Towards VM Consolidation Using a Hierarchy of Idle States

Rayman Preet Singh, Tim Brecht, and S. Keshav

Cheriton School of Computer Science, University of Waterloo

{rmmathar, brecht, keshav}@uwaterloo.ca

Abstract

Typical VM consolidation approaches re-pack VMs into fewer physical machines, resulting in energy and cost savings [13, 19, 23, 40]. Recent work has explored a just-in-time approach to VM consolidation by transitioning VMs to an inactive state when idle and activating them on the arrival of client requests [17, 21]. This leads to increased VM density at the cost of an increase in client request latency (called *miss penalty*). The VM density so obtained, although greater, is still limited by the number of VMs that can be hosted in the one inactive state. If idle VMs were hosted in *multiple* inactive states, VM density can be increased further while ensuring small miss penalties. However, VMs in different inactive states have different capacities, activation times, and resource requirements.

Therefore, a key question is: How should VMs be transitioned between different states to minimize the expected miss penalty? This paper explores the hosting of idle VMs in a hierarchy of multiple such inactive states, and studies the effect of different idle VM management policies on VM density and miss penalties. We formulate a mathematical model for the problem, and provide a theoretical lower bound on the miss penalty. Using an off-the-shelf virtualization solution (LXC [2]), we demonstrate how the required model parameters can be obtained. We evaluate a variety of policies and quantify their miss penalties for different VM densities. We observe that some policies consolidate up to 550 VMs per machine with average miss penalties smaller than 1 ms.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design, Distributed Systems

Keywords Virtualization, virtual machines, cloud computing, VM density, VM consolidation, VM hierarchy

VEE '15, March 14–15, 2015, Istanbul, Turkey..
Copyright © 2015 ACM 978-1-4503-3450-1/15/03…$15.00.
http://dx.doi.org/10.1145/2731186.2731195

1. Introduction

Virtual machine (VM) consolidation [13, 19, 23, 40] allows cloud-providers to pack multiple VM instances running on underutilized physical machines into fewer machines, enabling some machines to be turned off, resulting in energy and cost savings. This allows cloud-providers to minimize costs, and maximize profits, while continuing to meet SLAs.

Typical VM consolidation approaches simply re-pack VMs into fewer physical machines using VM migration. The VM density (average number of VMs/machine) such methods yield is bounded by the maximum number of VMs that can be co-hosted on a single machine. However many VM workloads exhibit frequent, often long, and uncorrelated idle periods. Examples of such workloads include certain web-hosting workloads [21], cyber-foraging workloads [28], and personal servers [14, 25, 31]. When multiple VMs with such workloads are co-hosted on a machine, decreasing the resource footprint of idle VMs allows for a much denser VM packing, thus reducing hosting costs for both tenants and cloud-providers. For such workloads, it is possible to reclaim resources from idle VMs by transitioning them to inactive state(s), and activating them on the arrival of client requests. Table 1 shows inactive states proposed or supported in a few virtualization solutions. Such a consolidation effort is compatible with existing migration-based consolidation methods since it incurs little network overhead. Moreover, this type of consolidation is able to leverage transient idle periods to reclaim resources whereas conventional VM migration-based consolidation methods rely solely on long idle periods.

Virtualization Solution	Inactive states
LXC [2]	Frozen(stock) [24], Shutdown (stock)
Xen	Suspended, Shutdown (stock), Substrates [35]
VMWare ESXi	Suspended, Shutdown (stock), Fast-resume [38, 39]
KVM	Suspended, Shutdown (stock)

Table 1: Proposed and natively supported (denoted "stock") inactive states for a few virtualization solutions.

Recent work [17, 21] has explored the use of one inactive state (e.g., substrate [35], fast-resume [38, 39]) for managing idle VMs. In doing so, VM density is limited by the maximum number of VMs (per machine) in that inactive state that can be hosted on a machine. If idle VMs were hosted in more than one such inactive state (Table 1), VM density can be increased. Unfortunately, due to differences in their design and resource requirements, different inactive states have varying VM activation and deactivation times, and VM capacities. Consequently, miss penalties for idle VMs in different inactive states vary significantly. This leads to the following questions: 1) How should idle VMs be transitioned across different inactive states? In other words, when a VM becomes idle which inactive state should it be transitioned to? 2) Subsequently, when should an idle VM be transitioned to other state(s) in anticipation of client requests? Therefore, what is needed is a policy that governs the transitions of idle VMs across inactive states so as to minimize the miss penalties and maximize VM density. We refer to these policies as *idle VM management policies*. Existing mechanisms (Section 2) can then be used to implement such policies and dynamically transition idle VMs across the inactive states.

In this work we study the effect of different idle VM management policies on VM density and miss penalties. First, we formally model the problem of multiplexing idle VMs across multiple inactive states. We divide the policy space into two parts, (i) *demand-based* (or reactive) policies, and (ii) *proactive* policies. Using our model formulation, we provide a lower-bound on the miss penalty incurred by demand-based policies. Then, by finding similarities between this problem and the problems of page replacement and multi-level cache management, we propose *SlidingWindow*, a proactive policy which leverages inter-arrival time prediction to further reduce miss penalties. We obtain the model parameters (i.e., inactive state capacities, and transition times) for LXC [2], a widely used OS-level virtualization solution. We then use the measured parameters to evaluate different reactive and proactive policies, while using *personal servers* as a sample low duty-cycle workload. Our evaluation shows that at low-to-medium VM densities, a simple proactive policy can deliver up to an order of magnitude lower average miss penalty than widely known reactive policies, whereas reactive policies perform better under higher VM density.

This work makes the following key contributions:

- We present a formal model for idle VM management policies, and provide a lower bound on the miss penalty of reactive policies.
- With LXC [2] as our example virtualization solution, we demonstrate the measurement of model parameters using microbenchmarks.
- We study a few representative VM management policies, quantify their miss penalties using a simulation-based evaluation, and provide insight into their behaviour.

2. Background and Related work

Target Applications: Numerous rapidly-emerging applications are designed to execute (either completely or in part) on a per-user VM to provide a variety of services such as cloud-backed mobile applications (cyber-foraging) [17, 28], private data collection [16] and mining [25, 31], private online social networks [11], and other private VM-based applications [20]. Remote management of home sensors [10], and privacy-preserving community-wide sensing [14] are other examples. These VM-hosted applications service workloads where each VM is idle for large periods of time, and at any instant, only a small fraction (across a given number of VMs) are actively serving clients. Hence these idle periods can be leveraged to increase VM density and lower hosting costs. Similarly, recent work [21] has focused on *lower-end* consumers hosting user-facing services with frequent idle-periods, that can tolerate relinquishing of resources, when idle, in exchange for lower hosting costs.

Inactive states for VMs: Traditionally, VMs hosted on a machine are thought of as always being in a booted active state, and thus utilizing the host's CPU, memory, and disks [13, 19, 23, 40]. Inactive states are additional states providing a middle ground between the booted and shutdown states, where VMs consume only a fraction of resources of the booted state. This presents an opportunity for further increasing VM density. As different states utilize different amounts of each resource, it is possible to have a hierarchy of states where, at any instant, VMs actively serving workloads are in the booted state, while each idle VM is in one of the inactive states. Note also that a VM in the shutdown state only consumes the host's disk.

Recent work has proposed inactive states for VMs to reduce their resource footprint, albeit for different reasons e.g., for reducing VM activation time. Wang et al. [35] propose stateful in-memory *VM substrates* which are less resource-intensive than a running VM, and have small VM activation times. Knauth et al. [22] propose a fast-resume state which leverages lazy disk reads to lower VM activation time. Likewise, Twinkle [41] demonstrates the use of different optimizations–working set estimation, demand prediction, and free page avoidance, to lower VM resume (from suspended) times. This body of existing work focuses on reducing resource footprint of inactive VM states and/or reducing activation times, and does not study the multiplexing of VMs across inactive states, and its impact on miss penalties and VM density. To our knowledge, no prior work has studied the design of policies to transition VMs between different states, even for a two-state hierarchy.

Just-in-time provisioning of VMs: Existing work has explored using one inactive state for hosting idle VMs. DreamServer [21] demonstrates the use of a lazy, eager, and hybrid VM resume [22, 38, 39] for just-in-time provisioning of VMs for web-hosting workloads exhibiting idle periods. Similarly, Ha et al. [17] explore just-in-time VM provisioning for offloading computation from a mobile device. This body of existing work leverages only a single inactive state, which limits VM density to the maximum number of inactive VMs that can co-exist in that particular state. Since inactive states differ in their resource requirements, VM density can be improved by multiplexing idle VMs across multiple inactive states, which is the focus of our work.

Existing work has also demonstrated activation of VMs on request arrival, using different mechanisms, such as a reverse-proxy server running on the host [21]. Other possible mechanisms include using a DNS server running on the host [32], or a host-kernel module which uses the target IP address in a request to identify and activate the target VM.

Determining VM idleness: The amount of time a VM is idle depends entirely on the workload it serves. For instance, if a VM hosts application servers, once an active VM stops serving clients, it can be classified as being idle. Existing work has shown how such rules to determine VM idle time can be created. For instance, in case of server-client workloads such rules can use the number of connected TCP clients [32], or can use CPU and memory utilization [37] (for a variety of workloads, e.g., involving server-client jobs, or VM-initiated jobs). Example mechanisms to implement such rules include *VM introspection* [18, 32, 37], and *client-request monitoring* using a DNS server [32].

3. Model Formulation

Once a VM becomes idle it can either be left in the booted state, or can be transitioned (either immediately or at a suitable later time) to one of the inactive states, depending on the VM management policy in place. Since different inactive states, due to differences in their design and resource intensiveness, have different transition-to-booted times, the policy's actions can significantly affect miss penalties for subsequent requests for this VM. Similarly, because the maximum number of VMs possible in each state is limited (typically by a certain system resource), the policy's actions may also affect miss penalties for other VMs depending on which state it chooses to place them in. Due to such implications on miss penalties, it is important to choose a VM management policy which minimizes miss penalties across all VMs while maximizing VM density.

Many cloud environment workloads today exhibit a great deal of VM heterogeneity. VMs may have varying resource demands, workloads, and SLAs. In this work, as a starting point, we study scenarios where VMs are relatively homogeneous in resource requirements, workloads, and SLAs (e.g., VMs for personal servers in Section 2). We defer the study of heterogeneous scenarios to future work.

To better understand and reason about different possible policies, we formulate a simple mathematical model of the problem. Such a formulation gives a sound theoretical foundation to the problem, and as we show, can be used to provide a lower bound on the miss penalty incurred by any demand-based policy.

Let $S_1, S_2 \ldots S_n$ (as shown in Figure 1) be the n inactive states, in addition let S_0 be the booted state, and S_{n+1} be the shutdown state. Similarly, let $V_1, V_2 \ldots V_v$ be v VMs provisioned on a machine. Let the maximum number of VMs feasible in state S_i be B_i. Let matrix $T_{(n+2)\times(n+2)}$ be the time to transition VMs across different states, i.e. $t_{i,j}$ is time to transition from S_i to state S_j where $i, j \in \{0, 1, \ldots n, n+1\}$. We realize that in practice, the transition times are stochastic variables (associated with some distribution). Our model can be viewed as a mean value analysis. We view B_i as a soft bound, i.e., if the number of VMs in S_i exceeds B_i, transition times ($\forall j$, $t_{i,j}$ and $t_{j,i}$) may degrade. To simplify the notation in a later proof, let t_i be the time to transition from S_i to S_0 (booted), i.e. $t_i = t_{i,0}, i \in \{0, 1, \ldots n, n+1\}$.

Let VM requests received over a time period t be denoted as $\omega = r_1, r_2 \ldots r_t$, a string of tuples r_i, where $r_i = (V_j, d_i)$ where V_j, and d_i denotes the duration for which V_j is active (serving request r_i). We refer to ω as the *request string*. For a given ω, let $P_\pi(\omega)$ denote the total miss penalty incurred by VM management policy π. Thus for an optimal VM management policy θ, $\forall\omega\forall\pi, P_\theta(\omega) \leq P_\pi(\omega)$.

Policies can be divided into two classes: i) reactive (or demand-based) policies, and ii) proactive policies. We now address each class separately and describe how VM management policies are related to other types of resource management policies.

3.1 Reactive policies

Reactive policies configure, transition, or provision a system resource, such as a memory page, only when a demand for that resource is received, and are also referred to as *demand-based* policies. Examples include the widely used demand-based page replacement policies such as LRU, FIFO, Clock, and others [12]. Another prominent example is multi-level cache management policies such as DEMOTE [36]. Belady's OPT algorithm [9], which simply evicts the page referenced furthest in the future, is provably the optimal demand-based policy for single-level caches [7, 15] (e.g., page replacement). However the optimal demand-based policy for multi-level caches remains unknown [15]. Nevertheless, we build upon existing results on page replacement and caching policies and formulate a lower bound on the miss penalty $P_\pi(\omega)$ incurred by any demand-based VM manage-

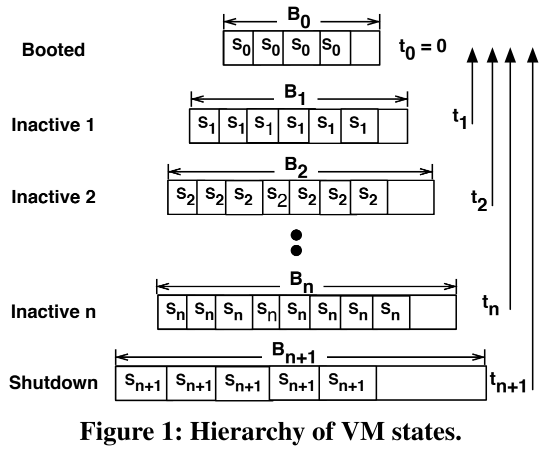


Figure 1: Hierarchy of VM states.

ment policy π. Policies can be either *online*, which only use information about the past, or *offline*, which also use information about the future. Our lower bound assumes knowledge of future arrivals and cannot be implemented in an online fashion. It serves as a theoretical lower bound for comparison with other demand-based policies.

For any given ω, a VM management policy π is a demand-based policy if, π transitions a VM V_i to the booted state at time t (if not already booted), only if $\exists r_t \in \omega$, such that, $r_t = (V_i, d)$ for some duration d. For any given ω, we define $h_i, i \in \{0, 1, \dots n, n+1\}$ to be the number of requests $r_t \in \omega$, such that the target VM was in state S_i on arrival of the request. On arrival of the request r_t the target VM is transitioned to the booted state from its current state. Since the maximum number of VMs in each state is limited (B_i), this transition *may* require additional transitions for (a) transitioning other VMs from the booted state into inactive states, and (b) transitioning other VMs from one inactive state to another. Even if all transitions are conducted in parallel, these additional transitions, depending upon their duration, may contribute towards increasing the miss penalty. For instance, an additional transition may take more time than the time to transition the target VM to the booted state. Thus for any demand-based VM management policy π,

$$P_\pi(\omega) \geq Min_\pi(\omega),$$

where $Min_\pi(\omega) = \sum_{i=0}^{n+1} h_i.t_i$.

3.1.1 Lower bound on demand-based policies

Gill et al. [15] propose a demand-based multi-level cache management policy which provides the lowest average response time and lowest inter-cache bandwidth. Along the same lines, we define a VM management policy ϕ such that

$$\forall \pi, \forall \omega, Min_\phi(\omega) \leq Min_\pi(\omega),$$

and hence, $\forall \pi, Min_\phi(\omega) \leq P_\pi(\omega)$. That is $Min_\phi(\omega)$ forms a lower bound on the total miss penalty for any demand-based VM management policy π, for any request string ω for length $|\omega|$.

We now compute the lower bound $Min_\phi(\omega)$. First, consider a state hierarchy which consists only of the booted and shutdown states (i.e., $n = 0$). Consider the application of Belady's OPT algorithm on this hierarchy for managing VMs, i.e., when a VM needs to be transitioned to the booted state and the number of booted VMs equals B_0, the booted idle VM which will receive a request farthest in the future is shut down. Using this algorithm on a two-state hierarchy, for any given ω, let the optimal number of hits $hOPT(\omega, B_0)$ be the number of requests in ω such that the target VM is in the booted state (S_0) on arrival of the request, where B_0 is the maximum number of VMs possible in the booted state.

Let ϕ be a demand-based VM management policy which, for a reference string ω, for each state S_i ($i \in 0, 1, \dots n$), exhibits h_i (number of hits to state S_i), such that

$$h_i = hOPT(\omega, \sum_{j=0}^{i} B_j) - hOPT(\omega, \sum_{j=0}^{i-1} B_j). \qquad (1)$$

We show that amongst all demand-based policies, ϕ minimizes $Min_\pi(\omega)$, and hence $Min_\phi(\omega)$ forms the lower bound on the total miss penalty of any demand-based policy.

Lemma I Among all demand-based policies, policy ϕ maximizes $\sum_{i=0}^{k} h_i, \forall k \in \{0, 1 \dots n\}$.

Proof: Summing (1) over the range $i = 0, 1 \dots k$ we get,

$$\sum_{i=0}^{k} h_i = hOPT(\omega, \sum_{j=0}^{k} B_j).$$

This is the same as operating Belady's OPT algorithm on a hierarchy with just two states– booted and shutdown, with the maximum possible number of VMs in booted equal to $\sum_{j=0}^{k} B_j$. Since Belady's algorithm is known to be optimal demand-based policy on a two-state hierarchy, $\sum_{i=0}^{k} h_i$ is maximized.

Lemma II For any ω, no other demand-based policy π has $Min_\pi(\omega) < Min_\phi(\omega)$.

Proof: We prove by contradiction. Let $\widehat{\pi}$ be a demand-based policy (with respective $\widehat{h_i}$), such that $Min_{\widehat{\pi}}(\omega) < Min_\phi(\omega)$. Therefore,

$$\sum_{i=0}^{n+1} \widehat{h_i}.t_i < \sum_{i=0}^{n+1} h_i.t_i$$

$$\text{Or, } \sum_{i=0}^{n} \widehat{h_i}.(t_i - t_{n+1}) + (\sum_{i=0}^{n+1} \widehat{h_i}).t_{n+1} < \sum_{i=0}^{n} h_i.(t_i - t_{n+1}) + (\sum_{i=0}^{n+1} h_i).t_{n+1}.$$

Since $\sum_{i=0}^{n+1} \widehat{h_i} = \sum_{i=0}^{n+1} h_i = |\omega|$,

$$\sum_{i=0}^{n} \widehat{h_i}.(t_{n+1} - t_i) > \sum_{i=0}^{n} h_i.(t_{n+1} - t_i). \qquad (2)$$

$$\text{Or, } \sum_{i=0}^{n} \widehat{h_i}.(t_n - t_i + t_{n+1} - t_n) > \sum_{i=0}^{n} h_i.(t_n - t_i + t_{n+1} - t_n).$$

$$\text{Or, } \sum_{i=0}^{n} \widehat{h_i}.(t_n - t_i) > \sum_{i=0}^{n} h_i.(t_n - t_i) + (\sum_{i=0}^{n} h_i - \sum_{i=0}^{n} \widehat{h_i}).(t_{n+1} - t_n).$$

The second term on right hand side is non-negative because $t_{n+1} \geq t_n$, and by Lemma I, $\sum_{i=0}^{n} h_i \geq \sum_{i=0}^{n} \widehat{h_i}$. This means,

$$\sum_{i=0}^{n} \widehat{h_i}.(t_n - t_i) > \sum_{i=0}^{n} h_i.(t_n - t_i).$$

Since the nth term in the summation on either side is zero, we get,

$$\sum_{i=0}^{n-1} \widehat{h_i}.(t_n - t_i) > \sum_{i=0}^{n-1} h_i.(t_n - t_i). \qquad (3)$$

In reducing (2) to (3), the summation reduces from n to $(n-1)$. Since $\forall j \in \{1, \ldots n\}, t_{j-1} \leq t_j$, these steps can be repeated until the summation reduces to $n=1$. That is, $\widehat{h_0}.(t_1 - t_0) > h_0.(t_1 - t_0)$, which implies $\widehat{h_0} > h_0$, which contradicts Lemma I (which states that ϕ maximizes $\sum_{i=0}^{k} h_i, \forall k \in \{\mathbf{0}, 1, \ldots n\}$).

Note that the lower bound $Min_\phi(\omega)$ assumes future knowledge and cannot be implemented in an online fashion. Nevertheless, it serves as theoretical lower bound for comparing with other demand-based policies. In Section 5, we compare a few widely used demand-based policies with the lower bound and compare them with proactive policies.

3.2 Proactive Policies

Proactive policies configure, transition, or provision a system resource prior to a demand for it being received. They have previously been explored in the context of page replacement, with the goal of producing lower page faults than demand-based page replacement. Existing work [34] has shown that Belady's OPT algorithm is the optimal demand-based policy that minimizes the number of page fetches but does not minimize the number of page faults, because it does not prefetch pages. Trivedi et al. [34] explored the use of proactive policies to lower the number of page faults, and proposed and proved DPMIN [34] as the optimal pre-paging algorithm. DPMIN proceeds as follows: at the time of a page fault, DPMIN scans the future page reference string and fetches the first m pages that will be referenced in the future (including the page that caused the page fault), where m is the number of memory page frames. For the purposes of this paper, we define *proactive policy* as any policy that is not a demand-based policy (as defined above).

SlidingWindow: We extend the DPMIN algorithm and define the SlidingWindow policy for managing VMs across different states. Our rationale is that, since VMs can be transitioned in parallel, provisioning VMs in anticipation of requests would reduce average miss penalties. SlidingWindow proceeds as follows: whenever a request $r_t \in \omega$ is received such that the target VM is not in the booted state, it computes a new state configuration for all VMs. To compute this new configuration, SlidingWindow scans the future reference string (illustrated in Figure 2). All VMs that are currently active are placed in the booted state in the new configuration. If A is the number of currently active VMs, the first $(B_0 - A)$ VMs that will be requested in the future are placed into the booted state (S_0) (illustrated as a time window W_0) in the new configuration. Similarly, the next B_1 VMs that will be requested next are placed into S_1 (illustrated as time window W_1). This process continues up to state S_n (time window W_n), and the remaining $(v - \sum_{i=0}^{n} B_i)$ VMs are placed in the shutdown state. After the new configuration is computed, VMs are moved from their existing to this new configuration.

When VM V_i becomes idle, SlidingWindow re-scans the future reference string to compute t_{next}, i.e., the time at which V_i will be requested next. If t_{next} falls in the time window W_j it transitions V_i into S_j, and one VM from S_k to S_{k-1} (the one that is referenced the soonest) $\forall k \in \{j, j-1, \ldots 1\}$. In effect, it slides the windows $W_0, W_1 \ldots W_{j-1}$ to the right. If t_{next} falls in the window W_0, V_i remains in the booted state.

Similar to DPMIN, our SlidingWindow policy assumes knowledge of the future reference string, and cannot be implemented in an online fashion. Therefore, in addition to the offline implementation of the policy, we provide an online implementation (called *SlidingWindow+ARMA*) which uses a predictor to estimate t_{next}, and uses the predicted value to perform its proactive VM provisioning. We describe SlidingWindow+ARMA in further detail in Section 5.2, and compare with demand-based policies in Section 5.5.

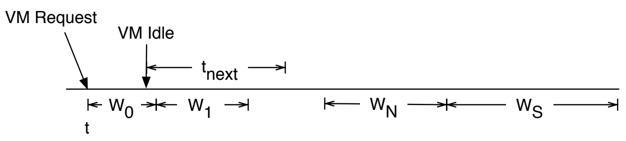


Figure 2: Proactive management, SlidingWindow policy.

4. Obtaining Model Parameters

Our formal model (described in Section 3) relies on a few input parameters, namely the transition times $(T_{(n+2)\times(n+2)})$ and the maximum number of VMs possible in each state (B_i). We use LXC [2], a widely used OS-level virtualization solution, as an example virtualization solution, and conduct an experimental analysis to obtain the model parameters. Note that we continue to refer to execution environments created using LXC as VMs, since our formal model (Section 3) is applicable to any virtualization solution. We now describe our methodology in detail and justify our choices.

4.1 LXC as a Case Study

OS-level virtualization approaches have been shown to incur 40-50% lower virtualization overhead than other approaches such as Xen paravirtualization [27, 33], thus promising potentially higher density. Example OS-level virtualization solutions include LXC [2], OpenVZ [3], and VServer [5]. We choose LXC as our example virtualization solution for several reasons: i) it is open source, allowing easy analysis of its implementation, ii) it is in production use and is part of the mainstream Linux kernel, iii) it offers a low latency inactive VM state called "frozen", iv) it is being used in other projects which can benefit from increased VM density, such as Docker [1] (to provide "frozen in state apps"), and Confidential Commuting [14] (to provide per-user private VEEs).

As noted, in addition to the booted (S_0) and shutdown state (S_2), LXC implements a *frozen* state (S_1) which forms a middle ground between booted and shutdown states. When an idle VM in the booted state is transitioned to the frozen state, it retains its memory and disk footprint, while relinquishing CPU cycles. In addition, frozen-to-booted transition times are significantly smaller than shutdown-to-booted transitions. Thus, LXC provides us with a three-state hierarchy with the booted, frozen, and shutdown states. Menage [24] provides a detailed description of the implementation of the frozen state.

4.2 Experimental Setup

We use LXC [2] (v.0.8.0) to create VMs. VMs are hosted on a machine which has four Intel Xeon processors with six 3.46 GHz cores each, and 128 GB RAM. It uses a 7200 RPM 1 TB SATA hard disk to store VMs OS images. All experiments are repeated 50 times and experiment results are reported using averages with 95% confidence intervals.

VMs in an OS-level virtualization solution, such as LXC, share the host kernel's process pool, data structures, and devices. Therefore, when increasing VM density, some host kernel parameters need to be increased. For instance, since all processes of all LXC VMs share the host kernel's pool of open file descriptors (FD) the total number of open FDs allowed by the host kernel limits the number of VMs. Similarly, the kernel's maximum allowed process identifier (PID) (set to a default of 32,767), and the number of Unix98 pseudoterminals (set to a default of 4,096), also limit the number of VMs. For conducting our experiments we increased these values to 14,000,000, 65,535, and 8,000 respectively.

4.3 Quantifying Density

We first determine the maximum number of booted, frozen, and shutdown VMs that can be supported in our testbed.

Shutdown VMs: The only system resource consumed by shutdown VMs is disk space. It is required for storing their OS image, applications, and libraries. In our testbed, each VM consumes 476 MB of disk space. Thus, 2,000 VMs can be created on a 1 TB disk, using the EXT4 filesystem.

Booted VMs: To determine the maximum number of booted VMs (B_0) we conduct a simple experiment where the number of booted but idle VMs on the machine is gradually increased, while all other VMs are shutdown. VMs are transitioned from shutdown to booted sequentially, with a delay of 30 seconds between successive VMs to ensure that the system reaches a steady state; essential for recording system measurements. Measurements are recorded using *vmstat*.

We observe that the system memory consumption increases steadily with increasing number of VMs. This is because each additional VM consumes approximately 42 MB of memory for initializing its processes (such as its SSH server, and other daemons). Figure 3 shows the CPU system time (time spent running kernel code), and CPU idle time (time spent idle) averaged over all 24 cores on the server machine, with increasing number of booted idle VMs. We observe that up to 250 VMs, the CPU is largely idle. This is because idle VMs do not perform any significant computation, and only a few VM processes (such as udevd, and other daemons) are running, causing a small increase in CPU idle time, while other processes remain blocked. Note that, all VM processes are in user space. However, beyond 250 VMs, we observe an increase in CPU system time, eventually reaching 100% at 450 VMs. At this stage, no additional VMs can be booted as all 24 cores are completely busy. We believe that this is because of the inability of LXC's cgroup handler [24] to scale with increasing number of processes. LXC uses the cgroup handler for bookkeeping of resources used by processes of different VMs and to maintain isolation. As a result, this limits the number of booted idle VMs to approximately 250.

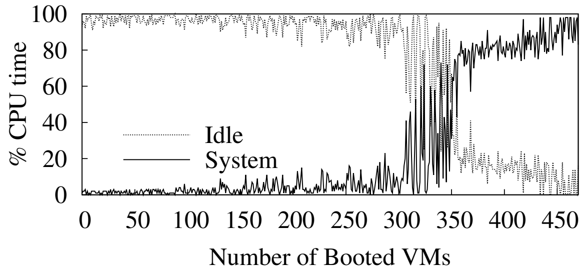


Figure 3: CPU utilization versus booted VMs.

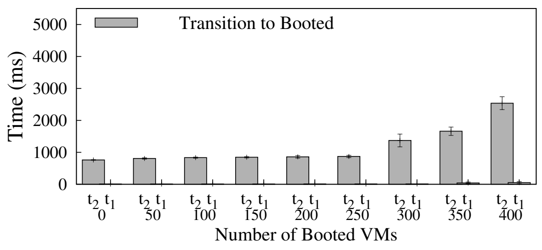
Frozen VMs: We now determine the maximum number of frozen VMs (B_1), by conducting a simple experiment where the number of frozen VMs on the machine is gradually increased. Each VM is booted, allowed to initialize, and then transitioned to frozen. To determine when a VM has finished booting, we use *netcat* to detect if the VM's SSH server has started. We measure the time taken to boot-up the VM and start its SSH server. All other VMs remain in the shutdown state.

We observe a steady increase in the system memory consumption (at approximately 42 MB per VM; figure omitted due to space constraints). This is because processes in a frozen VM retain their memory footprint (due to the absence of memory pressure). Moreover, we observe that the time to transition a VM from shutdown to booted (before transitioning to frozen) increases considerably as the number of frozen VMs on the machine increases. This is because many system calls used by LXC for booting a VM, such as fork, wait, and open, take more time to complete, as the number of frozen VMs increases. As described in Section 4.4, this increase in transition time eventually limits the number of frozen VMs (to approximately 300 frozen VMs).

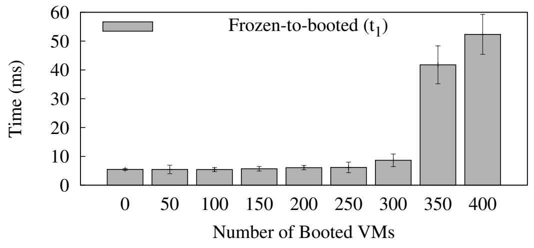
4.4 Impact of Density on Transition Time

We study the effect of VM density in each state on the transition times. We vary the number of booted VMs and measure the different state transition times (while keeping the number of frozen VMs at zero). Similarly we vary the number of frozen VMs and measure the transition times.

Varying Number of Booted VMs: The number of booted VMs is increased in steps of 50. At each step the different transition times are measured (for a given VM). All other VMs are in the shutdown state.



(a) Transition times for Shutdown-to-Booted (t_2) and Frozen-to-Booted (t_1).



(b) Transition times for Frozen-to-Booted (t_1).

Figure 4: Transition times with increasing number of booted VMs.

Figure 4a shows the shutdown-to-booted (denoted t_2) and frozen-to-booted (denoted t_1) transition times. We observe that up to 250 VMs, the shutdown-to-booted transition time remains largely constant. However, beyond 250 booted VMs we see a considerable increase in boot-up time. This is due to a surge in CPU system time at 300 VMs and beyond (as explained in Section 4.3, Figure 3), which reduces available CPU time to zero. Figure 4b shows the frozen-to-booted transition time (denoted t_1 in Figure 4a), with a magnified time axis. We observe that frozen-to-booted transition times increase only slightly with increasing number of booted VMs, and is considerably smaller than shutdown-to-booted transition time (because frozen-to-booted is a comparatively faster and less resource-intensive transition). The abrupt rise beyond 300 booted VMs, is attributed to the surge in CPU system time, as explained in Section 4.3. Along the same lines we also measure the booted-to-shutdown ($t_{0,2}$) and booted-to-frozen ($t_{0,1}$) transition times with increasing number of booted VMs. However, we do not observe any significant variation in these transition times, remaining constant at approximately 640 ms and 0.15 ms respectively (not shown).

Varying Number of Frozen VMs: In a fashion similar to the previous experiment, the number of frozen VMs is gradually increased, and the different transition times are measured.

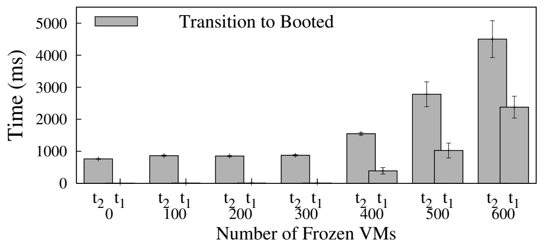
Figure 5a shows the shutdown-to-booted (denoted t_2) and frozen-to-booted (denoted t_1) transition times. We observe that the shutdown-to-booted transition time increases only slightly up to 300 frozen VMs, and are considerably larger beyond that point. This is because, as discussed in Section 4.3, certain system calls used by LXC for booting, require more time to complete when the number of frozen VMs increases, leading to increased transition times. Figure 5b shows the frozen-to-booted transition times (denoted as t_1 in Figure 5a) with a magnified time axis. We observe that below 300 VMs, the transition time is largely constant. However, beyond 300 frozen VMs, the transition time increases considerably. This is because LXC's cgroup freezer [24] mechanism begins consuming more time for unfreezing a VM, and hence does not scale. Therefore, the number of frozen VMs is limited by this increased transition time, to approximately 300. As in the previous experiment, we find that the booted-to-shutdown ($t_{0,2}$) and booted-to-frozen ($t_{0,1}$) transition times do not change significantly with increasing number of frozen VMs (measured at approximately 642 ms, 0.17 ms respectively, which is nearly identical to their values in case of varying number of booted VMs).

4.5 Deriving Model Parameters

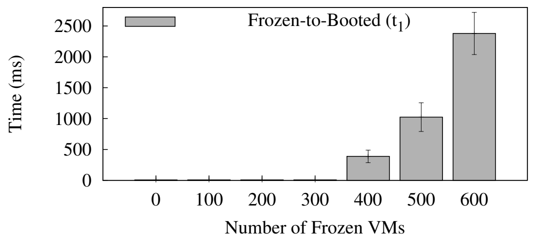
Using the experimental analysis (described above), we derive the model parameters as follows.

Due to observed density and transition time with increasing number of booted VMs (as explained above, Figures 3, 4), we define the maximum number of VMs possible in the booted state (B_0) to be approximately 250. Similarly, given the variation in transition times with increasing number of frozen VMs (Figure 5), the maximum number of VMs possible in the frozen state (B_1) is approximately 300. Note that both B_0 and B_1 values derive from limitations in LXC's design and implementation (described above). While it may be interesting to explore these limitations in greater detail (and alleviate them), it lies outside the scope of this work.

We compare the shutdown-to-booted (t_2) and frozen-to-booted (t_1) transition times in the two sensitivity analysis experiments (i.e., Figure 4a and Figure 5a). Interestingly, we find that within the operating range of up to 250 booted VMs, and up to 300 frozen VMs, the respective transition times in either experiments differ insignificantly. For instance, shutdown-to-booted transition time when varying only the number of booted VMs (up to 250 booted VMs, Figure 4a), is comparable to shutdown-to-booted transition time when varying only the number of frozen VMs (up to 300 frozen VMs, Figure 5a). Other transitions times (frozen-to-booted $t_{1,0}$, booted-to-frozen $t_{0,1}$, and booted-to-shutdown $t_{0,2}$) exhibit similar behaviour. Thus, we believe that within this operating range (up to 250 booted, 300 frozen VMs) all transition times remain largely constant, *regardless of the number of booted and frozen VMs*. Therefore, we represent the average transition times (within the operating range) as the transition matrix $T_{3\times3}$ (shown in Table 2, where $t_{i,j}$=time to transition from state S_i to S_j). To transition a frozen LXC VM to shutdown (and vice versa), it must first be transitioned to booted.



(a) Transition times for Shutdown-to-Booted (t_2) and Frozen-to-Booted (t_1).



(b) Transition times for Frozen-to-Booted (t_1).

Figure 5: Transition times with increasing number of frozen VMs.

From \ To	Booted (S_0)	Frozen (S_1)	Shutdown (S_1)	B_i
Booted (S_0)	0.00 ms	0.17 ms	641.64 ms	250
Frozen (S_1)	4.43 ms	0.00 ms	-	300
Shutdown (S_2)	802.16 ms	-	0.00 ms	2000 (Disk limited)

Table 2: Transition matrix ($T_{3\times3}$) and B_i values for LXC.

5. Policy Comparison

Using the parameters obtained above we study the effect of different policies using a simulation-based analysis, which makes exploring a large design space feasible.

In the next section, we discuss the design and implementation of our simulator, followed by the description of the policy implementations (Section 5.2), personal server workloads (Section 5.3), our evaluation metric (Section 5.4), and simulation results (Section 5.5).

5.1 Simulator Design and Implementation

Our simulator consists of a *policy module*, a *cost-model module*, and a *workload module*. The cost-model module encapsulates the VM hierarchy parameters (B_i and $T_{3\times3}$). The policy module encapsulates all logic pertaining to a particular policy and maintains any in-memory state required for implementing that policy, e.g., any per-VM bookkeeping. The workload module encapsulates all logic required for simulating the VMs' workloads such as distributions for request inter-arrival time and request durations. Such modularity allows us to easily extend the simulator to study different workloads, different VM management policies, as well as different VM hierarchies, simply by implementing the respective module. The simulator maintains the current state assignment for each VM. In addition, it contains a single time-sorted event queue, and a single thread which processes events in this queue. Events are either of type VMrequest or VMidle.

At initialization time, the workload module generates the simulated VMrequest events for the required number of VMs, and populates the event queue. When processing a VMrequest event, the simulator passes the current state assignment, the request event, and event queue to the policy module, and receives the updated state assignment. It then compares the updated state assignment with the current one, and computes a list of required VM transitions. It then computes the time required to perform the transitions, updates the state assignment, and enqueues a VMidle event with a later timestamp (derived using the request duration in the VMrequest event) into the event queue. Note that, all invocations of the policy module are serialized. For instance, if a VMrequest event occurs while the policy module is computing the updated VM state assignment, it is processed after the policy module finishes its computation. Lastly, when computing the time taken to perform a set of transitions, the simulator assumes that different VMs' transitions take place in parallel. This assumption is justified because in doing so we are able to measure the best case behavior of any policy. Any additional overhead in the system in performing parallel transitions would only serve to increase the transition times, and the miss penalty so obtained would still be bounded by the best case scenario. VMidle events are processed in the same fashion as VMrequest events. However, certain policies (e.g, demand-based policies) may choose to not take any action when a VM becomes idle.

Implementation: We have implemented the simulator using C# over the .NET v4.5 framework. We have implemented policy modules for different reactive and proactive policies, the workloads described in the next section, and a cost-model module for LXC. We validate the simulator using sample deterministic workloads and state hierarchies. The implementation is publicly available online [4].

5.2 Policy Implementations

LRU: Least Recently Used (or LRU) [12] is a policy that is widely studied for page replacement. We apply it to idle VM management in a cascaded fashion. For each VM, it maintains the timestamp of the last request (t_r). All VMs are initially in the shutdown state. As requests for different VMs start arriving, they are transitioned to the booted state. Eventually, as the number of VMs in the booted state (i.e., including active and idle VMs) reaches the limit B_0, for each VM transitioning into booted thereafter, LRU chooses to transition the booted VM with the minimum t_r into the frozen state. In effect, for each VM it uses the "time since last request" to estimate the likelihood that it will be requested again. Similarly, as the number of VMs in the frozen state reaches B_1, for each VM transitioning into frozen thereafter, the frozen VM with the minimum t_r is shut down. Note that, unlike traditional implementations of LRU (e.g., in page replacement) where all timing information of the resource is deleted after its eviction, we maintain t_r for each VM after eviction in order to apply it across multiple states.

Cascaded Belady's OPT: This policy is simply an extension of Belady's OPT algorithm to multiple states. We apply it to idle VM management. That is, when number of VMs in any state S_i exceeds B_i, the VM that is referenced the furthest in the future is transitioned to S_{i+1}. This version of Belady's OPT algorithm is known to suboptimal [15]. However, we implement it in order to compare it with demand-based policies that have no future knowledge (e.g., LRU) and the lower bound $Min_\phi(\omega)$. To the best of our knowledge, such a comparison has not been conducted in previous work.

Lower Bound on demand-based policies: As explained in Section 3.1.1, $Min_\phi(\omega)$ forms the lower bound on demand-based policies. We first obtain the h_0, h_1, h_2 values for LXC as per Eq. 1 for the different workloads we study (i.e., different ω values). For each ω, we obtain $h_0 = hOPT(\omega, B_0)$, $h_1 = hOPT(\omega, B_0 + B_1) - hOPT(\omega, B_0)$, and $h_2 = |\omega| - h_0 - h_1$. We then determine the lower bound $Min_\phi(\omega) = (h_0.t_0 + h_1.t_1 + h_2.t_2)$.

SlidingWindow: As explained in Section 3.2, our SlidingWindow policy requires knowledge of the future. Therefore we provide an online implementation of this policy (called *SlidingWindow+ARMA*) which implements a predictor to predict the inter-arrival time for each VM, and updates the prediction model at each request for that VM. We employ the widely-used *auto regressive moving average* (ARMA) time-series model for predicting the inter-arrival times. To find the order of the ARMA model we employ the Bayesian information criterion and at each model update we perform a maximum likelihood fit on the inter-arrival times. The policy then uses the predicted inter-arrival time for each VM to perform its proactive actions. Therefore, its miss penalty greatly depends on the prediction error of the ARMA model.

To evaluate SlidingWindow+ARMA, we also implement the offline version of SlidingWindow which uses knowledge of the future (called *SlidingWindow+GroundTruth*).

5.3 Workload Analysis: Personal Servers

In order to perform a comparison of the miss penalties incurred by different policies, we chose the personal server workload. As outlined in Section 2, in these workloads each user owns and controls a separate private VM, which hosts application instances that serve client requests from that user. We chose this workload because a) they are low duty-cycle in nature [32], i.e. private VMs have uncorrelated idle times, thus allowing greater multiplexing across inactive states, b) they are a topic of active research with numerous applications such as privacy-preserving online social networks [11, 29, 30], private sensor data collection and processing [14, 16, 31], and privacy-preserving offloading from mobile devices [17].

We use three categories based on the requests' inter-arrival and duration times. We believe that such categorization (as opposed to a mixed workload) allows us to better understand the behavior of different policies.

Fixed inter-arrival time, Fixed duration: A common use of personal VMs is for periodic and fixed amount of data uploads from in-home sensors [31], or a user's smartphone [14, 25]. This leads to a request sequence where requests have fixed inter-arrival times and durations. For such requests both the arrival and departure of requests are highly predictable, and form an interesting point of comparison between proactive and reactive policies.

Stochastic inter-arrival time, Fixed duration: Recent work has proposed using personal VMs as *virtual individual servers* [11, 29, 30] where users host their private instances of common application servers. Due to the user-facing nature of these application servers, the requests they receive have stochastic inter-arrival times, and commonly involve downloading or uploading fixed amounts of data, thus leading to requests with a relatively fixed duration.

Stochastic inter-arrival time, Stochastic duration: A new and evolving use of personal VMs is as personal data and compute environments, which host and process user data. Examples include private data analytics [31], VM-backed mobile applications [28], and other similar applications [20]. In these applications the requests are user-generated, and thus have stochastic inter-arrival times. In addition these requests have a varying nature, e.g., size of data processed, and type of computation performed, which results in them having stochastic durations.

5.4 Metric

In order to compare different policies for a request string ω, we use the *average miss penalty* incurred by any policy π.

$$\text{Average miss penalty} = \frac{P_\pi(\omega)}{|\omega|}.$$

This metric i) captures the miss penalty across all VMs in the system, ii) allows us to observe the behavior of any policy with increasing number of total VMs, and iii) allows us to easily observe the differences between reactive and proactive policies. Note that the lower bound on average miss penalty of reactive policies is simply $\frac{Min_\phi(\omega)}{|\omega|}$.

5.5 Simulation Results

We now study the effect of increasing VM density on the average miss penalty for different policies. The request string ω in all simulations contains 100 arrivals per VM, i.e., $|\omega|$=Total #VMs $\times$ 100.

In order to simulate stochastic inter-arrival times, we use the request inter-arrival times from publicly available web trace data [8], since it has been used in evaluating on-demand VM provisioning in existing work [21]. Similarly in order to simulate stochastic request durations, we use the web connection duration characterization provided by Newton et al. [26], since we believe it to be a representative request duration characterization for personal servers. Lastly, we use the respective mean values from the two datasets for generating the fixed inter-arrival time (of 50 s) and fixed duration (of 10 s) workload. Table 3 shows the variability of inter-arrival times for the three cases we study. We do not include details for duration because the policies we consider do not utilize duration (policies that do so will be considered in future work). In each workload experiment (below), we start the x-axis at 250 VMs since it is the maximum number of VMs possible in the booted state (and there are no miss penalties below 250 VMs). We increase VM density to the point that the total number of simultaneously active VMs increases above 250 (equal to B_0) and cannot be hosted using this hierarchy. Note that this limit is a result of the current workload (longer idle times would increase this limit).

Case		**Mean**	**Standard**	**Min**	**Max**
Duration	Inter-arrival		**deviation**		
Fixed	Fixed	50.0 s	0.0 s	50.0 s	50.0 s
Fixed	Stochastic	50.0 s	160.5 s	10.0 s	941.2 s
Stochastic	Stochastic	50.0 s	163.0 s	0.9 s	941.2 s

Table 3: Variation of inter-arrival times.

5.5.1 Fixed inter-arrival time, fixed duration

Figure 6 shows the average miss penalty for a request string ω with a fixed inter-arrival time of 50 seconds, and a fixed request duration of 10 seconds, for different policies with increasing VM density. Figure 6 also shows the shutdown-to-booted and frozen-to-booted transition times for comparison. For each VM, the time at which its first request is received is chosen uniformly at random from [0,50 s]. Later in this section, we analyze the behavior of policies under other inter-arrival time and duration values.

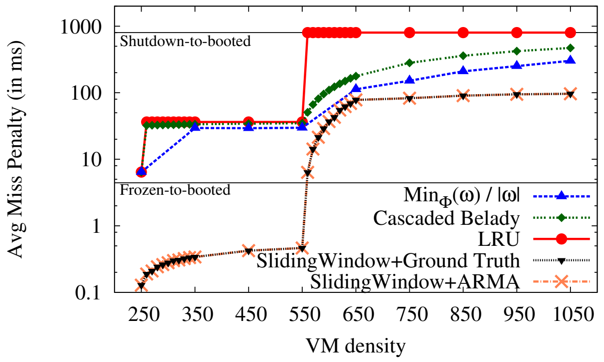


Figure 6: Fixed inter-arrival time and duration.

We first explain the behaviour of the reactive policies. We observe that at a VM density of 250, both reactive policies, LRU and Cascaded Belady's OPT, incur the same average miss penalty which matches the lower bound. This is because in this case, all reactive policies transition each VM into booted once its first request is received, and no idle VMs need to be transitioned out of the booted state thereafter, since VM density equals B_0. As VM density increases further, not all VMs can remain booted. Depending on the policy, some VMs are transitioned to frozen (when idle), and are transitioned to the booted state when their request arrives, which increases the average miss penalty. Similarly, as VM density increases beyond 550, not all VMs can be in either booted or frozen states. That is, all other VMs are transitioned to the shutdown state (when idle), and are brought into the booted state when their request arrives, which contributes to increase the average miss penalty. Since shutdown-to-booted transition times are significantly higher than frozen-to-booted (approximately 800 ms vs. 4 ms), we see a much larger increase in the average miss penalty at VM density ≥ 550 (than at 250 VMs). Note that, LRU has significantly higher average miss penalty than the cascaded Belady's algorithm because LRU has no knowledge of the future reference string. In case of LRU, when VM density is greater than 550, at each request the target VM is always in the shutdown state thus incurring the maximum miss penalty. This is because in this workload, between two consecutive requests to any VM V_i, there are (v-1) requests for other VMs (i.e., one per VM). Since LRU evicts the least recently used VM from booted to frozen, and frozen to shutdown, when $v > 550$, V_i will get transitioned to shutdown after the intermediate (v-1) requests are serviced.

We now explain the behavior of proactive policies. Due to easily predictable fixed request inter-arrival times and durations, the average miss penalty of the SlidingWindow policy using our ARMA predictor (denoted 'SlidingWindow+ARMA') equals that of SlidingWindow with future knowledge (denoted 'SlidingWindow+Ground Truth'). We observe that either policy incurs a significantly lower average miss penalty than the reactive policies. This is because whenever a request whose target VM is not in the booted state is received, in addition to transitioning that VM to booted, SlidingWindow also transitions other VMs (as many as possible) to booted and frozen states (as explained in Section 3.2). Note that the proactive transitions in SlidingWindow are triggered by a request whose target VM is not in the booted state. Therefore, as VM density increases up to 550, VMs span the booted and frozen states, and beyond 550, VMs span all three states. This increases the average miss penalty (since the shutdown-to-booted transition time is significantly larger than the frozen-to-booted time). Moreover, the number of idle VMs that SlidingWindow can proactively transition to booted, depends on the number of active VMs at that instant (since the number of active booted + the number of idle booted = 250). As VM density increases, the number of simultaneously active VMs increases, and hence the number of VMs that can be proactively transitioned to booted decreases. This reduction in the number of possible proactive transitions also contributes to increase the average miss penalty.

For this workload, comparing the two online (implementable) policies (LRU and SlidingWindow+ARMA), we conclude that SlidingWindow+ARMA incurs the lower average miss penalty. It increases VM density from 250 to 550 (a gain of more than 2.2×), while keeping average miss penalty under 1 ms, for a fixed inter-arrival time of 50 seconds and fixed duration of 10 sec. Note that each VM is active for 10 sec, then becomes idle for 40 seconds before being active again, i.e., a *mean duty-cycle* of 20%. If the maximum number of frozen VMs was not limited to 300 by LXC's implementation, for up to 250 simultaneously active VMs, a maximum of $\frac{250}{0.2}$ or 1250 VMs can be hosted using this state hierarchy while keeping the average miss penalty under 1 ms. Similarly, for an idle time of 10 seconds and active time of 40 seconds (duty cycle=0.8), the maximum VM density with average miss penalty under 1 ms, would be $\frac{250}{0.8}$ or 312 VMs. To generalize, maximum VM density, for average miss penalty ≤1 ms with maximum number of simultaneously active VMs ≤ 250, equals $\text{MIN}\left(B_0 + B_1, \frac{B_0}{\text{mean duty-cycle}}\right)$.

5.5.2 Stochastic inter-arrival time, fixed duration

Figure 7 shows the average miss penalty for a request string ω with stochastic inter-arrival time (as described above), and a fixed request duration of 10 seconds, for different policies with increasing VM density. Figure 7 also shows the shutdown-to-booted and frozen-to-booted transition times for comparison.

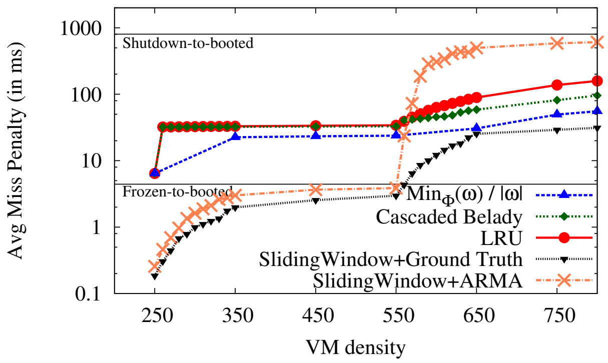


Figure 7: Stochastic inter-arrival time, fixed duration.

We observe that the behaviour of reactive policies in this case is similar to that in the case of fixed inter-arrival fixed duration workloads (as described above in Section 5.5.1). That is, their average miss penalty is equal and lowest at VM density of 250 VMs, thereafter it increases with VM density (and remains higher than the lower bound). Miss penalty is significantly higher at VM density level of 550 VMs (and more) than that at 250-550 VMs. This is because at VM density more than 550, VMs span booted, frozen, and shutdown states (and shutdown-to-booted transition time is significantly larger than frozen-to-booted time).

We now explain the behaviour of the proactive policies. Due to reasons explained above in Section 5.5.1, the average miss penalty of the SlidingWindow with future knowledge (denoted 'SlidingWindow+Ground Truth') increases as VM density increases. The increase is large at VM density of more than 550 VMs due to the shutdown-to-booted transition time being significantly larger than frozen-to-booted. Our online implementation of SlidingWindow which uses the ARMA predictor, SlidingWindow+ARMA, incurs a higher average miss penalty than SlidingWindow+GroundTruth. This is because SlidingWindow+ARMA uses the predicted inter-arrival time for each VM to make proactive transition decisions, and thus error in prediction causes some VMs to be transitioned to sub-optimal states, which increases the average miss penalty. We observe significantly larger average miss penalties beyond VM density of 550 VMs, even though the normalized root mean square error of the prediction remains largely constant (at 0.04) with increasing VM density. This is because beyond a VM density of 550, VMs span all three booted, frozen, shutdown states (as compared to only booted and frozen below 550), and the shutdown-to-booted, booted-to-shutdown transition times are significantly higher than the frozen-to-booted, frozen-to-shutdown transition times respectively. Hence, with increasing number of VMs, for a relatively constant degree of mis-predictions, the number of VMs that get transitioned sub-optimally to the shutdown state due to a mis-predicted inter-arrival time increases. It is the large transition times for the shutdown state that causes the significant increase in average miss penalty. Due to this increase the average miss penalty of SlidingWindow+ARMA exceeds that of reactive approaches (beyond 560 VMs).

For this workload, when comparing the two online policies, LRU and SlidingWindow+ARMA, we conclude that for a desired VM density of up to 550 VMs (2.2 × the density of current solutions), SlidingWindow+ARMA incurs a much lower average miss penalty (less than 4 ms). However, for all VM density levels larger than 560, LRU outperforms SlidingWindow+ARMA. At these density levels, some idle VMs need to be in the shutdown state (which has large transition times), and any error in inter-arrival time prediction, causes a significant increase in the average miss penalty.

5.5.3 Stochastic inter-arrival time, stochastic duration

Figure 8 shows the average miss penalty for a request string ω with stochastic inter-arrival time and stochastic duration, for different policies with increasing VM density. Figure 8 also shows the shutdown-to-booted and frozen-to-booted transition times for comparison.

We observe that the stochastic request duration has little effect on the average miss penalty, which is very similar to that in case of stochastic inter-arrival time and fixed duration (described above in Section 5.5.2). We believe that this is because both the reactive and proactive policies we study (LRU, Cascaded Belady's algorithm, and SlidingWindow), only use request inter-arrival time in making their reactive or proactive transition decisions. It may be possible to formulate policies which also take expected request durations into account, which we defer to future work. The behaviour of our current policies can be explained using reasoning similar to that in Section 5.5.2.

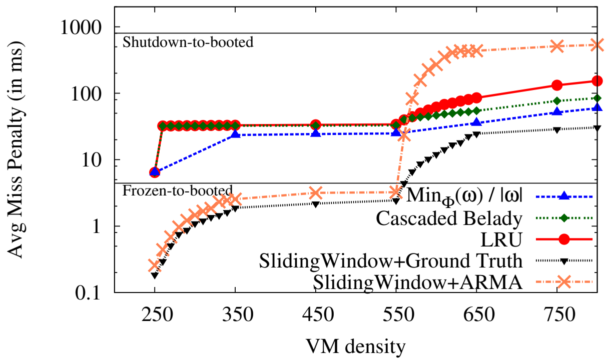


Figure 8: Stochastic inter-arrival time and duration.

5.5.4 Summary of Simulation Results

The key findings of the policy comparison (described above) can be summarized as follows:

- Online proactive policies such as SlidingWindow+ARMA are highly sensitive to prediction error. Thus for any given hierarchy, to ensure low miss penalties such policies should be used only when the desired VM density is such that all VMs can be accommodated within inactive states with low transition times. That is, when the penalties for mis-predictions are relatively small. Under all other conditions (e.g., larger VM density levels), reactive and stateless policies such as LRU should be used.
- Miss penalties of proactive and reactive (online or offline) policies which only make use of request inter-arrival times, are largely unaffected by request durations being fixed or stochastic.
- For certain workloads, using an online per-VM predictor based proactive policy results in up to a 2.2× gain in VM density with average miss penalty less than 1 ms.

6. Discussion, Limitations, and Future Work

We demonstrate multiplexing of idle VMs across inactive states using LXC as our example virtualization solution. However, our model formulation and lower bound on demand-based policies (Section 3) is applicable to any virtualization solution. Similarly, our experimental analysis to determine the model parameters (Section 4) can also be adapted to any virtualization solution. Our simulator can then easily be used (after encoding a cost-model module), to observe the effect of policies on any VM hierarchy, and policies already implemented can be re-used. We defer such extension of this work to other virtualization solutions (e.g. Xen, KVM) to future work. Nevertheless, we believe our evaluation of VM density and miss penalty can benefit existing projects [1, 14] which use LXC.

Our work is not without limitations. We have implemented only a few sample, well-known, reactive and proactive policies (i.e., LRU, Cascaded Belady's, SlidingWindow, SlidingWindow+ARMA), and have studied only one example workload (i.e., personal servers). Our goal in this work was to understand and compare the behaviour of reactive and proactive policies for idle VM management, and to compare reactive policies with our theoretical lower bound, on a given workload. We chose the personal server workload since they are a topic of active research in many areas [11, 14, 29, 31] (Section 5.3). Using this workload, we provide valuable insights into the behaviour of a few policies. However, we plan to extend our work to a broader range of policies and workloads in future work. Our modular simulator design, which isolates policy, workload, cost-model modules (Section 5.1), ensures that such a broader analysis can be conducted easily.

We have provided only one online implementation of the SlidingWindow policy (using the ARMA model). We chose the ARMA predictor since it delivered relatively small prediction error for the current workload. To use the SlidingWindow policy on other workloads other prediction methods may need to be considered, while demand-based policies such as LRU can be applied directly.

We model the problem by using a mean value analysis of transition times. Hence we derive the model parameters by defining an operating range of state capacities within which transition times are largely constant. We also assume that different VMs transitions can take place in parallel, and transition times remain unchanged. Stochastic analysis can be leveraged to model any variations in transition times. However, additional experimental analysis of density and transition times will be required to obtain parameters for a stochastic value model, which we defer to future work. Additionally, our current experimental analysis of LXC (Section 4) identifies a number of barriers to state capacities in LXC, which we plan to address in future work.

7. Conclusion

We examine the problem of multiplexing idle VMs across a hierarchy of inactive states. Our simulation-based evaluation and comparison of different policies, shows that VM density can be increased by multiplexing VMs across multiple inactive states, at the cost of a negligible increase in client request latency. Therefore, we encourage virtualization solution providers to natively support such inactive states, to allow cloud providers to increase VM density, leading to reduced hosting costs for providers (by increasing consolidation levels) and tenants (through fine-grained billing based on VM active time [6]).

Acknowledgments

Tim Brecht's research is partially supported by NSERC Discovery and Discovery Accelerator Supplement Grants. S. Keshav is supported by NSERC Discovery and CRD grants. Rayman Preet Singh is partially supported by an NSERC PGS-D scholarship, and University of Waterloo Cheriton and President's scholarships. This work is also partially funded by Microsoft Research.

References

[1] Docker: An Open Platform for Distributed Applications for Developers and Sysadmins. http://www.docker.com.

[2] Linux Containers. http://lxc.sourceforge.net/.

[3] OpenVZ. http://openvz.org.

[4] VMSim. http://github.com/rayman7718/VMSim.

[5] Linux vServer. http://linux-vserver.org.

[6] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The Resource-as-a-Service (RaaS) cloud. In *USENIX HotCloud*, 2012.

[7] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of Optimal Page Replacement. *Journal of the ACM (JACM)*, 1971.

[8] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *ACM SIGMETRICS Performance Evaluation Review*, 1996.

[9] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal*, 1966.

[10] A. B. Brush, E. Filippov, D. Huang, J. Jung, R. Mahajan, F. Martinez, K. Mazhar, A. Phanishayee, A. Samuel, J. Scott, and R. P. Singh. Lab of Things: A Platform for Conducting Studies with Connected Devices in Multiple Homes. In *ACM UbiComp 2013, Adjunct Proceedings*, 2013.

[11] R. Cáceres, L. Cox, H. Lim, A. Shakimov, and A. Varshavsky. Virtual Individual Servers as Privacy-Preserving Proxies for Mobile Devices. In *Proc. of ACM MobiHeld, 2009*.

[12] E. G. Coffman, Jr. and P. J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[13] A. Corradi, M. Fanelli, and L. Foschini. VM Consolidation: A Real Case Based on OpenStack Cloud. *Future Generation Computer Systems*, 2014.

[14] C. Elsmore, A. Madhavapeddy, I. Leslie, and A. Chaudhry. Confidential Carbon Commuting. In *Proc. of the First Workshop on Measurement, Privacy, and Mobility*, 2012.

[15] B. S. Gill. On Multi-level Exclusive Caching: Offline Optimality and Why Promotions are Better Than Demotions. In *Proc. of USENIX FAST*, 2008.

[16] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. Bolt: A Storage System for Connected Homes. In *Proc. of NSDI*, 2014.

[17] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time Provisioning for Cyber Foraging. In *Proc. of ACM MobiSys*, 2013.

[18] J. Hizver and T.-c. Chiueh. Real-time Deep Virtual Machine Introspection and its Applications. In *Proc. of ACM VEE*, 2014.

[19] B. Jennings and R. Stadler. Resource Management in Clouds: Survey and Research Challenges. *Journal of Network and Systems Management*, 2014.

[20] J. Kannan, P. Maniatis, and B.-G. Chun. A Data Capsule Framework For Web Services: Providing Flexible Data Access Control To Users. *CoRR*, 2010.

[21] T. Knauth and C. Fetzer. DreamServer: Truly On-Demand Cloud Services. In *Proc. of SYSTOR*, 2014.

[22] T. Knauth and C. Fetzer. Fast Virtual Machine Resume for Agile Cloud Services. In *Proc. of IEEE ICCGC*, 2013.

[23] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder. Validating Heuristics for Virtual Machines Consolidation. *Microsoft Research, MSR-TR-2011-9*, 2011.

[24] P. B. Menage. Adding Generic Process Containers to the Linux Kernel. In *Ottawa Linux Symposium*, 2007.

[25] R. Mortier, C. Greenhalgh, D. McAuley, A. Spence, A. Madhavapeddy, J. Crowcroft, and S. Hand. The Personal Container, or Your Life in Bits. *Digital Futures*, 2010.

[26] B. Newton, K. Jeffay, and J. Aikat. The Continued Evolution of Web Traffic. In *IEEE MASCOTS*, 2013.

[27] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. *HP Labs Technical Report*, 2007.

[28] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 2009.

[29] A. Shakimov, H. Lim, R. Caceres, L. Cox, K. Li, D. Liu, and A. Varshavsky. Vis-a-Vis: Privacy-preserving Online Social Networking via Virtual Individual Servers. In *Proc. of COMSNETS, 2011*, .

[30] A. Shakimov, A. Varshavsky, L. P. Cox, and R. Cáceres. Privacy, Cost, and Availability Tradeoffs in Decentralized OSNs. In *Proc. of ACM WOSN, 2009*, .

[31] R. P. Singh, S. Keshav, and T. Brecht. A Cloud-Based Consumer-centric Architecture for Energy Data Analytics. In *Proc. of ACM e-Energy, 2013*.

[32] R. P. Singh, T. Brecht, and S. Keshav. IP Address Multiplexing for VEEs. *ACM SIGCOMM CCR*, April 2014.

[33] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proc. of ACM EuroSys 2007*.

[34] K. S. Trivedi. Prepaging and Applications to Array Algorithms. *IEEE Transactions on Computers*, 1976.

[35] K. Wang, J. Rao, and C.-Z. Xu. Rethink the Virtual Machine Template. In *Proc. of ACM VEE 2011*.

[36] T. M. Wong and J. Wilkes. My Cache Or Yours?: Making Storage More Exclusive. In *USENIX ATC*, 2002.

[37] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proc. of NSDI 2007*.

[38] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr. Fast Restore of Checkpointed Memory Using Working Set Estimation. In *Proc. of ACM VEE*, 2011.

[39] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *Proc. of USENIX ATC*, 2013.

[40] Q. Zhang, L. Cheng, and R. Boutaba. Cloud Computing: State-of-the-Art and Research Challenges. *Journal of Internet Services and Applications*, 2010.

[41] J. Zhu, Z. Jiang, and Z. Xiao. Twinkle: A Fast Resource Provisioning Mechanism for Internet Services. In *Proc. of IEEE INFOCOM*, 2011.