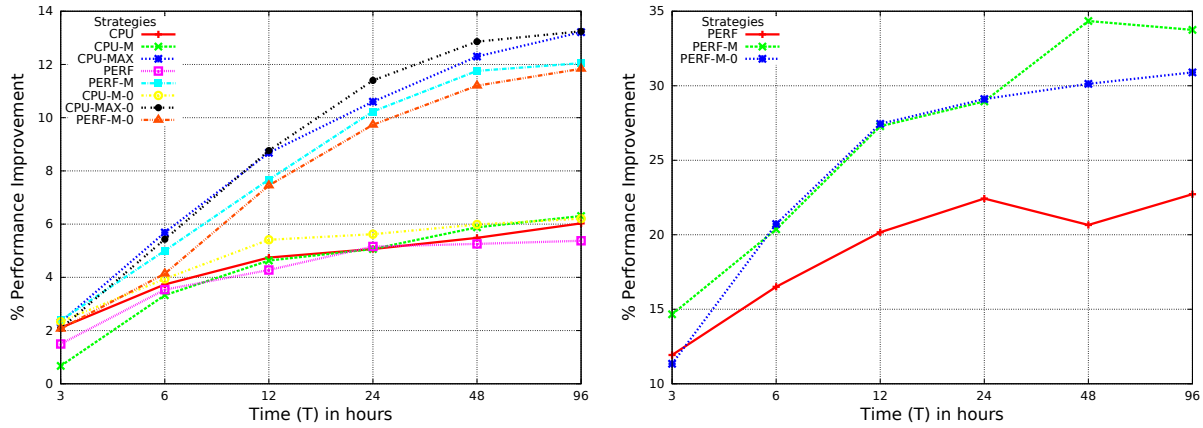
Figure 5 shows the results of our simulations. The top two charts show the average performance over 100 trials of the various strategies applied to NER and Apache for $A \in \{10, 20\}$, for $B \in \{0, 0.5A, A, 2A, 4A, 6A\}$, and for $T = 24$. The error bars indicate one standard deviation. Note that with $B = 0$ the CPU and PERF strategies are equivalent to the null strategy, and hence have no speedup. We highlight two trends. First, as $A + B$ increases, the CPU, CPU-MAX, PERF, and PERF-M strategies converge in performance. Second, we note that the best performing GB strategy is CPU-MAX, which seeks to place all instances on high-performing architectures. Surprisingly, the best BB strategy PERF-M performs almost as well.

The middle two charts show speedups as a function of T. For clarity we have omitted error bars, but note that Apache's variance is high (10%) for large values of T. This is because speedup is the ratio of two already variable performances. As expected, performance improves with increasing T, and the improvement curve for the strategies incorporating opportunistic. Finally, the average numbers of migrations for strategies using opportunistic replacement are shown in Figure 6. The same trends as in NER arise.

We note that the number of migrations with PERF-M for Apache is much less than for NER. This is due to the much higher performance gap observed in Apache, whereas



(a) Varying A and B with T=24



(b) Varying T with A=10 and B=10

Figure 5: Simulated NER (left column) and Apache (right column) performance for various strategies as A, B vary (top), speedups as T increases (middle), and number of migrations due to opportunistic replacement (bottom).

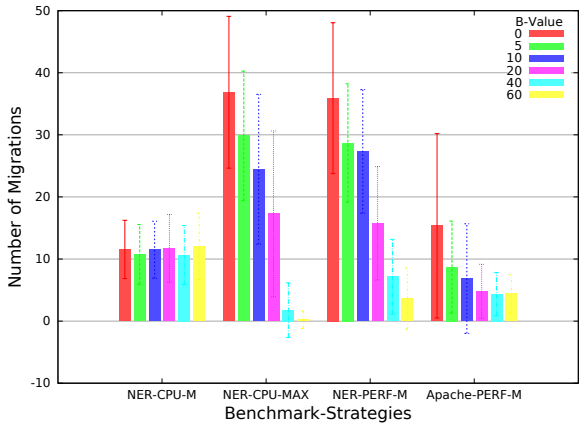


Figure 6: Average Number Migrations with varying B, A=10, T=24

in NER temporal variation can occasionally overwhelm the performance gap between CPUs of different types.

6. EXPERIMENTS ON EC2

The prior section laid out strategies for use with the NER and Apache workloads and showed that, in theory, they will provide speedups. In this section we develop a proof of concept controller implementing the PERF-M strategy for use with EC2. We chose PERF-M due to its applicability to both NER and Apache, and to assess the overheads of opportunistic replacement in contemporary clouds.

6.1 Cost and Performance Model

The goal of placement gaming is to improve the efficiency, the work accomplished per unit price. For NER, the work is the number of unique records processed across the experiment. This accounts for migration costs, as periods when an instance started or stopped will process fewer records. For Apache, where we measure bandwidth periodically, we in-

stead compute work as the total number of bytes that could have been delivered at that bandwidth for the portion of the hours that Apache was running. If the instance neither started nor stopped during the hour, then it is the full hour. Otherwise we conservatively estimate that stopping takes one minute, to ensure we stop before the next hour starts, and that starting takes two minutes to boot the operating system, start Apache, and fetch the data file from EBS. In practice, we often see Apache starting much more quickly.

EC2 bills contain only coarse-grained information. We therefore approximate cost as the number of hours for which we ran instances for a particular experiment. We assume that we shut them down before triggering another hour of billing. We do not include the cost of the controller, which runs for the duration of the experiments, because it is a fixed cost independent of the number of instances. In addition, we did not include the storage cost of launching an instance. Amazon charges $0.10 per million I/O requests to EBS, which stores the virtual disks for our instances. However, we were unable to directly measure the number of requests performed during system startup. Should this cost be prohibitive, one can also use Amazon's Instance Store, which does not charge transfer costs but is slightly slower to launch an instance.

6.2 Experiments with PERF-M

As a proof of concept that the ideas we have discussed can be put into practice, we created a system that implements the PERF-M strategy for both the NER and Apache jobs. To do so, we implement a prototype controller that initially launches the desired number of worker instances and forks off threads for each instance in order to monitor instance performance and make migration decisions. For NER, workers used a shared queue hosted on the controller (via NFS) to coordinate which records need to be processed.

To analyze the efficacy of PERF-M in the setting of EC2, we use the following experimental procedure. First launch the target number of instances and run the appropriate workload using the null strategy (i.e., no migrations). For NER, we run for as long as it takes to complete processing of the data set (approximately 12 hours with $A = 10$), while for Apache we run for 12 hours. When the null strategy finishes, rather than shutting the instances down, we retain them in order to perform an execution using the PERF-M strategy (i.e., with migrations). This allows us to measure the performance of both strategies for the same set of initial A instances. We repeat this three separate times for NER and for Apache.

Table 8 summarizes the results of the 6 experiments, giving the speedups achieved by PERF-M, the number of migrations, and the total migration cost (in seconds). As we would expect from the simulations in the last section, we see significant variability in speedups. For NER speedups range from 1.49% to 5.68%. For Apache the range is even wider, and we see here the potential for significant speedups up to 34.73%. The variability in both cases arises from the makeup of the first A instances: if these end up, by chance, to be high performers, then the null strategy will perform well.

Runs	Speedup	# Migrations	Total migration cost (sec)
NER 1	5.68%	13	376
NER 2	1.49%	12	402
NER 3	3.82%	16	472
Apache 1	3.71%	10	283
Apache 2	22.57%	7	296
Apache 3	34.73%	16	472

Table 8: Experiments on EC2 for NER and Apache with PERF-M strategy.

We can also calculate the actual cost savings for NER. (For Apache, we achieve more throughput for the same price.) If we include in the calculation paying for both hours and partial hours of the workers, then the cost savings varied between 2% and 4% across the experiments. We expect that improvements to our (unoptimized) implementation, not to mention further strategy refinement, will surface increases in cost savings.

7. RELATED WORK

Our work draws on past work looking at performance heterogeneity in cloud computing and in scheduling for heterogeneous systems. Most similar is the just-published work of Ou et al. [22]. They similarly observe the mix of architectures with different performance characteristics. However, they show small differences between instances of the same architecture, while we observe much larger differences. Furthermore, this work presents only analytic results indicating possible speedups, while we present the benefit of specific strategies together with simulations and experiments.

Evidence of heterogeneity. Many works offer evidence of performance heterogeneity in cloud providers [7, 29, 28, 20, 34, 16, 18]. Most of these works focus on variability over time, which often arises from competing workloads [25]. Several works break down the variations by type and show long-term patterns, such as periods of stability [9, 13], which can be used to predict performance for better scheduling. Similar studies of heterogeneity have been performed for data-center workloads such as Hadoop [15].

In contrast to these works, our research shows discernible long-term patterns in performance between architectures, within an architecture, and within a node. Thus, it presents a unified view of performance heterogeneity as compared to a select instance of heterogeneity.

Scheduling for heterogeneity. Several past works focus on levering heterogeneity in scheduling. Recent interest in asymmetric multicore processors raises the issue within the OS (*e.g.*, [27]), while others have looked at accommodating heterogeneity in MapReduce clusters [40, 37]. One approach is to benchmark machines initially, then use this in-

formation for load balancing or job placement [10]. However, these systems all assume that an application must use the set of machines it is given and cope with the heterogeneity. Furthermore, much of the focus is on performance differences leading to stragglers. In contrast, our work focuses on dropping poor performing machines rather than adjusting their workload. This is possible in a cloud environment, but may not make sense in cluster environment, where the machines have already been paid for.

Other efforts at managing heterogeneity include improved isolation (*e.g.*, [31]). While this can address temporal variation, it does not address inherent performance difference between machines. Other works evaluate using virtual machine migration for job scheduling [19] in the cloud, but rely on the cloud provider. In contrast, we present mechanisms for cloud customers to manage their own scheduling.

Formal models. Recent work [39] builds models to show the effects of heterogeneity and demonstrates limits on the useful range of performance. Their work demonstrates that many applications perform poorly when performance differences grow too large, which motivates the use of our techniques to ensure a more uniform pool of machines.

The scheduling problem we address is related to the multi-armed bandit problem [6, 35], in which a player must choose which slot machines to play based on the unique rewards of each machine. Our scheduling model in Section 3 is inspired by this work, but we use it largely as a framework for developing strategies rather than to analyze their outcomes.

8. CONCLUSION

Cloud computing environments will inevitably demonstrate some level of performance heterogeneity due to hardware variation and competing workloads. Our work shows how cloud customers can deliberately guide the placement of their workloads to improve their performance by selectively switching workloads off poorly performing machines.

Our work opens up a host of interesting questions. Most immediately: how can the models and simple strategies we introduce be improved? And, can controllers be built that provide improvements for other workloads? Should placement gaming become prevalent, can providers prevent or, conversely, better facilitate customer-controlled placement gaming? In the former context, widespread gaming might disrupt a provider's operations by introducing extraneous load, such as use of network-storage bandwidth to launch additional short-lived instances. Providers might respond with added fees for the first hour or raising the minimum instance lifetime beyond an hour to try to make gaming more expensive. Alternatively, cloud providers could recognize the interests of customers and provide explicit support for placement, such as hints about the desired architectures or resources needed (disk, network) so as to remove the need for customers to game the system.

9. REFERENCES

[1] Amazon Ltd. Amazon elastic compute cloud (EC2). `http://aws.amazon.com/ec2/`.

[2] Amazon Web Services. Amazon EC2 instance types. `http://aws.amazon.com/ec2/instance-types/`.

[3] Amazon Web Services. Amazon Elastic Block Store (EBS). `http://aws.amazon.com/ebs/`.

[4] Apache Software Foundation. ab - apache http server benchmarking tool. `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[5] Apache Software Foundation. Apache HTTP server project. `http://httpd.apache.org/`.

[6] Manjari Asawa and Demosthenis Teneketzis. Multi-armed bandits with switching penalties. *IEEE Transactions on Automatic Control*, 41(3), March 1996.

[7] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *MMsys*, pages 35–46, 2010.

[8] Russker Coker. Bonnie++ benchmark version 1.03e. `http://www.coker.com.au/bonnie++/`, 2008.

[9] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Ec2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the 2009 international conference on Service-oriented computing*, pages 197–207, 2009.

[10] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Resource provisioning of web applications in heterogeneous clouds. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 5–5, 2011.

[11] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware*, 2006.

[12] J. L. Henning. Spec cpu2006 benchmark descriptions. In *SIGARCH Computer Architecture News*, 2006.

[13] Alexandru Iosup, Nezih Yigitbasi, and Dick H. J. Epema. On the performance variability of production cloud services. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 104–113, 2011.

[14] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. Cruise: cache replacement and utility-aware scheduling. In *ASPLOS*, 2012.

[15] Vivek Kale, Jayanta Mukherjee, and Indranil Gupta. Hadoopjitter: The ghost in the machine and how to tame it. `http://hdl.handle.net/2142/17084`, 2010.

[16] Yaakoub El Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. Exploring the performance fluctuations of hpc workloads on clouds. In

CloudCom, pages 383–387, 2010.

[17] Dan Klein, Joseph Smarr, Huy Nguyen, and Christopher D. Manning. Named entity recognition with character-level models. In *Proceedings the Seventh Conference on Natural Language Learning*, pages 180–183, 2003.

[18] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: Comparing public cloud providers. In *IMC*, 2010.

[19] Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, and Chita R. Das. Migration, assignment, and scheduling of jobs in virtualized environment. In *HotCloud*, 2010.

[20] Dave Mangot. EC2 variability: The numbers revealed. `http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_revealed`, May 2009.

[21] Microsoft Corp. Windows azure: Pricing details. `http://www.windowsazure.com/en-us/pricing/details/`.

[22] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of amazon EC2. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[23] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[24] Rackspace Inc. How we price cloud servers. `http://www.rackspace.com/cloud/cloud_hosting_products/servers/pricing/`.

[25] M.S. Rehman and M.F. Sakr. Initial findings for provisioning variation in cloud computing. In *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 473 –479, 2010.

[26] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off my cloud: exploring information leakage in third party compute clouds. In *CCS*, 2009.

[27] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proc. EuroSys*, 2010.

[28] Jorg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. In *VLDB*, September 2010.

[29] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. The seven deadly sins of cloud computing research. In *HotCloud*, June 2012.

[30] Burr Settles. Biomedical named entity recognition using conditional random fields and rich feature sets. In *NLPBA*, 2004.

[31] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[32] The Stanford Natural Language Processing Group. Stanford named entity recognizer (NER) version 1.2.4. `http://nlp.stanford.edu/software/CRF-NER.shtml`, 2012.

[33] A. Tirumala, F. Qin, J Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool, version 2.0.5. `http://sourceforge.net/projects/iperf/`, 2010.

[34] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon EC2 data center. In *IEEE INFOCOM*, 2010.

[35] P. Whittle. Sequential scheduling and the multi-armed bandit, chapter 14. In *Optimization over Time–Dynamic Programming and Stechastic Control. vol. 1*, pages 210–219. John Wiley and Sons Ltd., 1982.

[36] Wikipedia. Markov decision process. `http://en.wikipedia.org/wiki/Markov_decision_process`.

[37] J. Xie, S. Yin, X.-J. Ruan, Z.-Y. Ding, Y. Tian, J. Majors, , and X. Qin. Improving MapReduce performance via data placement in heterogeneous Hadoop Clusters. In *Proc. 19th Int'l Heterogeneity in Computing Workshop*, April 2010.

[38] Jun-Ming Xu, Kwang-Sung Jun, Xiaojin Zhu, and Amy Bellmore. Learning from bullying traces in social media. In *NAACL HLT*, 2012.

[39] Sungkap Yeo and Hsien-Hsin S. Lee. Using mathematical modeling in provisioning a heterogeneous cloud computing environment. *IEEE Computer*, 44(8), August 2011.

[40] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.

[41] Ce Zhang, Feng Niu, Christopher Ré, and Jude Shavlik. Big data versus the crowd: Looking for relationships in all the right places. In *ACL*, 2012.