vmware®

VOL. 2, NO. 2 – DECEMBER 2013

VMWARE TECHNICAL JOURNAL

Editors: Curt Kolovson, Steve Muir, Rita Tavilla

TABLE OF CONTENTS

vmware®

VMware Request for Proposals (RFP)

RFP Title: Datacenter Automation in Virtualized Environments

VMware Technical Liaison: Rean Griffith, Curt Kolovson

VMware invites university-affiliated researchers worldwide to submit proposals for funding in the general area of Datacenter Automation. Our research interests in this field are broad, including but not limited to the following topics.

- Scalable Machine Learning/Analytics for datacenter management
- Management of multi-tenant infrastructure for data-intensive applications
- Algorithms, protocols, design patterns and systems for the management of large-scale, geo-distributed, data-intensive services
- Automation of storage/network QoS in the presence of I/O resource contention for data-intensive applications

Initial submissions should be extended abstracts, up to two pages of length, describing the research project for which funding is requested and must include articulation of its relevance to VMware. After a first review phase, we will create a short list of proposals that have been selected for further consideration, and invite authors to submit detailed proposals, including details of milestones, budget, personnel, etc. Initial funding of up to $150,000 will be for a period of one year, with options to review and recommit to funding subsequently.

2014 Key Dates

March 17 – Two page extended abstracts due
April 7 – Notification of interest from VMware
May 5 – Full proposals due
May 27 – Announcement of final acceptance of proposals

All proposals and inquiries should be sent to **rtavilla@vmware.com**

Introduction

Welcome to the third volume of the VMware Technical Journal, which includes papers by VMware employees, interns and collaborators in academia. The breadth of papers presented reflects VMware's focus on simplifying the delivery and consumption of applications, and the many problems and challenges associated with achieving this.

At the highest level, the mission of VMware is to develop the software that hides the complexity of physical and virtual infrastructure, i.e. servers, disks, networks, desktops, laptops, tablets and phones, and allows our customers to simply, automatically manage their applications, their data and their users.

VMware's strategy is simple.

The software defined data center (SDDC) empowers our customers to deliver their applications, with the right service levels, at the right price; flexibly, safely, securely and compliant. We use virtualization to separate the applications from the underlying physical infrastructure, and once separated we can automate their management. VMware Hybrid Cloud environments provide application owners with choices about where they run those applications—within their own data centers, on their own SDDCs, within VMware's data centers or within our partners'. And finally End User Computing enables the safe and secure consumption of those applications, anytime, anyplace and on any device.

Realizing this vision is hard!

Success requires that VMware continues to innovate, and driving innovation is the primary purpose of the Office of the CTO here at VMware. Our mission is to drive thought leadership and to accelerate technology leadership. We do this through collaboration: with our product teams, with our customers and partners, and with academia. The goal is to sow the seeds for ideas and to cultivate them, no matter where they sprout. To put the right ideas, in the right place, at the right time, and with the right people, on the right trajectory to realize their potential and deliver the maximum value to our customers. Whether those ideas result in simple feature addition to existing products, or in fundamentally new businesses, the goal is to unleash ideas, and to enable all of VMware, and our collaborators outside, to participate.

The contributions to this third volume of VMware's technical journal reflect this broad collaboration, and the scope of the challenge in making the SDDC real. From core hypervisor performance, i.e. performance at the point of supply, to the perceived performance of virtual desktops, i.e. performance at the point of consumption. From product quality to service discovery. Topics also include latency sensitive applications and Hadoop, reflecting that the SDDC must be a universal platform for all applications. These topics are being addressed by a broad range of contributors - VMware employees, interns and our academic research partners, reflecting the need for collaboration amongst the best and the brightest, no matter where they work, to make the vision a reality.

So with that, I'll leave you with their papers. As always feedback is welcome.

Paul Strong, Vice President and Interim CTO

Analysis of the Linux Pseudo-Random Number Generators

Yevgeniy Dodis
New York University
dodis@cs.nyu.edu

David Pointcheval
DI/ENS, ENS-CNRS-INRIA
david.pointcheval@ens.fr

Sylvain Ruhault
Oppida, France
ruhault@di.ens.fr

Damien Vergnaud
DI/ENS, ENS-CNRS-INRIA
vergnaud@diens.fr

Daniel Wichs
Northeastern University
wichs@ccs.neu.edu

Abstract

A pseudo-random number generator (PRNG) is a deterministic algorithm that produces numbers whose distribution is indistinguishable from uniform. A PRNG usually involves an internal state from which a cryptographic function outputs random-looking numbers. In 2005, Barak and Halevi proposed a formal security model for PRNGs *with input*, which involve an additional (potentially biased) external random input source that is used to refresh the internal state. In this work we extend the Barak-Halevi model with a stronger security property capturing how the PRNG should accumulate the entropy of the input source into the internal state after state compromise, even with a low-entropy input source—contrary to the Barak-Halevi model, which requires a high-entropy input source. More precisely, our new robustness property states that a good PRNG should be able to eventually recover from compromise even if the entropy is injected into the system at a very slow pace. This expresses the real-life expected behavior of existing PRNG designs.

We show that neither the model nor the specific PRNG construction proposed by Barak and Halevi meets our robustness property, despite meeting their weaker robustness notion. On the practical side, we discuss the Linux `/dev/random` and `/dev/urandom` PRNGs and show attacks proving that they are not robust according to our definition, due to vulnerabilities in their entropy estimator and their internal mixing function.

Finally, we propose a simple PRNG construction that is provably robust in our new and stronger adversarial model. We therefore recommend the use of this construction whenever a PRNG with input is used for cryptography.

Keywords: randomness, entropy, security models, /dev/random

1. Introduction

Pseudo-Random Number Generators. Generating random numbers is an essential task in cryptography. Random numbers are necessary not only for generating cryptographic keys, but also in several steps of cryptographic algorithms or protocols (e.g., initialization vectors for symmetric encryption, password generation, nonce generation, etc.). Cryptography practitioners usually assume that parties have access to perfect randomness. However, quite often this assumption is not realizable in practice, and random bits in protocols are generated by a *pseudo-random number generator* (PRNG). When this is done, the security of the scheme depends on the quality of the (pseudo-) randomness generated.

The lack of assurance about the generated random numbers can cause serious damage, and vulnerabilities can be exploited by attackers. One striking example is a failure in the Debian Linux distribution [4] that occurred when commented code in the OpenSSL PRNG with input led to insufficient entropy gathering and then to concrete attacks on the TLS and SSH protocols. More recently, Lenstra, Hughes, Augier, Bos, Kleinjung, and Wachter [16] showed that a nonnegligible percentage of RSA keys share prime factors. Heninger, Durumeric, Wustrow, and Halderman [10] presented an analysis of the behavior of Linux PRNGs that explains the generation of low-entropy keys when these keys are generated at boot time. Besides key generation cases, several works demonstrated that if nonces for the DSS signature algorithm are generated with a weak PRNG, then the secret key can be quickly recovered after a few key signatures are seen (see [17] and references therein). This illustrates the need for precise evaluation of PRNGs based on clear security requirements.

A user who has access to a truly random, possibly short, bit-string can use a *deterministic* (or *cryptographic*) PRNG to expand this short seed into a longer sequence whose distribution is indistinguishable from the uniform distribution to a computationally bounded adversary (which does not know the seed). However, in many situations, it is unrealistic to assume that users have access to secret and perfect randomness. In a PRNG with input, one only assumes that users can store a secret internal state and have access to a (potentially biased) random source to refresh the internal state.

In spite of being widely deployed in practice, PRNGs with input were not formalized until 2005, by Barak and Halevi [1]. They proposed a security notion, called *robustness*, to capture the fact that the bits generated should look random to an observer with (partial) knowledge of the internal state and (partial) control of the entropy source. Combining theoretical and practical analysis of PRNGs with input, this paper presents an extension of the Barak-Halevi security model and analyzes the Linux `/dev/random` and `/dev/urandom` PRNGs.

Security Models. Descriptions of PRNGs with input are given in various standards [13, 11, 8]. They identify the following core components: the *entropy source*, which is the source of randomness used by the generator to update an *internal state*, which consists of all the parameters, variables, and other stored values that the PRNG uses for its operations.

Several desirable security properties for PRNGs with input have been identified in [11, 13, 8, 2]. These standards consider adversaries with various means (and combinations of them): those who have access to the output of the generator; those who can (partially or totally) control the source of the generator; and those who can (partially or totally) control the internal state of the generator. Several requirements have been defined:

- **Resilience** – An adversary must not be able to predict future PRNG outputs even if the adversary can influence the entropy source used to initialize or refresh the internal state of the PRNG.
- **Forward security** – An adversary must not be able to identify past outputs even if the adversary can compromise the internal state of the PRNG.
- **Backward security** – An adversary must not be able to predict future outputs even if the adversary can compromise the internal state of the PRNG.

Desai, Hevia, and Yin [5] modeled a PRNG as an iterative algorithm and formalized the above requirements in this context. Barak and Halevi [1] model a PRNG with input as a pair of algorithms (`refresh`, `next`) and define a new security property called *robustness* that implies resilience, forward security, and backward security. This property assesses the behavior of a PRNG after compromise of its internal state and responds to the guidelines for developing PRNGs given by Kelsey, Schneier, Wagner, and Hall [12].

Linux PRNGs. In UNIX-like operating systems, a PRNG with input was implemented for the first time for Linux 1.3.30 in 1994. The entropy source comes from device drivers and other sources such as latencies between user-triggered events (keystroke, disk I/O, mouse clicks). It is gathered into an internal state called the *entropy pool*. The internal state keeps an estimate of the number of bits of entropy in the internal state, and (pseudo-)random bits are created from the special files `/dev/random` and `/dev/urandom`. Barak and Halevi [1] discussed briefly the `/dev/random` PRNG, but its conformity with their robustness security definition is not formally analyzed.

The first security analysis of these PRNGs was given in 2006 by Gutterman, Pinkas, and Reinman [9]. It was completed in 2012 by Lacharme, Röck, Strubel, and Videau [15]. Gutterman et al. [9] presented an attack based on kernel version 2.6.10, for which a fix was published in the following versions. Lacharme et al. [15] give a detailed description of the operations of the PRNG and provide a result on the entropy preservation property of the mixing function used to refresh the internal state.

Our Contributions. On the theoretical side, we propose a new formal security model for PRNGs with input that encompasses all previous security notions. This new property captures how a PRNG with input should accumulate the entropy of the input data into the internal state, even if the former has low entropy only. This property was not initially formalized in [1], but it expresses the real-life expected behavior of a PRNG after a state compromise, where it is expected that the PRNG quickly recovers enough entropy, whatever the quality of the input.

On the practical side, we give a precise assessment of the two Linux PRNGs, `/dev/random` and `/dev/urandom`. We prove that these PRNGs are not robust and do not accumulate entropy properly, due to the behavior of their entropy estimator and their internal mixing function. We also analyze the PRNG with input proposed by Barak and Halevi [1]. This scheme was proven robust in [1], but we prove that it does not generically satisfy our expected property of entropy accumulation. On the positive side, we propose a PRNG construction that is robust in the standard model and in our new stronger adversarial model.

In this survey we give a high-level overview of our findings, leaving many lower-level details (including most proofs) to the conference version of this paper [7].

2. Preliminaries

Probabilities. When X is a distribution, or a random variable following this distribution, we denote $x \xleftarrow{\$} X$ when x is sampled according to X. The notation $X \leftarrow Y$ says that X is assigned the value of the variable Y, and that X is a random variable with a distribution equal to that of Y. For a variable X and a set S (e.g., $\{0,1\}^m$ for some integer m), the notation $X \xleftarrow{\$} S$ denotes both assigning X a value uniformly chosen from S and letting X be a uniform random variable over S. The uniform distribution over n bits is denoted by $\mathbf{U}_n$.

Entropy. For a discrete distribution X over a set S we denote its *min-entropy* by

$$\mathbf{H}_\infty(X) = \min_{x \in \mathrm{Supp}(X)} \{-\log \Pr[X = x]\}$$

where $\mathrm{Supp}(X) \subseteq S$ is the support of the distribution X.

Game Playing Framework. For our security definitions and proofs we use the code-based game playing framework of [3]. A game GAME has an $\mathbf{initialize}$ procedure, procedures to respond to adversary oracle queries, and a $\mathbf{finalize}$ procedure. A game GAME is executed with an adversary A as follows.

First, $\mathbf{initialize}$ executes, and its outputs are the inputs to A. Then A executes, its oracle queries being answered by the corresponding procedures of GAME. When A terminates, its output becomes the input to the $\mathbf{finalize}$ procedure. The output of the latter is called the output of the game, and we let $\mathrm{GAME}^{\mathrm{A}} \Rightarrow y$ denote the event that this game output takes value y.

In the next section, for all $\mathrm{GAME} \in \{\mathrm{RES}, \mathrm{FWD}, \mathrm{BWD}, \mathrm{ROB}, \mathrm{SROB}\}$, $\mathrm{A}^{\mathrm{GAME}}$ denotes the output of the adversary. We let $\mathrm{Adv}_{\mathrm{A}}^{\mathrm{GAME}} = 2 \times \Pr[\mathrm{GAME}^{\mathrm{A}} \Rightarrow 1] - 1$. Our convention is that Boolean flags are assumed initialized to be $\mathbf{false}$ and that the running time of the adversary A is defined as the total running time of the game with the adversary in expectation, including the procedures of the game.

3. PRNG with Input: Modeling and Security

Definition 1 (PRNG with Input). A PRNG with input is a triple of algorithm $\mathrm{G} = (\mathbf{setup}, \mathbf{refresh}, \mathbf{next})$ and a $(n, \ell, p) \in \mathbb{N}^3$ triple where

$\mathbf{setup}$ is a probabilistic algorithm that outputs some public parameters $\mathbf{seed}$ for the generator.

$\mathbf{refresh}$ is a deterministic algorithm that, given $\mathbf{seed}$, a state $S \in \{0,1\}^n$, and an input $I \in \{0,1\}^p$ outputs a new state $S' = \mathrm{refresh}(S, I) = \mathrm{refresh}(\mathrm{seed}, S, I) \in \{0,1\}^n$

$\mathbf{next}$ is a deterministic algorithm that, given $\mathbf{seed}$ and a state $S \in \{0,1\}^n$, outputs a pair $(S', R) = \mathrm{next}(S) = \mathrm{next}(\mathrm{seed}, S)$ where $S \in \{0,1\}^n$, is the new state and $R \in \{0,1\}^\ell$, is the output.

The integer n is the *state length*, ℓ is the output length, and p is the input length of G.

Before defining our security notions, we notice that there are two adversarial entities that we need to worry about: the *adversary* A, whose task is (intuitively) to distinguish the outputs of the PRNG from random, and the *distribution sampler* D, whose task is to produce inputs $I_1, I_2, \ldots$, which have high entropy *collectively*, but somehow help A in breaking the security of the PRNG. In other words, the distribution sampler models a potentially adversarial environment (or "nature") where our PRNG is forced to operate. Unlike prior work, we model the distribution sampler *explicitly* and believe that such modeling is one of the important technical and conceptual contributions of our work.

3.1. Distribution Sampler

The distribution sample D is a stateful and probabilistic algorithm which, given the current state σ, outputs a tuple (σ', I, γ, z), where

σ' is the new state for D.

$I \in \{0,1\}^p$ is the next input for the *refresh* algorithm.

γ is some *fresh entropy estimation* of I, as discussed below.

z is the leakage about I, given to the attacker A.

We denote by q_D the upper bound on the number of executions of D in our security games, and say that D is *legitimate* if[1]

$$\mathbf{H}_\infty(I_j \mid I_1, \ldots, I_{j-1}, I_{j+1}, \ldots, I_{q_\mathsf{D}}, z_1, \ldots, z_{q_\mathsf{D}}, \gamma_1, \ldots, \gamma_{q_\mathsf{D}}) \geq \gamma_j \quad (1)$$

for all $j \in \{1, \ldots, q_\mathsf{D}\}$ where $(\sigma_i, I_i, \gamma_i, z_i) = \mathsf{D}(\sigma_{i-1})$ for $i \in \{1, \ldots, q_\mathsf{D}\}$ and $\sigma_0 = 0$.

We now explain the reason for explicitly requiring D to output the entropy estimate γ_j used in (1). Most complex PRNGs, including the Linux PRNGs, are concerned with the situation in which the system might enter a prolonged state during which no new entropy is inserted in the system. Correspondingly, such PRNGs typically include some ad hoc *entropy estimation procedure* E whose goal is to block the PRNG from outputting output value R_j until the state has not accumulated enough entropy γ^* (for some entropy threshold γ^*). Unfortunately, it is well-known that even approximating the entropy of a given distribution is a computationally hard problem [19]. This means that if we require our PRNG G to explicitly come up with such a procedure E, we are bound to either place some significant restrictions (or assumptions) on D, or rely on some ad hoc and nonstandard assumptions. Indeed, as part of this work we will demonstrate some attacks on the entropy estimation of the Linux PRNGs, illustrating how hard (or, perhaps, impossible) it is to design a sound entropy estimation procedure E. Finally, we observe that the design of E is anyway completely *independent* of the mathematics of the actual `refresh` and `next` procedures, meaning that the latter can and *should* be evaluated independently of the "accuracy" of E.

Motivated by these considerations, we do not insist on any "entropy estimation" procedure as a mandatory part of the PRNG design, which allows us to elegantly side-step the practical and theoretical impossibility of sound entropy estimation. Instead, we chose to place the burden of entropy estimations on D *itself*, which allows us to concentrate on the *provable* security of the `refresh` and `next` procedures. In particular, in our security definition we will not attempt to verify if D's claims are accurate (as we said, this appears hopeless without some kind of heuristics), but will only require security when D is *legitimate*, as defined in (1). Equivalently, we can think that the entropy estimations γ_j come from the entropy estimation procedure E (which is now "merged" with D) but only provide security assuming that E is correct in this estimation (which we know is hard in practice, and motivates future work in this direction).

However, we stress that: (a) the entropy estimate γ_j will only be used in our security definitions, but not in any of the actual PRNG operations (which will only use the "input part" I, returned by D); b) we do not insist that a legitimate D can perfectly estimate the fresh entropy of its next sample I_j but only provide a *lower bound* γ_j that D is "comfortable" with. For example, D is free to set $\gamma_j = 0$ as many times as it wants and, in this case, can even choose to leak the entire I_j to A via the leakage z_j![2] More generally, we allow D to inject new entropy γ_j as slowly (and maliciously!) as it wants, but will only require security when the counter c keeping track of the current "fresh" entropy in the system[3] crosses some entropy threshold γ^* (since otherwise D gave us "no reason" to expect any security).

3.2. Security Notions

In the literature, four security notions for a PRNG with input have been proposed: *resilience* (RES) *forward security* (FWD), *backward security* (BWD), and *robustness* (ROB), with the last being the strongest notion among them. We now define the analogs of these notions in our stronger adversarial model. Each of the games below is parameterized by some parameter γ^* (since which is part of the claimed PRNG security, and intuitively measures the minimal "fresh" entropy in the system when security is expected. In particular, minimizing the value of γ^* corresponds to a stronger security guarantee.

All four security games $(\mathrm{RES}(\gamma^*)$, $(\mathrm{FWD}(\gamma^*)$, $(\mathrm{BWD}(\gamma^*)$, $(\mathrm{ROB}(\gamma^*)$, are described using the game playing framework discussed earlier, and they share the same $\mathbf{initialize}$ and $\mathbf{finalize}$ procedures in Table 1.

1 Since conditional min-entropy is defined in the worst-case manner in (1), the value γ_j in the bound below should not be viewed as a random variable, but rather as an arbitrary fixing of this random variable.

2 Jumping ahead, setting $\gamma_j = 0$ bad-refresh (I_j)corresponds to the oracle in the earlier modeling of [1], which is not explicitly provided in our model.

3 Intuitively, "fresh" refers to the new entropy in the system since the last state compromise.

As we mentioned, our overall adversary is modeled via a pair of adversaries (A, D) where A is the actual attacker and D is a stateful distribution sampler. We already discussed the distribution sampler D, so we turn to the attacker A, whose goal is to guess the correct value b picked in the **initialize** procedure, which also returns to A the public value **seed** and initializes several important variables: corruption flag **corrupt**, "fresh entropy counter" c, state S, and sampler's D initial state σ.[4] In each of the games (**RES**, **FWD**, **BWD**, **ROB**) A has access to the several oracles depicted in Table 2. We briefly discuss these oracles:

proc. initialize	**proc.** finalize (b^*)
$\text{seed} \xleftarrow{s} \text{setup}$;	**IF** $b=b^*$ *RETURN* 1
$\sigma \leftarrow 0$; $S \xleftarrow{s} \{0,1\}^n$; $c \leftarrow n$; corrupt $\leftarrow$ false; $b \xleftarrow{s} \{0,1\}$	**ELSE** *RETURN* 0
OUTPUT seed	

Table 1. The initialize and finalize procedures for **G**=(setup,refresh,next)

proc.D – refresh	**proc.** next – ror	**proc.** get – next	**proc.** get – state
$(\sigma, I, \gamma, z) \xleftarrow{s} \mathsf{D}(\sigma)$	$(S, R_0) \leftarrow \text{next}(S)$	$(S, R) \leftarrow \text{next}(S)$	$c \leftarrow 0$, corrupt $\leftarrow$ true
$S \leftarrow \text{refresh}(S, I)$	$R_1 \xleftarrow{s} \{0,1\}^\ell$	**IF** corrupt=true,	**OUTPUT S**
$c \leftarrow c + \gamma$	**If** corrupt=true,	$c \leftarrow 0$	
IF $c \geq \gamma^*$	$c \leftarrow 0$	**OUTPUT R**	**proc.** set–state (S^*)
corrupt $\leftarrow$ false	**RETURN** R_0		$c \leftarrow 0$, corrupt $\leftarrow$ true
OUTPUT (γ, z)	**ELSE OUTPUT** R_b		$S \leftarrow S^*$

Table 2. Procedures in games $\text{RES}(\gamma^*)$, $\text{FWD}(\gamma^*)$, $\text{BWD}(\gamma^*)$, and $\text{ROB}(\gamma^*)$, for **G**=(setup,refresh,next)

D–refresh. This is the key procedure in which the distribution sampler D is run, and whose output I is used to refresh the current state S. Additionally, one adds the amount of fresh entropy γ to the entropy counter c and resets the **corrupt** flag to **false** when c crosses the threshold γ^*. The values of γ and the leakage z are also returned to A. We denote by q_D the number of times A calls **D–refresh** (and hence D), and notice that by our convention (of including oracle calls into run-time calculations) the total run-time of D is implicitly upper bounded by the run-time of A.

next–ror/get–next. These procedures provide A calls with either the real-or-random challenge (provided **corrupt=false**) or the true PRNG output. As a small subtlety, a "premature" call to **get–next** before **corrupt=false** resets the counter c to 0, because then A might learn something nontrivial about the (low-entropy) state S in this case.[5] We denote by q_R the total number of calls to **next–ror** and **get–next**.

get–state/set–state. These procedures give A the ability either to learn the current state S or to set it to any value S^*. In either case c is reset to 0 and **corrupt** is set to **true**. We denote by q_S the total number of calls to **get–state** and **set–state**.

4 With a slight loss of generality, we assume that when S is random it is safe to set the corrupt corruption flag to false.

5 We could slightly strengthen our definition by only reducing c by ℓ bits in this case, but we chose to go for a more conservative notion.

We can now define the corresponding security notions for PRNGs with input. For convenience, we denote in the sequel we sometime denote the "resources" of A, by $T = (t, q_D, q_R, q_S)$.

Definition 2 (Security of PRNG with Input). A PRNG with input G=(**setup,refresh,next**) is called $(T = (t, q_D, q_R, q_S), \gamma^*, \varepsilon)$ *-robust (resp. resilient, forward-secure, backward-secure)* if, for any adversary A running in time at most t making at most q_D calls to **D–refresh**, q_R calls to **next–ror/get–next** and q_S calls to **get–state/set–state**, and any legitimate distribution sampler D inside the **D–refresh** procedure, the advantage of A in game $\text{ROB}(\gamma^*)$ (resp. $\text{RES}(\gamma^*)$, $\text{FWD}(\gamma^*)$, $\text{BWD}(\gamma^*)$), is at most ε, where

$\text{ROB}(\gamma^*)$ is the unrestricted game where A is allowed to make the above calls.

$\text{RES}(\gamma^*)$ is the unrestricted game where A makes no calls to **get–state/set–state** (i.e., $q_S = 0$).

$\text{FWD}(\gamma^*)$ is the restricted game where A makes no calls to **set–state** and a single call to **get–state** (i.e., $q_S = 1$), which is the last call that A is allowed to make.

$\text{BWD}(\gamma^*)$ is the restricted game where A makes no calls to **get–state** and a single call to **set–state** (i.e., $q_S = 1$), which is the first oracle call that A is allowed to make.

Intuitively,

- Resilience protects the security of the PRNG when not corrupted against arbitrary distribution samplers D.
- Forward security protects past PRNG outputs if the state S is compromised.
- Backward security ensures that the PRNG can successfully recover from state compromise, provided enough fresh entropy is injected into the system.
- Robustness ensures arbitrary combinations of resilience, forward security, and backward security.

Hence, robustness is the strongest and the resilience is the weakest of the above four notions. In particular, all of our provable constructions will satisfy the robustness notion, but we will use the weaker notions to better pinpoint some of our attacks.

3.3. Comparison to Barak-Halevi Model

Barak-Halevi Construction. We briefly recall the elegant construction of PRNG with input attributable to Barak and Halevi [1], since it will help us illustrate the key new elements (and some of the definitional choices) of our new model. This construction (which we call BH) involves a randomness extraction function $\text{Extract}: \{0,1\}^{n'} \to \{0,1\}^n$ and a standard deterministic PRG $\mathbf{G}: \{0,1\}^n \to \{0,1\}^{n+\ell}$. The modeling of [1] did not have an explicit **setup** algorithm, and the **refresh** and **next** algorithms are

$$\text{refresh}(S, I) = \mathbf{G}'(S \oplus \text{Extract}(I))$$

$$\text{next}(S) = \mathbf{G}(S)$$

$\mathbf{G}'$ denotes the truncation of $\mathbf{G}$ to the first n output bits. However, we will also consider the "simplified BH" construction, wherein $\mathbf{G}'$ is simply the identity function (i.e., $\text{refresh}(S, I) = S \oplus \text{Extract}(I)$), since this variant will help us illustrate our attacks better and is already secure in a variant of the BH model that relaxes the strict requirement

of "state pseudorandomness at all times" (while keeping the pseudorandomness of all outputs, which is the main property one cares about)

Attack on Simplified BH. Consider the following very simple distribution sampler D. At any time period, it simply sets $I = \alpha^p$ for a fresh and random bit α and also sets entropy estimate $\gamma = 1$ and leakage $z = \varnothing$. Clearly, D is legitimate. Hence, for any entropy threshold γ^*, the simplified BH construction must regain security after γ^* calls to the D–refresh procedure following a state compromise. Now consider the following simple attacker A attacking the backward security (and thus robustness) of the simplified BH construction. It calls $\text{set-state}(0^n)$, and then makes γ^* calls to D–refresh followed by many calls to next–ror. Let us denote the value of the state S after j calls to D–refresh by S_j and let $Y(0) = \text{Extract}(0^p)$, $Y(1) = \text{Extract}(1^p)$. Then, recalling that $\text{refresh}(S, I) = S \oplus \text{Extract}(I)$ and $S_0 = 0^n$ we see that $S_j = Y(\alpha_1) \oplus \ldots Y(\alpha_j)$, where $\alpha_1 \ldots \alpha_j$ are random and independent bits. In particular, at any point of time there are only two possible values for S_j if j is even, then $S_j \in \{0^n, Y(0) \oplus Y(1)\}$ and if j is odd, then $S_j \in \{Y(0), Y(1)\}$. In other words, despite receiving γ^* random and independent bits from D, the refresh procedure failed to accumulate more than 1 bit of entropy in the final state $S^* = S_{\gamma^*}$. In particular, after γ^* calls to D–refresh, A can simply try both possibilities for S^* and easily distinguish real from random outputs with advantage arbitrarily close to 1 (by making enough calls to next–ror).

This shows that the simplified BH construction is *never* backward secure, despite being robust (modulo state pseudorandomness) in the model of [1].

Attack on "Full" BH. The above attack does not immediately extend to the full BH construction, due to the presence of the truncated PRG $\mathbf{G}'$. Instead, we show a less general attack for some (rather than any) extractor Extract and PRG $\mathbf{G}$. For Extract, we simply take any good extractor (possibly seeded) where $\text{Extract}(0^p) = \text{Extract}(1^p) = 0^n$. Such an extractor exists, since we can take any other initial extractor Extract, and simply modify it on inputs Extract', and simply modify it on inputs 0^p and 1^p, as above, without much affecting its extraction properties on *high-entropy* distributions I. By the same argument, we can take any good PRG $\mathbf{G}$ where $\mathbf{G}(0^n) = 0^{n+\ell}$, which means that $\mathbf{G}'(0^n) = 0^n$.

With these (valid but artificial) choices of Extract and $\mathbf{G}$, we can keep the same distribution sampler D and the attacker A as in the simplified BH example. Now, however, we observe that the state S always remains equal to 0^n, irrespective of whether is it updated with $I = 0^p$ or $I = 1^p$, since the new state $S' = \mathbf{G}'(S \oplus \text{Extract}(I)) = \mathbf{G}'(0^n \oplus 0^n) = 0^n = S$. In other words, we have not gained even a single bit of entropy into S, which clearly breaks backward security in this case as well.

One may wonder if we can have a less obvious attack for an Extract and $\mathbf{G}$, much like in the simplified BH case. This turns out to be an interesting and rather nontrivial question, which relates to the randomness extraction properties (or lack of thereof) of the "CBC-MAC" construction (considered by [6] under some idealized assumptions about $\mathbf{G}'$).

Instead of following this direction, below we give an almost equally simple construction that is *provably robust* in the standard model, without any idealized assumptions.

4. Provably Secure Construction

Let $\mathbf{G}: \{0,1\}^m \to \{0,1\}^{n+\ell}$ be a (deterministic) pseudorandom generator where $m<n$. We use the notation $[y]_1^m$ to denote the first m bits of $y \in \{0,1\}^n$. Our construction of PRNG with input has parameters n (state length), ℓ (output length), and p=n (sample length), and is defined as follows:

$\text{setup}()$: Output $\text{seed} = (X, X') \leftarrow \{0,1\}^{2n}$. $S' = \text{refresh}(S, I)$: Given $\text{seed} = (X, X')$, current state $S \in \{0,1\}^n$, and a sample $I \in \{0,1\}^n$ output: $S' := S \cdot X + I$, where all operations are over F_{2^n}. $(S', R) = \text{next}(S)$: Given $\text{seed} = (X, X')$ and a state $S \in \{0,1\}^n$, first compute $U = [X' \cdot S]_1^m$. Then output $(S', R) = \mathbf{G}(U)$.

Notice that we are assuming each input I is in $\{0,1\}^n$. This is without loss of generality: we can take shorter inputs and pad them with 0s, or take longer inputs and break them up into n-bit chunks, calling the refresh procedure iteratively.

Theorem Let $n > m, \ell, \gamma^*$ be integers. Assume that $\mathbf{G}: \{0,1\}^m \to \{0,1\}^{n+\ell}$ is a deterministic (t, ε_{prg}) -pseudorandom generator. Let G=(setup,refresh,next) be defined as above. Then G is a $((t', q_D, q_R, q_S), \gamma^*, \varepsilon)$ -robust PRNG with input where $t' \approx t$, $\varepsilon = q_R(2\varepsilon_{prg} + q_D^2 \varepsilon_{ext} + 2^{-n+1})$ as long as $\gamma^* \geq m + 2\log(1/\varepsilon_{ext}) + 1, n \geq m + 2\log(1/\varepsilon_{ext}) + \log(q_D) + 1$.

5. Analysis of the Linux PRNGs

The Linux operating system contains two PRNGs with input, `/dev/random` and `/dev/urandom`. They are part of the kernel and are used in the OS security services and some cryptographic libraries. We give a precise description[6] of them in our model as a triple LINUX=(setup,refresh,next) and we prove the following theorem:

Theorem The Linux `/dev/random` and `/dev/urandom` PRNGs are not robust.

Since the actual generator LINUX does not define any seed (i.e., the algorithm setup always outputs Ø), as mentioned above, it cannot achieve the notion of robustness. However, we additionally mount concrete attacks that would work even if LINUX had used a seed. The attacks exploit two independent weaknesses, in the entropy estimator and the mixing functions, which would need both to be fixed in order to expect the PRNGs to be secure.

5.1. PRNG Overview

Security Parameters. The LINUX PRNG uses the parameters n=6144, ℓ=80, p=96. The parameter n can be modified (but requires kernel compilation), and the parameters ℓ (size of the output) and p (size of the input) are fixed. The PRNG outputs the requested random numbers by blocks of ℓ=80, bits and truncates the last block if necessary.

6 All descriptions were done by source code analysis. We refer to version 3.7.8 of the Linux kernel.

Internal State. The internal state of LINUX PRNG is a triple $S = (S_i, S_u, S_r)$ where $|S_i| = 4096$ bits, $|S_u| = 1024$ bits and $|S_r| = 1024$ bits. New data is collected in S_i which is named the *input pool*. Output is generated from S_u and S_r which are named the *output pools*. When a call to `/dev/urandom` is made, data is generated from the pool S_u, and when a call to `/dev/random` is made, data is generated from the pool S_r.

Functions refresh and next. There are two refresh functions: refresh_i that initializes the internal state and refresh_c that updates it continuously. There are two next functions: next_u `/dev/urandom` and next_r for `/dev/random`.

Mixing Function. The PRNG uses a *mixing function* M to mix new data in the input pool and to transfer data between the pools.

Entropy Estimator. The PRNG uses an *entropy estimator* that estimates the entropy of new inputs and the entropy of the pools. The PRNG uses these estimations to control the transfers between the pools and how new input is collected. This is described in detail in Section 5.2. The estimations are named E_i (entropy estimation of S_i), E_u (of S_u), E_r (of S_r).

5.2. Attacks Overview

Overview of the Attack on the Entropy Estimator. The PRNG uses an *entropy estimator* on each input that continuously refreshes the internal state of the PRNG. This estimator can be fooled in two ways. First, it is possible to define a distribution of zero entropy that the estimator will estimate to be of high entropy; second, it is possible to define a distribution of arbitrary high entropy that the estimator will estimate to be of zero entropy. This is due to the estimator conception: As it considers the timings of the events to estimate their entropy, regular events (but with unpredictable data) will be estimated with zero entropy, whereas irregular events (but with predictable data) will be estimated with high entropy. With these distributions, an attacker can control the transfer of data between the pools and force the generator not to use fresh inputs when generating data.

Overview of the Attack on the Mixing Function. The PRNG uses a *mixing function* M to mix new data in the input pool. It is possible to define a distribution of arbitrary high entropy for which the mixing function is completely counterproductive (i.e., the entropy of the internal state does not increase whatever the size of the input is). This is due to the conception of the mixing function and its linear structure. With this distribution, an attacker can force the internal state of the PRNG to contain only one bit of entropy and therefore easily predict its output.

6. Conclusion

We have proposed a new property for PRNG with input that captures how it should accumulate entropy into the internal state. This property expresses the real expected behavior of a PRNG after a state compromise, when it is expected that the PRNG quickly recovers enough entropy, even with a low-entropy external input. We gave a precise assessment of the Linux `/dev/random` and `/dev/urandom` PRNGs. We proved that these PRNGs do not achieve this property, due to the behavior of their *entropy estimator* and their *mixing function*. As pointed out by Barak and Halevi [1], who advise against using run-time entropy estimation, our attacks are due to its use when data is transferred between pools in Linux PRNGs. We therefore recommend that the functions of a PRNG do not rely on such an estimator.

Finally, we proposed a construction that meets our new property in the standard model. Thus, from the perspective of provable security, our construction appears to be vastly superior to Linux PRNGs. We therefore recommend the use of this construction whenever a PRNG with input is used for cryptography.

7. Acknowledgments

Yevgeniy Dodis's research was partially supported by the gift from VMware Labs and NSF grants 1319051, 1314568, 1065288, and 1017471. Damien Vergnaud's research was supported in part by the French ANR-12-JS02-0004 ROMAnTIC Project. Daniel Wichs's research was partially supported by NSF grant 1314722.

8. References

1 Barak, B. and Halevi, S. A model and architecture for pseudo-random generation with applications to /dev/random. In *ACM CCS 05 12th Conference on Computer and Communications Security* (Nov. 2005), V. Atluri, C. Meadows, and A. Juels, Eds., ACM Press, pp. 203–212.

2 Barker, E. and Kelsey, J. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A, 2012.

3 Bellare, M. and Rogaway, P. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT Advances in Cryptology – EUROCRYPT 2006* (May / June 2006), S. Vaudenay, Ed., vol. 4004 of LNCS Lecture Notes in Computer Science, Springer, pp. 409–426.

4 CVE-2008-0166. Common Vulnerabilities and Exposures, 2008.

5 Desai, A., Hevia, A., and Yin, Y. L. A practice-oriented treatment of pseudorandom number generators. In *EUROCRYPT Advances in Cryptology – EUROCRYPT 2002* (Apr. / May 2002), L. R. Knudsen, Ed., vol. 2332 of *LNCS Lecture Notes in Computer Science*, Springer, pp. 368–383.

6 Dodis, Y., Gennaro, R., Håstad, J., Krawczyk, H., and Rabin, T. Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In *CRYPTO Advances in Cryptology – CRYPTO 2004* (Aug. 2004), M. Franklin, Ed., vol. 3152 of *LNCS Lecture Notes in Computer Science*, Springer, pp. 494–510.

7 Dodis, Y., Pointcheval, D., Ruhault, S., Vergnaud, D., and Wichs, Daniel, Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *ACM Conference on Computer and Communication Security (CCS)*, November 2013.

8 Eastlake, D., Schiller, J., and Crocker, S. *RFC 4086 - Randomness Requirements for Security*, June 2005.

9 Gutterman, Z., Pinkas, B., and Reinman, T. Analysis of the Linux random number generator. In *2006 IEEE Symposium on Security and Privacy* (May 2006), IEEE Computer Society Press, pp. 371–385.

10 Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium* (Aug. 2012).

11 Information technology - Security techniques - Random bit generation. ISO/IEC18031:2011, 2011.

12 Kelsey, J., Schneier, B., Wagner, D., and Hall, C. Cryptanalytic attacks on pseudorandom number generators. In *FSE Fast Software Encryption – FSE'98* (Mar. 1998), S. Vaudenay, Ed., vol. 1372 of *LNCS Lecture Notes in Computer Science*, Springer, pp. 168–188.

13 Killmann, W. and Schindler, W. A proposal for: Functionality classes for random number generators. AIS 20 / AIS31, 2011.

14 Koopman, P. 32-bit cyclic redundancy codes for internet applications. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2002), DSN '02, IEEE Computer Society, pp. 459–472.

15 Lacharme, P., Rock, A., Strubel, V., and Videau, M. The Linux pseudorandom number generator revisited. Cryptology ePrint Archive, Report 2012/251, 2012.

16 Lenstra, A. K., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., and Wachter, C. Public keys. *CRYPTO Advances in Cryptology – CRYPTO 2012* (Aug. 2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 LNCS Lecture Notes in Computer Science, Springer, pp. 626–642.

17 Nguyen, P. Q. and Shparlinski, I. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology* 15, 3 (2002), 151–176.

18 Nisan, N. and Zuckerman, D. Randomness is linear in space. *J. Comput. Syst. Sci. 52*, 1 (1996), 43–52.

19 Sahai, A. and Vadhan, S. P. A complete problem for statistical zero knowledge. *J. ACM 50*, 2 (2003), 196–249.

20 Shoup, V. *A computational introduction to number theory and algebra*. Cambridge University Press, 2006.

Toward Guest OS Writable Virtual Machine Introspection

Zhiqiang Lin
The University of Texas at Dallas
zhiqiang.lin@utdallas.edu

Abstract

Over the past decade, a great deal of research on virtual machine introspection (VMI) has been carried out. This approach pulls the guest OS state into the low-level hypervisor and performs external monitoring of the guest OS, thereby enabling many new security applications at the hypervisor layer. However, past research mostly focused on the read-only capability of VMI; because of inherent difficulties, little effort went into attempting to interfere with the guest OS. However, since hypervisor controls the entire guest OS state, VMI can go far beyond read-only operations. In this paper, we discuss writable VMI, a new capability offered at the level of the hypervisor. Specifically, we examine reasons of why to embrace writable VMI, and what the challenges are. As a case study, we describe how the challenges could be solved by using our prior EXTERIOR system as an example. After sharing our experience, we conclude the paper with discussions on the open problems and future directions.

1. Introduction

By virtualizing hardware resources and allocating them based on need, virtualization [18] [19] [28] has significantly increased the utilization of many computing capacities, such as available computing power, storage space, and network bandwidth. It has pushed our modern computing paradigm from multi-tasking computing to multi-operating-system computing. Located one layer below the operating system (OS), virtualization enables system developers to achieve unprecedented levels of automation and manageability—especially for large scale computing systems—through resource multiplexing, server consolidation [32], machine migration [4], and better security [16] [15] [14] [3] [34], reliability, and portability [2]. Virtualization has become ubiquitous in the realm of enterprise computing today, underpinning cloud computing and data centers. It is expected to become ubiquitous on the desktop and mobile devices in the near future.

In terms of security, one of the best applications enabled by virtualization is virtual machine introspection (VMI) [16]. VMI pulls the guest OS state into the outside virtual machine monitor (VMM), or *hypervisor* (the terms VMM and hypervisor are used interchangeably in this paper), and performs external monitoring of the runtime state of a guest OS. The introspection can be placed in a VMM, in another virtual machine (VM), or within any other part of the hypervisor, as long as it can inspect the runtime state of the guest OS—including CPU registers, memory, disk, and network. Because of such strong isolation, VMI has been widely adopted in many security applications such as intrusion detection (e.g., [16] [24] [25]), malware analysis (e.g., [22] [5] [6] [9]), process monitoring (e.g., [30] [31]), and memory forensics (e.g., [20] [7] [9]).

However, past research in VMI has primarily focused on read-only inspection capability for the guest OS. This is reasonable, because intuitively any writable operation to the guest OS might disrupt the kernel state and even crash the kernel. In other words, in order to perform writable operations, the VMM must know precisely which guest virtual address it can safely write to, and when it can perform the write (i.e., what the execution context is). Unfortunately, this is challenging because of the well-known semantic gap problem [2]. That is, unlike the scenario with the in-guest view—where we have rich semantics such as the type, name, and data structure of kernel objects—at the VMM layer, we can view only the low-level bits and bytes. Therefore, we must bridge the semantic gap.

Earlier approaches to bridging the semantic gap have leveraged kernel-debugging information, as shown in the pioneer work Livewire [16]. Other approaches include analyzing and customizing kernel source code (e.g., [26] [21]), or simply manually writing down the routines to traverse kernel objects based on the kernel data structure knowledge (e.g., [22] [24]). Recently, highly automated binary-code-reuse-based approaches have been proposed that either retain the executed binary code in a re-executable manner or operate through an online kernel data redirection approach utilizing dual-VM support.

Given the substantial progress in the all possible approaches to bridging the semantic gap at the VMM layer, today we are almost certain of the semantics of the guest OS virtual addresses that we may or may not write to. Then can we go beyond read-only VMI? Since the VMM controls the entire guest computing stack, VMM certainly can do far more than that, such as perform guest OS memory-write operations. Then what are the benefits of writable VMI? What is the state of the art? What are the remaining challenges that must be addressed to make writable VMI deterministic? How can we address them and realize this vision?

This paper tries to answer these questions. Based on our prior experiences with EXTERIOR [10], we argue that writable VMI is worthwhile and can be realized. In particular, we show that there

will be many exciting applications once we can enable writable VMI, such as guest OS reconfiguration and repair, and even applications for guest OS kernel updates. However, there are still many challenges to solve before we can reach that point.

The rest of the paper is organized as follows: Section 2 addresses further the need of writable VMI. Section 3 discusses the challenges we will be facing. In Section 4, we present the example of a writable-VMI prototype that we built to support guest OS reconfiguration and repair. Section 5 discusses future directions, and finally Section 6 concludes the paper.

2. Further Motivation

Past research on VMI primarily focused on retrieving the guest OS state, such as the list of running processes, active networking connections, and opening files. None of these operations requires modification of the guest OS state, which has consequently limited the capabilities of the VMI. By enabling VMI to write to the guest OS, we can support many other operations on the guest OS, such as configuring kernel parameters, manipulating the IP routing table, or even killing a malicious process.

For security, writable VMI would certainly share all of the benefits of readable VMI, such as strong isolation, higher privilege, and stealthiness. In addition, it can have another unique advantage—high automation. In the following, we discuss these benefits in greater detail. More general discussion of the benefits of hypervisor-based solutions can be found in other papers (c.f., [2] [17]).

- **Strong isolation** – The primary advantage of using the VMM is the ability to shift the guest OS state out of the VM, thereby isolating in-VM from out-of-VM programs. It is generally believed to be much harder for adversaries to tamper with programs running at the hypervisor layer, because there is a world switch from in-VM to out-of-VM (unless the VMM has vulnerabilities). Consequently, we can gain higher trustworthiness of out-of-VM programs. For instance, if we have a VMM-layer guest OS process kill utility, we can guarantee that this utility is not tampered before using it to kill the malicious processes inside the guest OS.
- **Higher privileges and stealthiness** – Traditional security software (e.g., antivirus) runs inside the guest OS, and in-VM malware can often disable the execution of this software. By moving the execution of security software to the VMM layer, we can achieve a higher privilege (same as the hypervisor's) for it and make it invisible to attackers (higher stealthiness). For instance, malicious code (e.g., a kernel rootkit) often disables the `rmmod` command needed to remove a kernel module. By enabling the execution of these commands at the VMM layer, we can achieve a higher privilege. Also, the VMM-layer `rmmod` command would certainly be invisible (stealthy) to the in-VM malware because of the strong isolation.
- **High Automation** – A unique advantage of writable VMI is the enabling of automated responses to guest OS events. For instance, when a guest OS intrusion is detected, it often requires an automated response. Current practice is to execute an automated response inside the guest OS and/or notify the administrators. Again, unfortunately, any in-VM responses can be disabled by attackers because they run at the same privilege level. However, with writable VMI, we can quickly take actions to stop and prevent the attack without the assistance from any in-VM programs and their root privileges. Considering that there are a great deal of read-only VMI-based intrusion-detection systems (e.g., [6] [8] [9] [16] [22] [23] [24]), writable VMI can be seamlessly integrated with them and provide a timely response to attacks—such as `kill`-ing a rootkit-created hidden process and running `rmmod` against a hidden malicious kernel module.

3. Challenges

However, it is non-trivial to realize writable VMI at the hypervisor layer. As in all the read-only VMI solutions, we must bridge the semantic gap and reconstruct the guest OS abstractions. In addition, we will also face a concurrency problem while performing guest OS writable operations.

3.1 Reconstructing the Guest OS Abstractions

Essentially, a hypervisor can be considered to be programmable hardware. Therefore, the view at the hypervisor layer is at a very low level. Specifically, we can observe all the CPU registers and all of the physical memory cells of the guest OS. Also, we can observe all the instruction executions if the hypervisor is an instruction-translation-based VMM; otherwise we can only observe some special VMM-level instructions (e.g., Intel VT-x instructions) and special kernel events such as page faults if the hypervisor is a hardware-virtualization-based VMM.

However, what we want is the semantic information of the guest OS abstractions. For instance, for a memory cell, we want to know the meaning of that cell—for example, what is the virtual address of this memory cell? Is it a kernel global variable? If so, what does this global variable stand for? For a running instruction inside the guest OS, we also would like to know if it is a user-level instruction or a kernel-level instruction? Which process does the instruction belong to? If the instruction belongs to kernel space, is it a system-call-related instruction, a kernel-module instruction, kernel-interrupt handler, or something else? For a running system call, we also want to know the semantics of this system call, such as the system call number and the arguments.

Therefore, we must bridge the semantic gap for this low-level data and these events. In general, we must be armed with detailed knowledge of the algorithms and data structures of each OS component in order to rebuild high-level information. However, due to the high complexity of modern OSs, acquiring such knowledge is a tedious and time-consuming operation, even for open source OSs. When the source code is not available, sustained effort is needed to reverse engineer the undocumented kernel algorithms and data structures.

Because of the importance of this problem, significant research in the past has focused on how to bridge the semantic gap more efficiently and with less constraint. Currently, the state of the art includes the kernel-data-structure-based approach (e.g., [16] [26] [21] [22] [24] [1]), and the binary-code-reuse-based approach (e.g., [6] [9] [10] [11] [29]). Each has its own pros and cons. The

data-structure-assisted approach is flexible, fast, and precise, but it requires access to kernel-debugging information or kernel source code; a binary-code-reuse-based approach is highly automated, but it is slow and can only support limited functionality (e.g., with fewer than 20 native utilities supported so far in VMST [9] and EXTERIOR [10]).

```
1.  execve("/bin/hostname", ["hostname", "test"], …)= 0
2.  brk(0)                                  = 0x8427000
3.  access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
4.  mmap2(NULL, 8192,…, -1, 0) = 0xb7716000
 …
36. brk(0x8448000)                          = 0x8448000
37. sethostname("test", 4)                   = 0
38. exit_group(0)                           = ?
```

(a)

```
c103c305 <sys_sethostname>:
 1. c103c305:      push   %ebp
 2. c103c306:      or     $0xffffffff,%ebp
 3. c103c309:      push   %edi
 4. c103c30a:      push   %esi
 5. c103c30b:      push   %ebx
                 //get the current task structure
 6. c103c356:      mov    %fs:0xc13f9454,%edx
                 //point to current->nsproxy
 7. c103c35d:      mov    0x2c4(%edx),%edx
                 //point to current->nsproxy->uts_ns
 8. c103c368:      mov    0x4(%edx),%ebp
                 //point to current->nsproxy->uts_ns->name
 9.  c103c36b:     add    $0x45,%ebp
                 //store the new hostname
 10. c103c36e:     mov    %ebp,%edi
 11. c103c370:     rep movsl %ds:(%esi),%es:(%edi)
```

(b)

Figure 1. (a) System call trace of the hostname command (b) Disassembled instructions for sys_sethostname system call

3.2 Addressing the Concurrency Issue

Unlike with read-only VMI, if we aim to perform writable operations on the guest OS, we must ensure that the memory write is safe. By *safe*, we mean that the newly written value should reflect the original OS semantics. In particular, for a memory write, even though we can bridge its semantic gap, we still need to know when it is the safe moment to launch the write operation. For instance, as shown in the Figure 1(b), when writable VMI executes the `rep movsl` instruction at the hypervisor layer to set the host name of the guest OS, we need to ensure there is no concurrent execution of this instruction inside the guest OS.

In addition, the OS is designed to manage hardware resources such as CPU and memory, which are often shared by multiple processes or threads (for multiplexing). Therefore, the OS kernel is full of synchronization or lock primitives against the concurrent access of the shared resources. These synchronization mechanisms (e.g., `spinlock` and `semaphores`) would set yet another obstacle when implementing writable VMI.

Note that the concurrency issue happens at very fine granularity—that is, the memory-cell level for a particular variable. Based on the semantics, if we are sure that there is no such concurrency, we can safely perform the memory write. In other words, the outside writable operation should be like a transaction (e.g., [27]), and self-contained. For instance, the execution of the `ps` command would not affect the kernel state, and this "transaction" is self-contained and can happen multiple times even inside the guest OS.

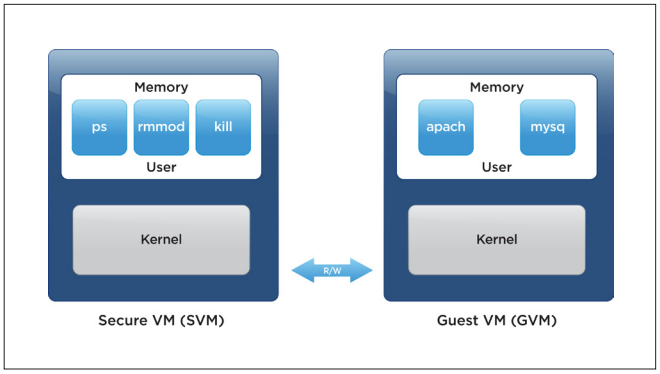


Figure 2. Overview of EXTERIOR.

There are also some other related issues, such as the performance trade-off. One intuitive approach for avoiding concurrency would be to stop guest OS execution and then perform the writable operation solely from VMI. This is doable if the performance impact on the guest OS is not so critical and if inside the guest OS there is no similar behavior to the outside writable operation.

4. Examples

In this section, we share our experiences in realizing a writable VMI system named EXTERIOR [10]. We first give an overview of our system in Section 4.1, and in Section 4.2 explain how we addressed the semantic-gap and concurrency challenges. Finally, we discuss the limitations of EXTERIOR in Section 4.3.

4.1 EXTERIOR Overview

Recently, we presented EXTERIOR, a dual-VM, binary-code-reuse-based framework for guest OS introspection, configuration, and repair. As illustrated in Figure 2, EXTERIOR enables native OS utilities such as `ps`, `rmmod`, and `kill` to execute in a secure VM (SVM) but transparently inspect and update the OS state in the guest VM (GVM). There are two requirements for EXTERIOR to work: (1) The OSs running in the two VMs must be the exact same version and (2) the SVM's hypervisor is an instruction-translation-based VMM.

The SVM is used to create the necessary running environment for the utility processes. The binary-translation-based VM is used to monitor all the instruction execution in the SVM, resolve the instruction execution context, and dynamically and transparently redirect and update the memory state at the hypervisor layer from SVM to GVM when the execution context of interest is executed, thus achieving the same effect—in terms of kernel state updates—as running the same utility inside the GVM.

We demonstrated that EXTERIOR can be used for automated management of a guest OS, including introspection (e.g., `ps`, `lsmod`, `netstat`) and reconfiguration (e.g., `sysctl`, `hostname`, `renice`) of the guest OS state without any user account in the guest OS. It also supports end users developing customized programs to repair the kernel damage inflicted by kernel malware, such as contaminated system-call tables.

4.2 Solutions to the Challenges

To bridge the semantic gap, EXTERIOR uses a binary-code-reuse-based approach. The key insight is that for compiled software, including an OS kernel, variables are usually updated by the compiled

instructions within a certain execution context. More specifically, by tracing how the traditional native program executes and updates the kernel state, we observe that OS kernel state is often updated within a certain kernel system call execution context. For instance, as shown in Figure 1, the `sethostname` utility, when executed with the `test` parameter, invokes the `sys_sethostname` system call to update the kernel host name with the new name. The instructions from line 6 to line 11 are responsible for this.

Therefore, if at the VMM layer we can precisely identify the instruction execution from line 6 to line 11 when system call `sys_sethostname` is executed, and if we can maintain a secure duplicate of the running GVM as a SVM, through redirecting both their memory read and write operations from the SVM to the running GVM, we can transparently update the in-VM kernel state of the GVM from the outside SVM. In other words, the semantic gap (e.g., the memory location of in-VM kernel state) is automatically bridged by the intrinsic instruction constraints encoded in the binary code in the duplicated VM. That is why it eventually leads to a dual-VM based architecture.

Regarding the concurrency issues, it is rare to execute these native utilities simultaneously in both SVM and GVM. For instance, the probability would be extremely low of executing a utility such as `sethostname` in the SVM at the same time that it is executed in the GVM. Meanwhile, the utilities that EXTERIOR supports are self-contained, and they can be executed multiple times in one VM. For instance, we can execute `kill` multiple times to kill a process, and we can also execute `ps` multiple times to show the running-processes list. Also, we can execute `kill` at an arbitrary time to kill a running process, because this operation is self-contained. That is why these operations can be considered transactions. A rule of thumb is that if we can execute a command multiple times in a VM and can get the same result in terms of kernel-state inspection or update, then that command can be executed in a SVM.

4.3 Limitations of EXTERIOR

The way in which EXTERIOR bridges the semantic gap and how it addresses the concurrency issue naturally lead to a number of limitations. First, it requires the two kernels to have identical kernel versions because of the nature of binary code reuse. Any new patch to the GVM kernel must be applied to the SVM. Second, it also requires the address space of kernel global variables not be randomized; otherwise it must derandomize it. Third, the execution of the monitored system call (e.g., `sys_sethostname`) will not be blocked, and the monitored system call should only operate on memory data.

Because of the above constraints, EXTERIOR cannot support the running of arbitrary administration utilities with arbitrary kernels. Also, EXTERIOR must precisely identify the instruction execution context. Currently, it can precisely identify the system-call execution context. However, a given system call can contain certain nonredirectable data, such as the variables accessed by `spin_lock` and `spin_unlock`, or semaphores accessed by `__up` or `__down`. If we cannot precisely identify the execution context of these functions, EXTERIOR is highly likely to make these relevant kernel lock primitives inconsistent when redirecting kernel data access. Currently, EXTERIOR uses a manual approach to derive the signatures for all those observed kernel locks. Such a manual approach is tedious and error-prone, and it must be repeated for different kernels.

5. Future Research

There are many directions to go in order to realize writable VMI. The two most urgent steps are to (1) push the technology further based on different constraints and (2) demonstrate the technology with more compelling applications. This section discusses both steps in more detail.

5.1 Improving the Techniques

Whether the hypervisor can access the guest OS kernel source code determines which of the two following strategies are possible: we can either retrofit the kernel source code to make it more suitable for writable VMI, or we can improve the binary code analysis of the OS kernel to automatically recognize a more fine-grained execution context such as `spin_locks`.

5.1.1 Retrofitting Kernel Source Code

As with writable VMI, we want to perform transaction-like operations. Also, at the binary-code level it is challenging to recognize the kernel-synchronization primitives. Then why not to retrofit the kernel source code to add hooks or wrappers such that, at the hypervisor layer, we can easily detect these events? This is certainly doable. For instance, much as in paravirtualization [32], we can modify kernel source code (with a compiler pass) to automatically recognize certain functions based on certain rules, and add hooks (e.g., [13]), or even rewrite some part of kernel code if the transaction-like behavior is missing (c.f., TxOS [27]).

On the other hand, to perform writable VMI at the hypervisor layer, essentially we are executing a program at the hypervisor layer to update kernel variables. Another route would be to change the binary code output (by compilers) of the given kernel in order to assist our VMI. For instance, if we can relocate the kernel variables to certain pages (instead of mixing them with all other unrelated kernel variables, which is the current practice), it would be much easier for the hypervisor to recognize and update the kernel introspection related information. For instance, through program analysis such as program slicing, if we can precisely identify the variables involved in the memory write and relocate them into special pages, we could map the pages between the SVM and the GVM such that the operation happening in the SVM is directly reflected in the GVM's state. It might be trivial to relocate the global variables, but for heap we might have to dynamically track them through pointer references. Part of our current research is working in this direction.

5.1.2 Recognizing Fine-Grained Execution Context

When we cannot retrofit the kernel source code, the only way we can move forward is to improve the binary code analysis to recognize the more fine-grained execution context. Currently, we can recognize the beginning and ending point of a system call, interrupt, and exception, through instrumenting kernel binary code and hardware events generation [9] [10] [11] [29]. We cannot recognize many other kernel functions such as context switch,

the bottom half of the interrupt handler, and many of the kernel synchronization primitives. Although we have mitigated the identification of these functions by instrumenting the timer to further disable the context switch, this is certainly not general enough and has limited functionality.

Therefore, determining how to identify fine-grained kernel function execution contexts and their semantics is a challenging problem. Manually inspecting each kernel function will not scale, and automatic techniques are needed. Having source code access is much easier, because we can instrument the code and inform the hypervisor of the execution context. When given only binary, we must automatically infer them from the information we gather.

5.2 Exploring More Applications

There will be many new exciting applications once we are able to perform fine-grained (i.e., memory-address-level) writable VMI. We have demonstrated that we can do guest OS reconfiguration such as resetting certain kernel parameters. We can also perform guest OS repair to clean the attack footprints.

Other possible applications include kernel updates. If we can quantify that a new kernel patch changes the kernel state in a transactional way, then we can certainly perform writable VMI to update the kernel defined in the patch. Other applications could be forensic applications that bypass authentication [12]. Again, the biggest advantage for writable VMI is that no explicit root privilege is required to perform a task (because it has the highest, hypervisor-level privilege).

In a broader scope, we can view writable VMI as a new program execution model that has certain components executed in-VM and certain components executed out-of-VM. These two types of components work together but have different trustworthiness and privileges. Some typical problems, such as the consumer and producer model, might be good examples of using writable VMI. For instance, we can use writable VMI to produce the kernel data for consumers inside the guest OS to consume. It might also be useful for high-performance computing (HPC), because this model splits program execution into two parts. With the support of special APIs, we might be able to improve the performance of specific HPC applications.

6. Conclusion

VMI has been an appealing application for security, but it only focuses on the guest OS read-only capability provided by the hypervisor. This paper discusses the possibility of exploring approaches for writable-capability that is necessary for changing certain kernel state. In particular, we discussed the demand for highly automated writable-VMI approaches that can be used as transactions. We also walked through the challenges that we will be facing, including the well-known semantic-gap problem, and the unique concurrency issues that occur when both in-VM and out-of-VM programs write the same kernel variables at the same time. We discussed how we can solve these challenges by using our prior EXTERIOR system as an example. Finally, we believe writable VMI is useful. It will open many new opportunities for system administration and security. There are still many open problems to work on in order to realize full-fledged writable-VMI.

References

1 Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)* (Chicago, IL, USA, 2009), pp. 555–565.

2 Chen, P. M., and Noble, B. D. "When virtual is better than real". In *Proceedings of the Eighth Workshop on Hot Topics i n Operating Systems (HOTOS'01)* (Washington, DC, USA, 2001), IEEE Computer Society, p. 133.

3 Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Confer-ence on Architectural support for programming languages and oper-ating systems* (Seattle, WA, USA, 2008), ASPLOS XIII, ACM, pp. 2–13.

4 Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation -Volume 2 (2005), NSDI'05, USENIX Association*, pp. 273–286

5 Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)* (Alexandria, Virginia, USA, 2008), pp. 51–62.

6 Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy* (Oakland, CA, USA, 2011), pp. 297–312.

7 Dolan-Gavitt, B., Payne, B., and Lee, W. Leveraging forensic tools for virtual machine introspection. *Technical Report*; GT-CS-11-05 (2011).

8 Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Sys-tems Design and Implementation (OSDI)* (2002).

9 Fu, Y., and Lin, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of 33rd IEEE Symposium on Security and Privacy* (May 2012).

10 Fu, Y., and Lin, Z. Exterior: Using a dual-vm based external shell for guest OS introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments* (Houston, TX, March 2013).

11 Fu, Y., and Lin, Z. Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. *ACM Transaction on Information System Security* 16(2) (September 2013), 7:1–7:29.

12 Fu, Y., Lin, Z., and Hamlen, K. Subverting systems authentication with context-aware, reactive virtual machine introspection. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)* (New Orleans, Louisiana, December 2013).

13 Ganapathy, V., Jaeger, T., and Jha, S. Automatic placement of authorization hooks in the Linux security modules framework. In *Proceedings of the 12th ACM Conference on Computer and communications security* (Alexandria, VA, USA, 2005), CCS '05, ACM, pp. 330–339.

14 Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)* (May 2007).

15 Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and BONEH, D. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM symposium on Operating Systems Principles* (Bolton Landing, NY, USA, 2003), SOSP'03, ACM, pp. 193–206.

16 Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium* (NDSS'03) (February 2003).

17 Garfinkel, T., and Rosenblum, M. When virtual is harder than real: Security challenges in virtual machine based computing environ-ments. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)* (May 2005).

18 Goldberg, R. P. Architectural principles of virtual machines. PhD thesis, Harvard University. 1972.

19 Goldberg, R. P. Survey of Virtual Machine Research. *IEEE Computer Magazine* (June 1974), 34–45.

20 Hay, B., and Nance, K. Forensics examination of volatile system data using virtual introspection. *SIGOPS Operating System Review 42* (April 2008), 74–82.

21 Hofmann, O. S., Dunn, A. M., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with osck. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA, 2011), ASPLOS '11, pp. 279–290.

22 Jiang, X., Wang, X., and Xu, D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communica-tions Security (CCS'07*) (Alexandria, Virginia, USA, 2007), ACM, pp. 128–138.

23 Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Ant-farm: tracking processes in a virtual machine environment. In *Proceedings USENIX '06 Annual Technical Conference* (Boston, MA, 2006), *USENIX Association*.

24 Payne, B. D., Carbone, M., and Lee, W. Secure and flexible monitor-ing of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (December 2007).

25 Payne, B. D., Carbone, M., Sharif, M. I., and Lee, W. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of 2008 IEEE Symposium on Security and Privacy* (Oakland, CA, May 2008), pp. 233–247.

26 Petroni, JR., N. L., and Hicks, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Confer-ence on Computer and Communications Security* (Alexandria, Virginia, USA, 2007), CCS '07, ACM, pp. 103–115.

27 Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA, 2009), SOSP '09, ACM, pp. 161–176.

28 Rosenblum, M., and Garfinkel, T. Virtual machine monitors: Current technology and future trends. *IEEE Computer* (May 2005).

29 Saberi, A., Fu, Y., and Lin, Z. Hybrid-Bridge: Efficiently Bridging the Semantic-Gap in Virtual Machine Introspection via Decoupled Execution and Training Memoization. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'14)* (February 2014).

30 Sharif, M. I., Lee, W., Cui, W., and Lanzi, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA, 2009), CCS '09, ACM, pp. 477–487.

31 Srinivasan, D., Wang, Z., Jiang, X., and Xu, D. Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (CCS'11) (Chicago, Illinois, USA, 2011), pp. 363–374.

32 Whitaker, A., Shaw, M., and Gribble, S. D. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference* (2002).

33 Whitaker, A., Shaw, M., and Gribble, S. D. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design And Implementation* (Boston, Massachu-setts, 2002), OSDI '02, ACM, pp. 195–209.

34 Zhang, F., Chen, J., Chen, H., and Zang, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), *SOSP '11, ACM*, pp. 203–216.

A Benchmark for User-Perceived Display Quality on Remote Desktops

Yue Gong
MTS
ygong@vmware.com

Winnie Wu
Sr. MTS
wuy@vmware.com

Alex Chen
MTS
chenm@vmware.com

Yang Liu
MTS
yangliu@vmware.com

Ning Ge
MTS
nge@vmware.com

Abstract

User experience is a key consideration when organizations prepare for and deploy virtual desktop infrastructure (VDI). Several existing tools on the market claim that they can measure the user experience by calculating the CPU utilization or latency that is required to operate applications on remote desktops. These methods no doubt make sense, but they are indirect and limited for assessing what the end users perceive. In the world of VDI, humans perceive display quality through their visual sense. At this time, the industry lacks a mechanism for estimating display quality

This paper presents a creative benchmarking approach that uses fair and precise data for measuring the display quality that end users perceive on remote desktops. To calculate the display quality, this benchmark compares the screen on the host side (the VDI client side), which is what end users really see, with the baseline—the screen on the remote side, which is what end users expect to see. Two indicators are delivered by this benchmark: (a) the relative frame rates and (b) the image quality score. Several protocols (PCoIP, RDP, Blast) with seven typical VDI user scenarios across different network environments (LAN, WAN) were tested to ensure that the metrics are correlated with what end users perceive.

Within VMware, development efficiency of the remote protocol could be improved by using this benchmarking method to automatically collect display-quality data. Beyond that, VMware could benefit more on the VDI market, once we could drive the industry-standard establishment of VDI performance based on this benchmark.

Keywords: display quality, remote desktop, user experience

1. Introduction

For a virtual desktop infrastructure (VDI) solution, one of the biggest challenges is to provide a good-enough desktop experience to the end user. The remote display protocol is responsible for sending the display data of the virtual desktop to the client and even for optimizing the data transmission and processing. As such, it becomes very important to understand and measure the user experience implemented by the display protocols in a variety of environments [1]. Several organizations have delivered test tools or methodologies based on resources occupied or the response time over protocols. Although these methods are useful, they are indirect and limited in their ability to assess what the end users perceive. Humans perceive display quality in the world of VDI through their visual sense. Unfortunately, at this time the industry lacks a metric to estimate the user experience based on this point.

This paper presents a creative benchmarking approach to measuring user-perceived display quality on remote desktops using fair and precise data. To calculate the display quality, it compares the screen on the host side (the VDI client side), which is what end users really see, with the baseline—the screen on the remote side, which is what end users expect to see; and it delivers two indicators: (a) the relative frame rates (b) the image quality score.

A timestamp is used to record the frame sequence. To ensure the timestamp's accuracy, this benchmark polarizes the brightness of the timestamp area, which might be blurred by some remote protocols. When identifying the baseline for a specified screen image, this benchmark carefully assesses the side effect of inevitable asynchronies between the remote-screen frame rate and the timestamp rate, and corrects it. When comparing the captured screen images on the client side with the baseline, this benchmark adopts both Peak Signal-to-Noise Ratio (PSNR) [2] and Structural Similarity (SSIM) [3] algorithms, which are most commonly used in the industry to measure image similarity. Based these considerations and adjustments, this benchmark strives to deliver fair and precise results.

To be more consistent with humans' visual perception, this benchmark combines the PSNR and SSIM algorithms to measure image similarity. It increases the weight of the intensive bad blocks and bad screen frames when calculating the image quality score and, furthermore, elaborately tunes the formulas' factors based on the test results of seven typical VDI user scenarios across different protocols (PCoIP, RDP, Blast) with different network environments (LAN, WAN). The experiment that is demonstrated in section 3 proves that this benchmark achieves about 90% approximation to the true end-user experience.

2. Display Quality Benchmark

2.1 Baseline: The Remote Screen

The benchmark uses the remote screen as the baseline when measuring display quality. In most cases, display protocols are responsible for rendering the screen of the remote desktop on

the client screen synchronously. When the remote side stutters, it is generally foreseeable that the client screen will stutter to the same degree. To minimize the unexpected impact on display-quality testing from various unrelated factors, such as limited CPU resources, this benchmark requires sufficient performance of the remote virtual machine.

Several technologies, such as multimedia redirection, accelerate display performance. In scenarios that use such technologies, the remote screen and the client screen can be expected to be different. This paper leaves this topic as future work.

2.2 Relative Frame Rate

Frame rate commonly refers to the speed at which the screen image is refreshed. It is typically expressed in frames per second (FPS). In this paper, frame rate refers in particular to the screen-refreshing rate after consolidation of contiguous screen frames that are identical. Typical LCD monitors on the market are locked at 60Hz, but the actual frame rate doesn't necessarily equal 60fps. When the screen image changes every 50ms, the frame rate = 1s/(50ms/frame)=20fps.

Ideally, the frame rate on the host side is the same as it on the remote side. However, in the real world, host FPS is always slower than its remote peer. This benchmark uses *relative frame rate*, which has a data range from 0 to 100%, as one important indicator for evaluating the efficiency of VDI protocols:

$$RelativeFrameRate = \frac{frame\ rate\ at\ host\ side}{frame\ rate\ at\ remote\ side}$$

2.2.1 Timestamp

Use of the timestamp is designed to:

- Minimize the impact to the original screen content
- Maximize the frame rate of the timestamp to ensure that all frames are captured
- Provide a sufficient domain of time span to ensure that the display quality can be assessed
- Maximize the signal-to-noise ratio to avoid the impact of possible quality decline through protocols

The timestamp that is used in this benchmark is represented by a prepared video. The video is in a small area (e.g., 4 x 4 pixels) and plays on the top of the screen at a very fast speed (e.g., 100fps). Pixels of each frame of the timestamp are black or white. If white is considered to be 1 and black to be 0, each frame becomes a two-dimensional code. For example, the timestamp in Figure 1 is: 0000110001000011B = 3139. It provides a time span of (2^16 frames)/(100fps) = 655.36 seconds ≈ 11 mins.

2.2.2 Frame Rate on the Remote Side

The steps to get the frame rate on the remote side are:

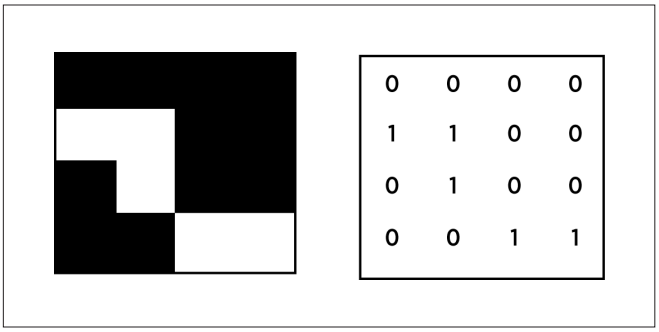


Figure 1. Timestamp Example

1. Capture the remote screen faster than its frame rate.
2. Select only one frame from the frames with the same timestamp.
3. Count the frames.
4. Calculate the frame rate as:

$$FrameRate = \frac{SUM\ of\ nonduplicate\ frames}{time\ span\ of\ the\ testing}$$

2.2.3 Frame Rate on the Client Side

The steps to get the frame rate on the host side are basically identical to the steps in section 2.2.2.

However, there is an issue for timestamps captured on the host side. Remote protocols can impart noise to the timestamp due to optimizations, such as bandwidth saving, that are widely used by most remote display protocols. The timestamp in Figure 2 shows an example. White pixels could be changed from 0xFFFFFF to 0xDDFFFF, and black pixels could be changed from 0x000000 to 0x001740.

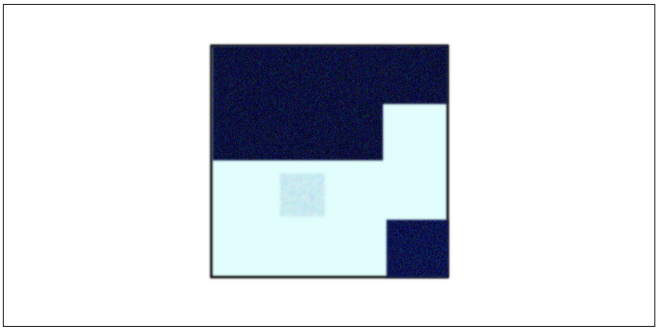


Figure 1. A Blurred Timestamp

To identify timestamps, the benchmark:

1. Calculates the perceived brightness of each pixel by its RGB values. The formula recommended by the W3C working draft on accessibility [4] is used:

$$Brightness = 0.299R + 0.587\ G + 0.144\ B$$

2. Polarizes the pixels to white or black: If the brightness of the pixel exceeds 128, treat it as white; otherwise treat it as black.
3. Calculates the number of the timestamp.

2.3 Image Quality Score

Image quality is also an important indicator of display quality. When the screens on the host side are not identical to the baseline, end users might perceive the difference. Assuming that the baseline is perfect, the degree of negative feeling depends on the difference between baseline images and host images, and the time span of relatively consecutive difference occurs. It has a data range from 0 to 100.

Generally there are four steps to getting a straightforward number as the image quality score:

1. Record screens on both the remote side and the host side, and discard the duplicate frames.
2. For each recorded frame on the host side, identify its baseline according to timestamps.
3. Calculate the image quality score of each frame on the host side.
4. Measure the aggregation effect of the "bad block" in the screen over time, and give the final score.

2.3.1 Screen Recording

Various conditions must be addressed to ensure the accuracy of the result:

- Reduce the latency of the starting point to synchronize the host-side and remote-side captures to maximize the match rate of frames. Virtual Channels can be used.
- Boost the recording speeds on both sides as much as possible. Approximately twice real FPS is adequate.
- Minimize the performance side-effect of frequent screen captures caused by slow I/O operations. Sufficient memory might be mandatory on certain machines.

2.3.2 Baseline Frames Identification

In most cases, frames are paired by matching the timestamps of captures from the host and remote sides.

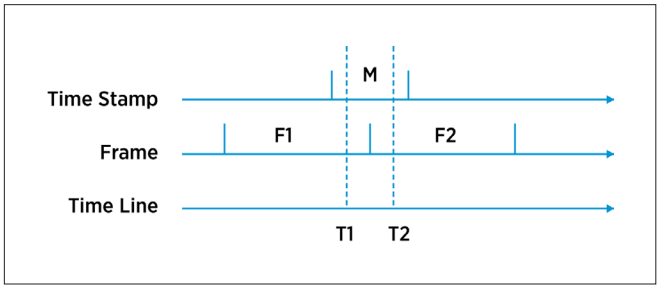


Figure 3. Frame Mismatch by Out-of-Sync Time Lines

However, because of network latency, timestamps are not exactly synchronized. As Figure 3 shows, at the period of timestamp M, the frame changes from F1 to F2. And the captured image can be either F1 (captured at T1) or F2 (captured at T2). So it is a potential risk that remote-side captures F1 and host-side captures F2 and both captured frames are stamped with the same timestamp M. Because of inaccurate time sync between remote and host machines, it is also possible for the remote side to capture F2 and the host side to capture F1. To avoid this condition, the following mechanisms are introduced:

For each frame on the host side, find the first frame (here called *target*) with the same timestamp from the recorded screen of the remote side, and:

- If the target frame is equal to its previous frame and its next frame, it is the baseline frame.
- If the target frame is not equal to either its previous frame or its next frame, respectively calculate the frame image quality scores (refer to section 2.3.3) with this target frame, its previous frame, and its next frame. The frame that gets highest score is used as the baseline frame.
- Otherwise, respectively calculate the frame image quality scores with the target's previous frame and the target's next frame. Then the frame that gets highest score is used as the baseline frame.

2.3.3 Image Quality for Each Frame

2.3.3.1 Block Similarity

In common circumstances, if the image quality for the entire frame is used directly as the final result, visually sensitive screen differences such as a straight line on the screen that impacts only a small portion of the image would be concealed unexpectedly. Dividing the frame into small pieces (e.g., 16 x 16 pixels) causes a single pixel shift to be weighted more than it is to the whole frame.

Based on the characteristic difference between PSNR and SSIM, the benchmark calculates the final score by using the following formula, which has a data range from 0 to 100%:

$$BlockSimilarity = \begin{cases} \frac{PSNR \times 50\%}{18}, & PSNR \le 18\ and\ SSIM \ge 50\% \\ SSIM, & otherwise \end{cases}$$

2.3.3.2 Bad-Block Density

For each frame, mark every block with BlockSimilarity lower than 50% as *bad*. Then:

1. Select a radius as the Concerned Density Area for all bad blocks. This benchmark recommends using 1%-2% of frame width.
2. For each bad block, count the other bad blocks in its Concerned Density Area.
3. Use the maximum result in the step 2 as *Density Max Num*.
4. Calculate the Max Bad Block Density Ratio:

$$MaxBadBlockDensityRatio = \frac{Density\ Max\ Num}{(radius \times 2 + 1)^2}$$

2.3.3.3 Frame Image Quality Score

Overall, weighted average is used to calculate the image quality score for each frame. For the blocks that have a BlockSimilarity lower than or equal to 50%, more weight is added to increase the sensitiveness of the bad blocks. Furthermore, the Max Bad Block Density Ratio also impacts the score. The data range of Frame Image Quality Score is from 0 to 100.

$$FrameImageQualityScore = \frac{\sum_{BS>50\%}^{BS=100\%} BS\times A + \sum_{BS=0}^{BS=50\%} BS\times A\times(1-BS)\times 10}{\sum_{BS>50\%}^{BS=100\%} A + \sum_{BS=0}^{BS=50\%} A\times(1-BS)\times 10} \times[(1-MaxBadBlockDensityRatio)\times 10\% + 90\%]$$

BS: Block Similarity

A: number of blocks whose Block Similarity is equal to BS

2.3.4 Image Quality Score

The VDI remote desktop is a series of related frames, not a pile of irrelevant images. For example, a noisy block displayed on the host screen at the same place for a period of time is quite noticeable to human eyes. So the aggregation effect of the bad block in the screen over time must be measured, as the weighted average of Frame Image Quality Scores do:

1. Set the initial value of ContinuousBadness to 0, which represents that no continuous bad frames are found.
2. For each frame, after step 3 of section 2.3.3.2, select the most noisy block.
3. For the two adjacent frames, calculate the PSNR value of their noisiest block. If the PSNR value is higher than 18, consider this as a ContinuousBadness, and increase ContinuousBadness by 1.

SSIM is very sensitive to image position shift. It is not suitable for this scenario.

4. Calculate the Image Quality Score by the following formula:

$$ImageQualityScore = \frac{\sum_{FIQS=0}^{FIQS=100} FIQS\times A}{\sum_{FIQA=0}^{FIQS=100} A} \times\left[\left(1-\frac{ContinuousBadness}{sum\ of\ frames\ -1}\right)\times 10\% + 90\%\right]$$

FIQS: FrameImageQualityScore
A: amount of the frames whose Frame Image Quality Score equal to FIQS

The data range of Image Quality Score is from 0 to 100.

3. Benchmark Evaluation

In all, 42 cases are used to test this benchmark:

7 typical VDI scenarios
x
3 protocols (RDP, PCoIP, Blast)
x
2 network environments (LAN, 2M bandwidth WAN).

3.1 Evaluation of Relative Frame Rate

To evaluate the Relative Frame Rate, a reliable mechanism is required to acquire the frame rates on both the remote and client sides. PCoIP protocol logs the frame rates it delivers, which could be reliable. Figure 4 demonstrates two groups of Relative Frame Rates. In summary, the mean error of the Relative Frame Rate delivered by this benchmark in all test cases under PCoIP protocol is less than 7% compared to the results from PCoIP logs.

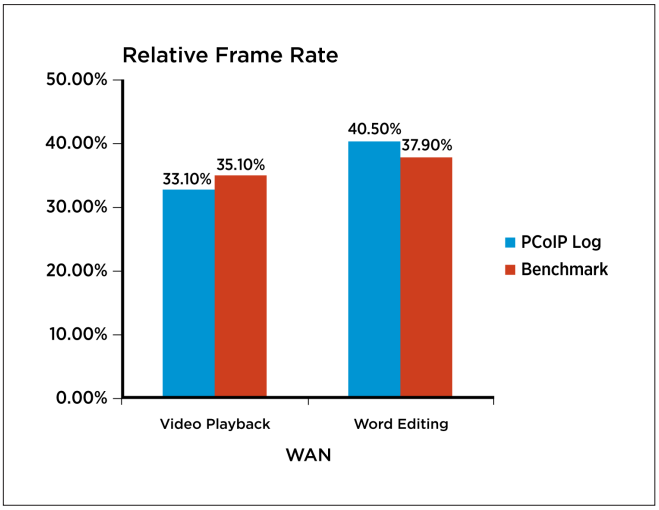


Figure 4. Comparisons of Relative Frame Rate

Mean error of RDP and Blast protocols are left as future work, because no comparable frame rate data is available from the protocols themselves.

3.2 Evaluation of Image Quality Score

To evaluate the Image Quality Score, a comparison between the benchmark and the end-user scores takes place. Four people execute all the test cases manually and record their scores. For each test case, the error rate is populated by comparing the score from the benchmark with the average scores perceived by end users. Figure 5 demonstrates four groups of Image Quality Scores. Over all the test cases, the mean error rate of the Image Quality Scores delivered by this benchmark when compared to the average score perceived by end users is less than 10%. Considering system errors, the benchmark could be used as a measurement of true VDI end user experience.

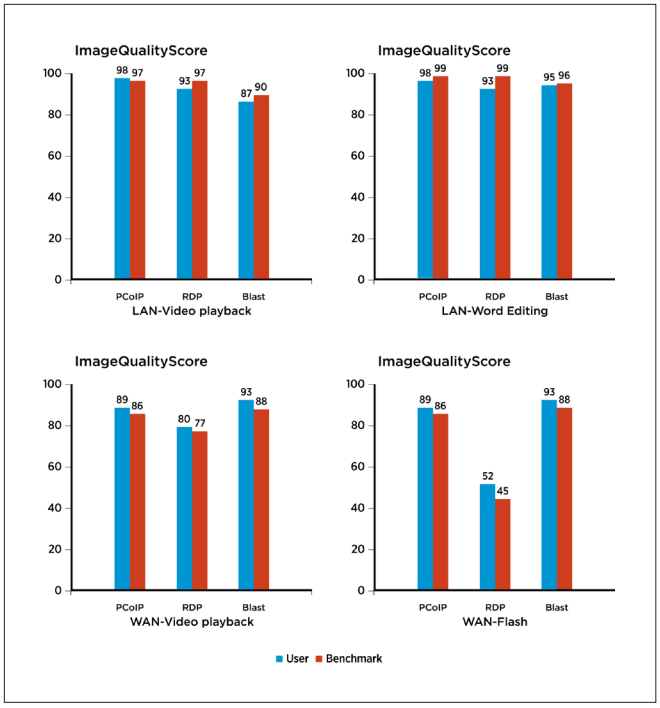


Figure 5. Comparisons of Image Quality Score

4. Related Work

To understand and measure the user experience implemented by the display protocols in variable environments, several companies/organizations/individuals have delivered test tools or methodologies based on the response time over various protocols.

Microsoft Tech Ed 2010 included a session named "RDP, RemoteFX, ICA/HDX, EOP and PCoIP – VDI Remoting Protocols Turned Inside Out" by Bernhard Tritsch (and recommended by Citrix) [5]. It used a methodology named Application Performance Index to test remote protocols. This method measured the times between user actions and the responses on the remote desktop, and transferred the time numbers into the levels perceived by the users. This method provides a valid way to measure the response time of the perceived performance of the remote desktops. However, it only measures response time and doesn't mention/measure display quality.

Login Consultants announced the new Client Side Performance Testing module for Login VSI in March 2012. At this time the product is a beta version. The release notes say: "The Client Side Performance Testing module can now be used to perform these tests: … Image quality and loading times…This image has been specifically designed to measure quality and speed of different protocols in an independent way." However, via its results [6], it measures only image response time—how long it takes to show a complex image onscreen—but does not measure display quality.

Scapa TPP says it "measures the end to end performance of the system, from the end users' perspective". But it in fact records "the overall latency down the ICA/RDP protocols per user, per location and per server/HVD" [7]. It doesn't measure display quality.

Simon Bramfitt, a former Gartner analyst, published a blog entitled "Developing a Remote Display Technology User Experience Benchmark" [8] in April 2012. As this blog showed, his method captured the output directly from the server and on the client, then compared any variations in timing and image quality to measure the impact that the remote display infrastructure had on the output quality. This is an idea quite similar with ours. However:

- The blog doesn't mention which image analysis algorithm or software was used, but only says, "The image analysis software that I have been using in my preliminary investigations is rather costly, however I am committed to working with the developers to see what can be done to repackaging it in such a way to make it more affordable for this type of activity".
- The blog doesn't mention if Bramfitt paid attention to how to synchronize the output from the server and on the client. This is an important technical problem impacting the accuracy of the display-quality verification.
- The blog mentions "a frame by frame comparison." In fact, the display on a remote desktop is a stream. Bramfitt's method lacks any consideration of this fact.
- The implementation of Bramfitt's method was still in progress. Although the blog includes a plan to release an implementation, we can't locate one.

5. Conclusion and Future Work

This paper presents a creative benchmarking approach. Two indicators are delivered by this benchmark: (a) the relative frame rates; (b) the image quality score. The comparison experiments prove that this benchmark achieves about 90% approximation to the true end-user experience and is fair and precise for measuring user-perceived display quality on remote desktops.

In the future, the benchmark will be extended to cover scenarios in which significant differences are expected between the remote screen and the client screen. For example, in some multimedia redirection solutions, screen areas displaying video are quite different between remote and client screens.

Acknowledgments

The authors would like to thank Clayton Wishoff and Jonathan Clark for their valuable suggestions for the idea, the workflow details, and the algorithm selection. The authors also appreciate the helpful reviews by John Green, Shixi Qiu, and Hayden Fowler.

References

1 Comprehensive Performance Analysis of Remote Display Protocols in VDI Environments. http://www.vmworld.com/docs/DOC-3574

2 Peak signal-to-noise ratio. http://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

3 Structural similarity. http://en.wikipedia.org/wiki/Structural_similarity

4 Techniques For Accessibility Evaluation And Repair Tools. http://www.w3.org/TR/2000/WD-AERT-20000426

5 RDP, RemoteFX, ICA/HDX, EOP and PCoIP – VDI Remoting Protocols Turned Inside Out, Microsoft MSDN. http://channel9.msdn.com/Events/TechEd/Europe/2010/VIR401

6 Client Side Performance Testing (Beta), CSPT Result, LoginVSI. http://www.loginvsi.com/documentation/v3/cspt-results

7 Taking Measurements from the End User Perspective, Scapa Test and Performance Platform. http://www.scapatech.com/wp-content/uploads/2011/04/ScapaSync_final.pdf

8 Developing a Remote Display Technology User Experience Benchmark. http://www.linkedin.com/groups/Developing-Remote-Display-Technology-User-117246.S.112304728

Virtualizing Latency-Sensitive Applications: Where Does the Overhead Come From?

Jin Heo
VMware, Inc.
heoj@vmware.com

Reza Taheri
VMware, Inc.
rtaheri@vmware.com

Abstract

This paper investigates the overhead of virtualizing latency-sensitive applications on the VMware vSphere® platform. Specifically, request-response workloads are used to study the response-time overhead compared to native configurations. When those workloads are virtualized, it is reasonable to anticipate that (1) there will be some overhead due to additional layers of processing required by the virtualization software and (2) a similar amount of overhead will be seen in response times across a variety of request-response workloads as long as the same number of transactions are executed and the packet size is similar. Findings in this paper confirm some overhead in virtual systems. However, response-time overhead increases with workload complexity instead of staying the same. The analysis of five different request-response workloads in vSphere shows that the virtualization overhead in the response time of such workloads consists of the combination of two different parts: (1) a constant overhead and (2) a variable part that monotonically increases with the complexity of the workload. This paper further demonstrates that the added layers for virtualizing network I/Os and CPUs bear the major responsibility for the constant overhead, whereas hardware-assisted memory management unit (MMU) virtualization plays a major role in the variable part of the overhead.

1. Introduction

Virtualization has been widely adopted to run the majority of business-critical applications, such as Web applications, database systems, and messaging systems [2]. As virtualizing those applications has been proven successful in recent years, growing interest has also arisen in virtualizing performance-critical applications such as those that are latency-sensitive—including distributed in-memory data management, stock trading, and streaming media. These applications typically have a strict requirement for the response time of a transaction, because violating the requirement can lead to loss of revenue. The question arises, then, of how well virtualization performs when running virtual machines (VMs) of such applications.

Virtualization requires some level of overhead. Extra processing layers inherent in virtualization enable the benefit of sharing resources while giving the illusion that VMs own the hardware. Successful virtualization of latency-sensitive applications therefore starts from fully understanding where the overhead lies. This paper studies the overhead in virtualizing latency-sensitive workloads on the most recent release (as of this writing) of the VMware virtualization platform, vSphere 5.1.

Specifically, this paper investigates response-time overhead of running request-response workloads in vSphere. In this type of workload, the client sends a request to the server, which replies with a response after processing the request. This kind of workload is very commonly seen: Ping, HTTP, and most remote procedure call (RPC) workloads all fall into the category. This paper primarily focuses on configurations in a lightly loaded system in which only one transaction of request-response is allowed to run. Any overhead induced due to additional efforts performed by the virtualization layer then will be directly shown as an extra latency. This is one of the main reasons that request-response workloads can be categorized as latency-sensitive.

When a workload is virtualized, a certain amount of overhead is expected due to the extra layers of processing. Because a request-response workload processes exactly one request-response pair per transaction, it is reasonable to anticipate a similar amount of overhead in response time across a different variation of the workload (as long as the packet size is similar). Very interestingly, however, the virtualization overhead in response time—that is, the difference between response times of native and virtualized setups—is not constant at all across different workloads. Instead, it increases with regard to the response time of the workload itself; the gap in response times becomes wider as the response time of a workload (i.e., transaction execution time) gets longer. On the contrary, the relative overhead (i.e., the percentage of overhead) decreases with regard to the response time, which implies that there also exists a nonnegligible constant component in the virtualization overhead that is more dominant in a simpler workload.

This paper demonstrates with five different request-response workloads that the virtualization overhead of such a workload consists of the combination of two different parts: (1) a constant overhead and (2) a variable part that monotonically increases with regard to the response time of the workload. This paper further shows that the added layers for virtualizing network I/Os and CPUs take the major responsibility for the constant overhead, while hardware-assisted memory management unit (MMU) virtualization plays a major role in the variable part of the overhead.

The rest of the paper is organized as follows. Section 2 describes the characteristics of the latency-sensitive workloads used in this paper. Sections 3 and 4 explain the breakdown of the response-time overhead of running latency-sensitive applications on the vSphere platform. Section 5 presents evaluation results and analysis with five different latency-sensitive workloads that leads to the breakdown of the response-time overhead explained in Sections 3 and 4. Section 6 presents more discussion. The related work is presented in Section 7. The paper concludes with Section 8.

2. Request-Response Workload and Disclaimers

Latency-sensitive workloads such as request-response workloads (hereafter, this paper refers to these types of workloads as RR workloads) would lose their characteristic of latency-sensitivity as the intensity of workload increases and contention for resources starts. For example, with a large number of latency-sensitive workload sessions running at the same time, the average response time of individual sessions would increase as they contend with one another for resources such as CPU and network bandwidth. The same thing would happen as the number of VMs increases with limited resources available to them. Queuing theory tells us that an increase in the response time with regard to workload intensity (i.e., the number of sessions, packet rate, or the number of VMs) is generally superlinear. The implication is that the system (or VMs) would easily move off the desired region where latency-sensitivity is kept as the system gets overloaded. With the response time substantially increased, one would hardly call such an application latency-sensitive. Using a request-response workload, this paper primarily focuses on the region where latency-sensitivity is kept and there is not much contention for resources.

In this paper, only one request-response session is generated, to avoid the effect of processing accumulated (batched) requests in the server. The purpose is to have configurations that are as simple as possible to keep the system in the region of latency-sensitivity. For a similar reason, this paper mainly studies 1-(v)CPU configurations to avoid the impact of parallel processing and any multi-CPU-related overhead, although virtual symmetric multiprocessing (VSMP) configurations are briefly studied later in the paper. Because only one request-response pair is processed at a time, everything is serialized and any overhead introduced due to virtualization will show up directly in latency. Only the server machine is virtualized for results presented in the paper so that virtualization overhead implies the overhead of virtualizing the server application of the given workload. Details about the setup are described later in the paper.

3. Constant Response-Time Overhead

This paper shows that, even with the same number of transactions executed (i.e., one transaction), the response-time overhead of RR workloads in a virtualization setup is not constant. Instead, it consists of the combination of both a constant overhead and a variable part that increases with the workload transaction response time. (i.e., the longer the workload runs, the larger the overhead becomes.) This paper takes a top-down approach to first explain the response-time overhead breakdown for both the constant and the variable components, followed by evaluation and analysis results with five RR workloads from which the breakdown is drawn. This section explains the constant response-time overhead, and section 4 presents the variable component.

This paper focuses on RR workloads that involve one request-and-response pair per transaction. Sending and receiving packets requires access to physical resources such as CPUs and network interface cards (NICs). In a virtualized environment, however, the guest cannot directly touch the physical resources in the host; there are extra virtual layers to traverse, and the guest sees only virtualized resources (unless the VM is specifically configured to be given direct access to a physical resource). This implies that handling one packet-transmission-and-reception pair per transaction requires a certain amount of effort by the virtualization layer, thereby incurring an overhead in response time.

This section describes the constant overhead component in response time that is due to virtualization when an RR workload is run.

3.1 Network I/O Virtualization

Because processing network I/Os involves access to a physical network interface card (PNIC) that is shared by multiple VMs, a VM should not directly control the device (see Figure 1). This might confuse other VMs accessing the same PNIC and badly affect their operations. At the same time, however, the VM needs to have the illusion that it completely owns its device.

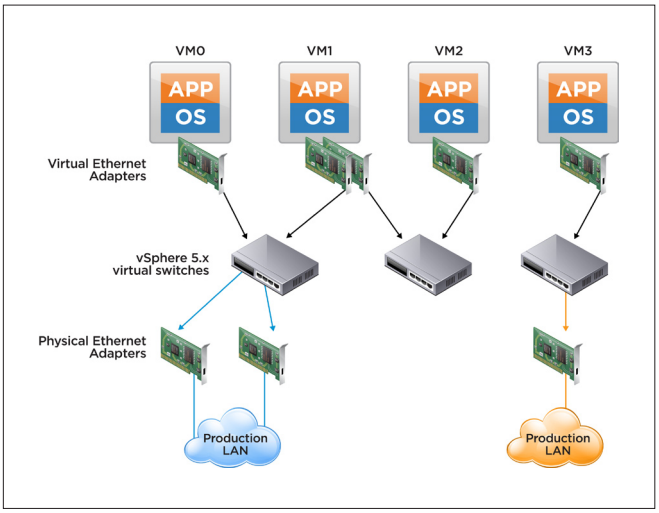


Figure 1. Network I/O Virtualization and Virtual Networking

Instead of directly accessing physical PNICs, a VM can be configured with one or more virtual NICs (VNICs) that are exported by the VM hardware [3]. Accessing VNICs incurs overhead because it needs an intervention of the virtualization layer. Further, to properly connect VMs to the physical networks (i.e., rout packets between VNICs and PNICs), or connect them to one another (i.e., rout packets between VNICs and VNICs), a switching layer is necessary, which essentially forms *virtual networks*. Network I/O virtualization therefore requires extra layers of network packet processing that perform two important operations: NIC virtualization and virtual switching. Packet sending

and receiving both involve these two operations of accessing VNICs and going through the virtual switching layer, the cost of which is directly added to the response time—a cost that does not show up in a native setup.

In vSphere, the *virtual machine monitor* (VMM) provides the VM hardware for the guest, and VMkernel manages physical hardware resources. When the guest accesses VNICs, for instance, to send a packet, the VMM must intervene and communicate with VMkernel, which finds the right PNIC to put the packet out on the wire. When packet transmission is completed or a packet is received, on the other hand, VMkernel first processes such an event and notifies the right VM (with the right VNIC) through the VMM. All the processing done in VMM and VMkernel combined is extra work, and hence counts toward the virtualization overhead. Throughout this paper, the virtualization layer combining both VMM and VMkernel is sometimes collectively referred to as *hypervisor*.

As noted, the communication between VMM and VMkernel happens in two ways: (1) VMM calling into VMkernel to send a packet and (2) VMkernel notifying VMM of an event for the guest. vSphere reduces CPU cost by batching (i.e., coalescing) the communication between VMM and VMkernel involving network I/Os in both directions. This also adds an extra delay and variance to the response time. Throughout this paper, this feature is disabled to simplify the analysis and reduce variance in the response-time measurement.

3.2 CPU Virtualization: Virtual CPU Halt/Wake-Up

Like other resources such as NICs, CPUs also need to be virtualized. VMs do not exclusively own physical CPUs, but they are instead configured with virtual CPUs (VCPUs) provided by the VM hardware. VMkernel's proportional-share-based scheduler enforces a necessary CPU time to VCPUs based on their entitlement [4]. This is an extra step in addition to the guest OS scheduling work. Any scheduling latency in the VMkernel's CPU scheduler therefore is added to the response-time overhead.

One common characteristic of RR workloads is that the VCPU becomes idle after a packet is sent out, because it must wait for a subsequent request from the client. Whenever the VCPU becomes idle and enters a halted state, it is removed from the scheduler because it does not need CPU time anymore. When a new packet comes in, the descheduled VCPU must be rescheduled when the CPU scheduler runs to perform necessary operations to make the VCPU ready again. This process of descheduling and rescheduling of a VCPU performed at each transaction incurs a nonnegligible latency that is added to the constant overhead in RR workload's response time. VCPU halting (and descheduling) cost might have less impact on response-time overhead, because it needs to wait for the response from the client anyway.

Note that this paper mainly studies configurations in which CPU is not overcommitted such that each VCPU is able to get its own physical core. Therefore, VCPU ready time, which is usually one of the biggest contributors to response-time overhead, is minimal and thus does not add much to response-time overhead.

4. Variable Response-Time Overhead

The virtualization overhead in response time with RR workloads is not completely constant. Instead, the gap in response time between VM and native setups increases with regard to the workload response time itself. Such an increase mostly comes from running the guest code. Even though the guest runs most of the time without the intervention of the hypervisor (i.e., the CPU executes guest instructions directly), the overhead is still observed. This paper demonstrates that the main causes of the variable overhead component are attributable to (1) the use of hardware-assisted MMU virtualization and (2) an increase in cache misses that is seemingly a byproduct of hardware-assisted MMU virtualization. Detailed evaluation and analysis results are presented later in this paper.

One of the major challenges in virtualizing MMU is that the physical memory perceived by the guest is different from the real machine memory in the host. This arises from the very same reason that the memory in the host also needs to be shared between VMs, like any other resources. Figure 2 describes how guest virtual pages (logical pages), guest physical pages, and machine pages are related to one another on the vSphere platform. Before hardware-assisted MMU virtualization was introduced, the mappings of guest virtual page numbers (VPNs) to machine page numbers (MPNs) in the host were stored in shadow page tables that are implemented in software and managed by VMM [1]. However, to maintain fresh information, the shadow page tables must be updated whenever the guest changes its page tables.

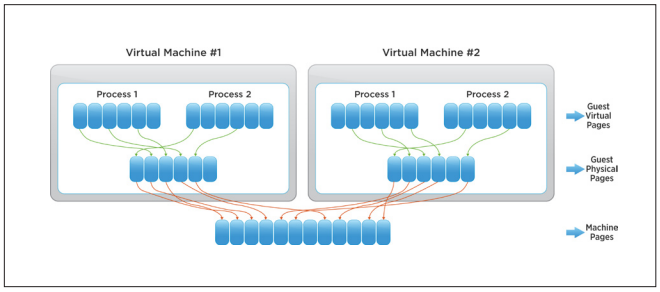


Figure 2. MMU Virtualization

Both Intel and AMD introduced a hardware-assisted MMU virtualization mechanism to overcome the drawback of the software-based MMU virtualization approach. For example, Intel introduces a separate page table structure, *Extended Page Tables* (EPTs), to maintain the mappings of VMs' physical page numbers (PPNs) to MPNs in hardware [6]. EPT improves performance considerably by avoiding calling into VMM too frequently to update shadow page tables. Because the approaches used by both Intel and AMD are similar, this paper takes Intel's mechanism, EPT, as an example of hardware-assisted MMU virtualization.

The overhead of hardware-assisted MMU virtualization, however, becomes more visible when translation lookaside buffer (TLB) misses happen and the hardware page walker walks both EPTs

and guest page tables. With hardware-assisted MMU virtualization, TLB misses become more costly and the cycles spent walking EPT become pure overhead. Even worse, page walks for two different page table structures (guest page tables and EPTs) might increase the data cache footprint, taking away CPU cycles to handle more cache misses.

Assuming (just for the sake of analysis) that the TLB miss rate remains the same when guest instructions are executed, the overhead caused by using EPT and associated cache misses increases in proportion to the duration of the guest code execution. In other words, the longer the guest code runs (i.e., the longer the response time of the workload), the more overhead will be shown in response time. This explains why there is a variable component in the response-time overhead of virtualization. The constant overhead will be dominant in a simpler workload, whereas the variable overhead becomes dominant in a more complicated workload (i.e., a workload with a longer response time).

5. Evaluating Response-Time Overhead with Five Different RR Workloads

This section evaluates the response-time overhead of five different RR workloads on vSphere (i.e., VM configuration) versus a native machine (native configuration).

5.1 Test Bed

The test bed (shown in Figure 3) consists of one server machine for serving RR workload requests and one client machine that generates RR requests. The server machine is configured with dual-socket, quad-core 3.47GHz Intel Xeon X5690 processors and 64GB of RAM, and the client machine is configured with dual-socket, quad-core 3.40GHz Intel Xeon X5690 processors and 48GB of RAM. Both machines are equipped with a 10GbE Broadcom NIC. Hyperthreading is not used.

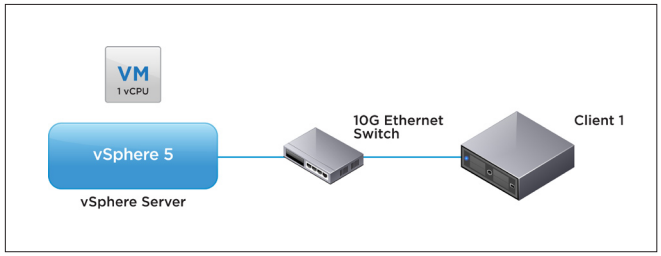


Figure 3. Test-Bed Setup

Two different configurations are used to compare against each other. A native configuration runs native Red Hat Enterprise Linux (RHEL) 6 on both client and server machines, and a VM configuration runs vSphere on the server machine and native RHEL6 on the client machine. The vSphere instance hosts a VM that runs the same version of RHEL6 Linux. For both the native and VM configurations, only one CPU (VCPU for the VM configuration) is configured to remove the impact of parallel processing and discard any multi-CPU-related overhead. In both configurations, the client machine is used to generate RR workload requests, and the server machine is used to serve the requests and send replies to the client. VMXNET3 is used for virtual NICs in the VM configuration. A 10 Gigabit Ethernet switch is used to interconnect the two machines. The detailed hardware setup is described in Table 1.

	SERVER	CLIENT
Make	HP ProLiant DL380 G7	Dell PowerEdge R710
CPU	2X Intel Xeon X5690 @ 3.47GHz, 4 cores per socket	2X Intel Xeon X5690 @ 3.40GHz, 4 cores per socket
RAM	64GB	48GB
NICs	1 Broadcom NetXtreme II 10Gbps adapter	1 Broadcom NetXtreme II 10Gbps adapter

Table 1. Test-Bed Hardware Specification

5.2 Workload

Five different request-response workloads are used to evaluate and analyze the response-time overhead: (1) Ping, (2) Netperf_RR, (3) Gemfire_Get, (4) Gemfire_Put, and (5) Apache.

- Ping – Default ping parameters are used except that the interval is set to .001, meaning that 1,000 ping requests are sent out every second.
- Netperf_RR – The Netperf micro benchmark [8] is used to generate an RR workload in TCP.
- Gemfire_Get – VMware vFabric™ GemFire™ [9] is a Java-based distributed data management platform. Gemfire_Get is a benchmark workload that is built with GemFire wherein the client node sends a request with a key to extract an object stored in server nodes. Two server nodes are replicated, and the client node already knows which server node to contact to extract the object with a given key. All nodes are Java processes. The client node runs in the client machine, and the server nodes run in the server machine (or in a VM for the VM configuration).
- Gemfire_Put – Gemfire_Put is a similar to Gemfire_Get except that the client node sends an object to a server node that stores it in its in-memory database. Because two server nodes are running, the object sent to one server node is replicated to the other one. Although two hops of communication between nodes (from the client node to a server node to the other server node) occur per transaction, there happens to be only one pair of request-response network I/O, because the two server nodes reside in the same machine (or in the same VM for the VM configuration).
- Apache – The client generates HTTP traffic that is sent to the Apache Web server [10]. The Apache Web server is configured to run only one server process so that there is always one transaction handled at a time.

In all five workloads, the request and response sizes are configured to be less than the MTU so that they can be put into one Ethernet packet.

5.3 Method

The five different workloads are used to study the overhead in the response times of RR workloads. Each workload runs for 120 seconds, and the average response time is measured in both the

VM and native configurations. When the response times of the two configurations are measured, the overhead can be easily calculated by taking the difference of the two. Because only the server is virtualized in the VM configuration, "virtualization overhead" implies the overhead of virtualizing the server application of the workload.

	PING	NETPERF	GEMFIRE_ GET	APACHE	GEMFIRE_ PUT
Native	26 us	38 us	72 us	88 us	134 us
VM	39 us	52 us	88 us	106 us	157 us

Table 2. Response Times of Five RR Workloads

5.4 Results

Table 2 compares the response times of the five workloads between the native and VM configurations. Ping exhibits the lowest response time because it is the simplest workload. Gemfire_Put shows the highest response time. The overhead of Ping is 13us, and the overhead of Gemfire_Put is 23us. Figure 4 takes a closer look at the response-time overhead of the VM configuration by plotting both the absolute overhead (i.e., the absolute difference between the two configurations) and the relative one. It is interesting to observe that the absolute overhead is obviously not constant but increases for a workload with a longer transaction-response time. On the contrary, the relative overhead moves in the opposite direction. This implies that a nonnegligible amount of constant overhead also exists.

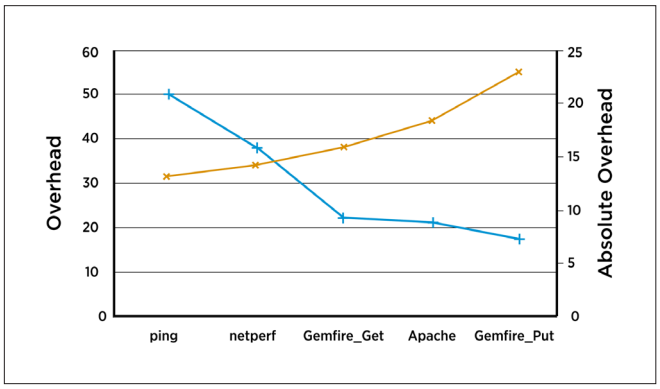


Figure 4. Response-time overhead in relative and absolute terms

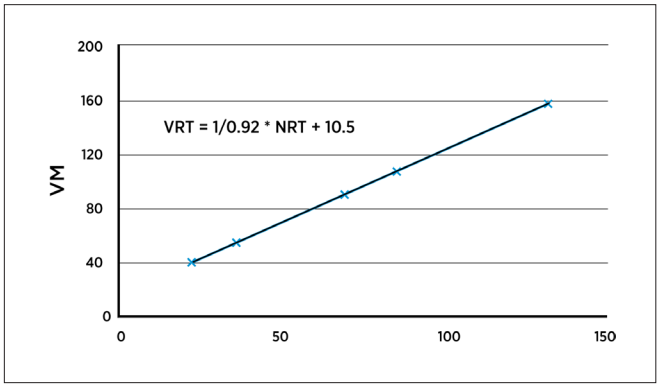


Figure 5. Linear regression on response time overhead of five workloads

5.5 Trend in Response-Time Overhead

To get a better picture of the trend in the response-time overhead across multiple RR workloads, linear regression is applied to curve-fit the overhead of the five different workloads. Linear regression is a simple but very useful method for identifying a trend in the observed data set.

The regression result is shown in Figure 5. It shows an almost perfect line with $R^2 = 0.999$. An R^2 value indicates how well the regression line fits the data points. An R^2 of 1.0 means a perfect fit of the regression line; 0 means that there is no linear relationship. It is somewhat interesting to see a strong linear relationship with five different workloads. According to the linear regression, the relation between VM and native response times for these five workloads can be expressed as follows:

$$VRT = 1.09 \times NRT + 10.5 . \quad (1)$$

VRT denotes the average response time of the VM configuration, and NRT denotes the average response time of the native configuration. Rearranging the equation to calculate the actual overhead yields the following equation:

$$Overhead = VRT - NRT = 1.09 \times (NRT - 1) + 10.5 . \quad (2)$$

Simply interpreting Equation 2 tells us that the VM's response-time overhead increases with the native machine's response time and has a nonnegligible constant term of 10.5us. Therefore, with the previously described five workloads, the response-time overhead can be divided into two different components: (1) a constant overhead and (2) a variable part that monotonically increases with regard to the complexity of workload. (i.e., the longer the workload runs, the larger the overhead becomes.)

Further, rewriting the first term of Equation 1 by replacing 1.09 with 1/0.92 (1.09 = 1/0.92) raises an interesting insight into better interpreting the variable part of the overhead.

$$VRT = 1/0.92 \times NRT + 10.5 . \quad (3)$$

Equation 3 says that the VM configuration runs more slowly at 92% of the native configuration's speed, which causes it to take 9% more response time (the 1.09 coefficient in Equation 1 indicates this), with an additional 10.5us of overhead. The existence of the variable overhead component (i.e., the first term in Equation 1 and Equation 3), therefore, indicates that the VM configuration effectively runs more slowly than the native configuration. The following subsections take a closer look at each component.

5.6 Constant Cost

It is previously explained that network I/O virtualization and VCPU halt/wake-up latency are the major elements of the constant overhead in response time for RR workloads. This section studies the detailed breakdown of the constant overhead. Three RR workloads (NetPerf, Gemfire_Get, and Gemfire_Put) are picked for the study, because all five RR workloads show exactly the same overhead trend as shown in Equation 3. Among the two main sources for the constant overhead, VCPU scheduling is first studied, followed by network I/O virtualization.

The amount of the overhead in response time due to VCPU halt/wake-ups per transaction can be indirectly calculated by removing such VCPU halt/wake-up operations. vSphere provides an option of preventing the VCPU from getting descheduled. The VCPU will be spinning instead, even when it is in a halted state, hence bypassing the VMkernel CPU scheduler. Table 3 shows that the response time gets reduced by around 5us across the three workloads that were evaluated, which is roughly a half of the total 10.5us of the constant overhead. Linear regression gives the following equation with $R^2 = 0.999$:

$$VRT = 1/0.92 \times NRT + 5.3 . \quad (4)$$

Equation 4 says that removing VCPU halt/wake-up reduces the constant cost by 5us. The relative speed of the VM configuration (i.e., the variable part) does not change, staying at 92% of the native configuration. This indicates that the VCPU halt/wake-up cost is indeed a significant part of the constant cost in response-time overhead, and it does not contribute to the variable overhead component in a meaningful way.

	NETPERF	GEMFIRE_GET	GEMFIRE_PUT
VM	53 us	43 us	116 us
VM with monitor_control.desched=false	48 us	46 us	121 us

Table 3. Response of three RR Workloads on VM with and without monitor_control.desched=false

As explained, network I/O virtualization can be divided into two different components: NIC virtualization and virtual switching. Among the two overhead sources, the virtual switching cost is first studied. A packet-tracing tool (developed internally) is used that reports the time spent in doing the virtual switching. The measured processing time of packet transmission and reception together is constant for the three workloads, taking 3us. Combining the 5us of VCPU halt/wake-up overhead with 3us of the virtual switching takes 8us out of total 10.5us of the constant overhead obtained from Equation 3. This leaves us 2.5us to explain further.

This remaining 2.5us is likely to come from NIC virtualization. When the guest accesses the virtual NIC to send/receive a packet or handle virtual interrupts for packet reception and completion, the VMM must intervene. This adds overhead that must have a constant component associated with it.

To confrm the 2.5us of NIC virtualization overhead, the time spent in the guest was measured using `tcpdump`. The measured time includes the time spent in the VMM handling VNIC accesses on behalf of the VCPU and the direct execution of guest instructions, but it does not include the time spent in VMkernel performing virtual switching and accessing the physical device. Applying linear regression to the measured guest time yields the following linear equation with $R^2 = 1.000$, confirming that there is indeed a constant overhead of 2.5us:

$$VRT = 1/0.93 \times NRT + 2.5 . \quad (5)$$

In summary, the constant overhead in response time incurred is completely broken down into (1) VCPU halting/wake-up overhead and (2) network I/O virtualization overhead that further consists of the virtual switching and NIC virtualization overhead.

Because VCPU halting/wake-up and the virtual-switching overhead are the main contributors to the time spent in VMkernel and they are constant, the variable cost should be added from layers other than VMkernel. In other words, the variable part of the overhead arises when the guest is running (the guest code directly executes or the VMM runs on behalf of the guest). This is also corroborated by Equation 5, which shows that the time spent in the guest (excluding VMkernel time) has indeed the (almost) identical variable component as the ones shown in Equation 3 and Equation 4. Further, because the time spent in the VMM for accessing VNICs is already (likely to be) captured in the constant overhead, the variable overhead in response time should mostly happen when guest instructions directly execute. The next section studies the variable component of the response-time overhead in detail.

5.7 Variable Cost

To analyze (1) why the overhead is added when guest instructions directly execute (without much intervention of the hypervisor) and (2) how executing guest instructions contribute to the variable overhead, hardware performance counters are used. Hardware performance counters provide an effective way to understand where CPU cycles are spent.

Note that there is a noticeable difference in the code path executed between the native and VM configurations, mainly because of the difference in the device drivers. The VM configuration uses a paravirtualized driver, VMXNET3, and the native configuration uses a native Linux device driver. This makes it hard to compare performance counters of the two configurations fairly because they execute different guest code. For these reasons, a simpler workload is used to find any difference in hardware performance counters between the two configurations. By running both the client and server applications on the same machine (or on the same VM for the VM configuration) with one CPU configured, the two configurations get to execute the same (guest) code. Any network I/Os are removed while the (guest) kernel TCP/IP code is still exercised. The workload becomes completely CPU-intensive, avoiding halting and waking up the VCPU and removing the difference in the halt/wake-up path. Another aspect of this kind of workload is that guest instructions directly execute without the intervention of the hypervisor 99% of the time. It is not 100% because the hypervisor sometimes must run, mainly to handle timer interrupts. Without the hypervisor overhead, the constant overhead is almost entirely removed.

	SIMPLIFIED NETPERF
Native	8.7 us
VM	9.2 us

Table 4. ResponseTime Comparison Using a Simplified Netperf RR Workload with No Network I/Os

The Netperf RR workload is chosen and simplified to compare hardware performance counters between the VM and native configurations. Before presenting the hardware performance counter comparison, Table 4 shows the response-time comparison

of the simplified workload between the native and VM configurations. An overhead in response time (of 6%) still shows up even when the workload becomes much simpler without the hypervisor costs.

	NATIVE	VPMC-GUEST (VM)	VPMC-HYBRID (VM)
IPC	0.70	0.64	0.64
Unhalted Cycles	$376 * 10^9$	$406 * 10^9$	$406 * 10^9$
# Instructions	$261 * 10^9$	$258 * 10^9$	$259 * 10^9$

Table 5. Performance Counters Comparison Using a Simplified Netperf RR Workload with No Network I/Os

	NATIVE	VPMC-HYBRID (VM)
IPC	0.70	0.64
Unhalted Cycles	$376 * 10^9$	$406 * 10^9$
# Instructions	$261 * 10^9$	$259 * 10^9$
TLB-misses-walk-cycles	$22.7 * 10^9$	$25.5 * 10^9$
EPT-walk-cycles	0	$16.4 * 10^9$
L1-dcache-misses-all	$4.53 * 10^9$	$6.14 * 10^9$

Table 6. More Performance Counters Comparison Using a Simplified Netperf RR Workload with No Network I/Os

Table 5 and Table 6 compare the major performance counters that are collected using a Linux perf profiling tool [11]. Two different modes of Virtual Performance Monitoring Counters (VPMC) [7] are used to collect performance counters for the VM: VPMC-Guest and VPMC-Hybrid. In the VPMC-Guest configuration, performance counters are updated only when guest instructions execute directly on the PCPU. They do not increment when the PCPU runs the hypervisor code (on behalf of the guest). On the other hand, in the VPMC-Hybrid configuration, all counters increment regardless of whether the PCPU is running guest or hypervisor instructions, except for the *instructions retired* and *branches retired* counters. The *instructions retired* and *branches retired* events count guest instructions only. The VPMC-Hybrid configuration is useful for measuring the overhead of the virtualization layer using a metric such as Instructions Per Cycles (IPC), because *unhalted cycles* includes the cycles spent in the hypervisor, whereas *instructions retired* counts only guest instructions.

The test runs for a fixed number of iterations (i.e., transactions) and hardware counters are collected during the entire period of a run. Because both the native and VM configurations execute the same (guest) instructions, the i*nstructions retired* counters are also the same. Extra cycles spent in the VM (compared to the native machine), therefore, become the direct cause of the increase in response time. Understanding where and why the extra cycles are spent in the VM configurations is the key to analyzing why the VM runs more slowly.

Table 5 compares IPC, *unhalted cycles*, and *instructions retired* between the native and the two VM configurations (two different VPMC modes). Note that, whereas VPMC-Guest counts only the cycles spent in the guest, the VPMC-Hybrid configuration counts the cycles spent in the hypervisor also. The fact that there is not much difference in the *unhalted cycles* between the two VM configurations confirms that the hypervisor overhead (i.e., the time spent in executing the hypervisor code) is minimal in this workload. This further indicates that the 8% increase in the cycles compared to the native configuration must have come when the guest instructions executed directly. With the same number of instructions executed but more cycles spent, the VM configurations yield an IPC that is lower by 9%, indicating that the VM indeed runs more slowly than the native machine.

From Table 6, the biggest contributor of the increased cycles when guest instructions directly execute are the *Extended Page Table (EPT) walk cycles*. This takes 4% out of the 8% cycles increase. The use of an additional page table mechanism (i.e., EPT) to keep track of the mappings from PPN and MPN requires extra computing cycles when TLB misses occur. The *EPT walk cycles* count these extra cycles. Interestingly, L1/L2/L3 data cache misses also increased. (Only L1 data cache misses are shown in Table 6). 3% out of 8% comes from L1/L2/L3 cache misses, with L1 data cache misses constituting the majority. It is suspected that additional page table structures (EPT page tables) seemingly incur more memory accesses, increasing L1 data cache misses. An increase in TLB cache misses (excluding EPT walk cycles) takes the remaining 1% of the 8% increase. This is likely to be due to more memory accesses.

The results show that in this simplified workload, hardware-assisted MMU virtualization directly causes most of the cycle increase, compared to the native configuration. It is also suspected that the extra cycles spent due to more data cache misses and TLB misses are seemingly a side-effect of using hardware MMU virtualization. Therefore, it can be concluded that hardware-assisted MMU virtualization is the main reason for the variable response-time cost.

A software MMU virtualization method such as those with shadow page tables might also incur an overhead contributing to the variable overhead component. Its overhead might in fact be higher than that of hardware MMU virtualization. Comparison between software and hardware-based MMU virtualization approaches is out of this paper's scope.

Figure 5 shows that there is a strong linear relationship in response-time overhead with five different RR workloads. Based on the analysis in this section, this is probably because all five workloads exercise a similar code path (i.e., Linux kernel network stack and the VNIC device driver routine) and the TLB miss rate is similar, therefore leading to a linearly increasing variable cost.

6. More Discussion

6.1 Impact of Using Direct Path I/O

With VMDirectPath I/O [22] configured, a VM is given complete and exclusive access to PNICs installed on the vSphere host. Because the VM directly owns the device, neither a VNIC nor the virtual switching layer is needed. Figure 6 shows the regression results of three different workloads with VMDirectPath I/O. *monitor_control.desched=false* is specified in a VMX file to remove the CPU scheduling overhead. Removing the halt/wake-up and network I/O virtualization costs,

the constant overhead from the regression result is virtually gone—the negative constant term might exist because there are not enough data points. Having exclusive and direct access to the hardware (i.e., CPU and NIC) removes the associated virtualization overhead and further prevents the response time from arbitrarily being increased due to resource contention. The drawback is that resources are not shared and thus can be wasted.

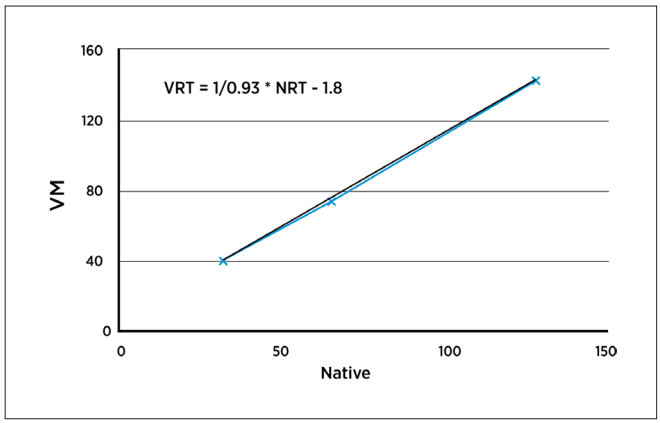


Figure 6. Linear Regression on Response Time: Native and VM with VMDirectPath I/O

6.2 Response-Time Overhead in Using VSMP

With multiple VCPUs, more sources of response-time overhead come into play, even when resources, especially CPUs, are not overcommitted. Many applications employ multiple processes/threads, resulting in multiple VCPU halt/wake-up operations per transaction to communicate and wake one another up. Such extra halt/wake-up operations will add more latency to the response time accordingly. For example, the Gemfire_Put workload used in this paper has two server nodes (two separate Java processes). They are likely to be distributed, running on a different VCPU. Consequently, three VCPU halt/wake-up operations instead of one pair occur per transaction, because there is one extra hop of communication between the two server nodes.

	NETPERF	GEMFIRE_GET	GEMFIRE_PUT
4 CPU Native	39 us	72 us	120 us
4 VCPU VM	54 us	89 us	159 us
4 VCPU VM with monitor_control.desched=false	49 us	83 us	143 us

Table 7. Response Times of Three RR Workloads on 4-VCPU VM with and Without **monitor_control.desched=false**

Table 7 presents response times of three RR workloads on a 4-VCPU VM with and without *monitor_control.desched=false*. The native configuration is also shown for the purpose of comparison. The difference in response time between the two VM configurations is around 5us for Netperf and Gemfire_Get, whereas Gemfire_Put shows 16us of difference. Indeed, the estimated cost of VCPU halt/wake-up operations with Gemfire_Put (16us) is three times that of Netperf and Gemfire_Get (5–6us) with three VCPU halt/wake-up operations instead of just one.

Waking up another CPU generates an inter-processor interrupt (IPI) that is sent to the target CPU. This process requires access to hardware, the local Advanced Programmable Interrupt Controller (APIC) of communicating CPUs. Sending and receiving IPIs in a VM (between VCPUs), therefore, are properly virtualized and managed by the hypervisor, and this adds a certain overhead. To better understand the impact of the overhead of sending and handling IPIs in a virtualized setup, the workload used in Table 4 and Table 5 (netperf TCP_RR traffic without network I/Os on a 1-VCPU VM) are modified a little bit such that the client and server run separately on a different VCPU of a 2-VCPU VM. The modified workload maintains the same characteristic of not having any network I/Os, but the client and the server must wake each other up to complete a transaction that incurs IPI traffic between the two VCPUs. Three different configurations are used: (1) Native, (2) VM, and (3) VM with *monitor_control.desched=false* configurations. Results are shown in Table 8.

	NETPERF
Native	8.3 us
VM	26 us
VM with monitor_control.desched=false	15 us

Table 8. Response-Time Comparison with Simplified Workloads with No Network I/Os on 2-VCPU VM

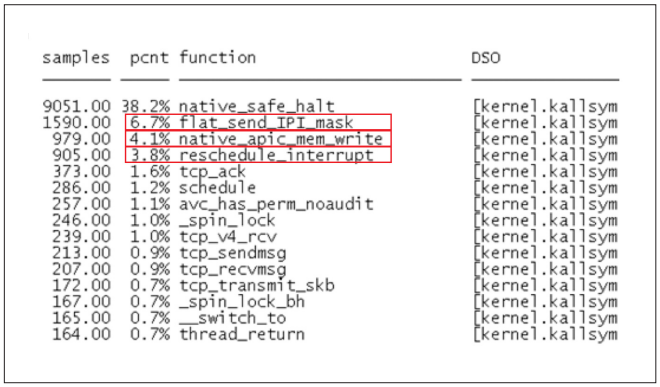
In this microbenchmark, two VCPU halt/wake-up operations happen per transaction because the client and server sit on a different VCPU. Because no network I/Os are incurred, the transaction rate is relatively higher, causing a very high number of IPIs. Using *monitor_control.desched=false* saves the cost of two VCPU halt/wake-up operations, improving by 11us (compared to the default VM configuration). However, even after using the option, the response time (15us) is still behind the native configuration (8.3us). One of the main reasons for the still remaining overhead in response time seems to be the IPI-related cost. Figure 7 and Figure 8 show performance-profiling results using Linux perf in the native and VM configurations respectively. For the VM configuration, the VPMC-Hybrid mode is used such that any hypervisor overhead is shown in the guest

```
samples  pcnt function                    DSO
_______ _____ ___________________________ _____________

 465.00  3.3% tcp_ack                     [kernel.kallsym
 364.00  2.6% schedule                    [kernel.kallsym
 277.00  2.0% _spin_lock                  [kernel.kallsym
 272.00  2.0% mod_timer                   [kernel.kallsym
 270.00  1.9% _spin_lock_irqsave          [kernel.kallsym
 253.00  1.8% tcp_recvmsg                 [kernel.kallsym
 229.00  1.6% tcp_sendmsg                 [kernel.kallsym
 225.00  1.6% tcp_transmit_skb            [kernel.kallsym
 218.00  1.6% thread_return               [kernel.kallsym
 213.00  1.5% __switch_to                 [kernel.kallsym
 184.00  1.3% try_to_wake_up              [kernel.kallsym
 183.00  1.3% resched_task                [kernel.kallsym
 182.00  1.3% _spin_lock_bh               [kernel.kallsym
 179.00  1.3% net_rx_action               [kernel.kallsym
 177.00  1.3% __netif_receive_skb         [kernel.kallsym
 176.00  1.3% sk_wait_data                [kernel.kallsym
 165.00  1.2% __alloc_skb                 [kernel.kallsym
```

Figure 7. Profiling Data with Simplified Workloads with No Network I/Os on Native (with 2 CPUs Configured)

profiling data. Indeed, in the VM configuration shown in Figure 8, the cost of IPI-related kernel functions (*flat_send_IPI_mask*, *native_apic_mem_write*, *reschedule_interrupt*) takes 15% of CPU time used, indicating that it is one of the major reasons for the response-time overhead together with VCPU halt/wake-up cost in this microbenchmark. Those samples do not show in the native profiling data in Figure 7.



samples	pcnt	function	DSO
9051.00	38.2%	native_safe_halt	[kernel.kallsym
1590.00	6.7%	flat_send_IPI_mask	[kernel.kallsym
979.00	4.1%	native_apic_mem_write	[kernel.kallsym
905.00	3.8%	reschedule_interrupt	[kernel.kallsym
373.00	1.6%	tcp_ack	[kernel.kallsym
286.00	1.2%	schedule	[kernel.kallsym
257.00	1.1%	avc_has_perm_noaudit	[kernel.kallsym
246.00	1.0%	_spin_lock	[kernel.kallsym
239.00	1.0%	tcp_v4_rcv	[kernel.kallsym
213.00	0.9%	tcp_sendmsg	[kernel.kallsym
207.00	0.9%	tcp_recvmsg	[kernel.kallsym
172.00	0.7%	tcp_transmit_skb	[kernel.kallsym
167.00	0.7%	_spin_lock_bh	[kernel.kallsym
165.00	0.7%	__switch_to	[kernel.kallsym
164.00	0.7%	thread_return	[kernel.kallsym

Figure 8. Profiling Data with Simplified Workloads with No Network I/Os on 2-VCPU VM

7. Related Work

The performance overhead of virtualization has been thoroughly evaluated. While doing so, researchers investigated ways to reduce the network I/O virtualization overhead [13] [17] [18] [14][21].

In [13], Menon et al. worked on the optimization of network I/O virtualization in Xen to reduce the performance gap between native and virtual configurations using various techniques. Cherkasova et al. [17] measured CPU overhead for processing HTTP workloads in Xen. Gamage et al. studied a low TCP throughput issue in a heavily consolidated virtualized environment [18]. They argued that the reason for the lowered throughput is that the TCP window does not open up as quickly as in a native configuration due to an increased RTT caused by CPU contention among multiple VMs. They proposed a way to get around the issue by allowing packets to be flooded from the VM to the driver domain.

All of the above papers studied network I/O virtualization overhead in terms of either CPU cost or throughput. In contrast to the above papers, our paper focuses on the response-time overhead of request-response workloads, which are the most common type of latency-sensitive applications.

In [14], Sugerman et al. described I/O virtualization architecture and evaluated its performance on VMware Workstation™. It investigated the breakdown of the latency overhead imposed by the virtualization layer. In contrast to our paper, however, it only focused on a hosted architecture, whereas our paper studies vSphere, which employs a bare-metal approach.

In [21], Ongaro et al. studied the effect of VCPU scheduling on I/O performance in Xen. They used various workloads including latency-sensitive ones. Their work is orthogonal to our paper, because they focused on the CPU scheduler's impact on performance (i.e., latency) when multiple VMs are running, whereas our paper studies the response-time overhead with one VM when the system is lightly loaded. Furthermore, they didn't compare the response time results to those of a native configuration.

Serebrin et al. introduced virtual performance monitoring counters (VPMCs), which enable performance counters for VMs on the vSphere platform [7]. Profiling tools that use hardware performance counters do not work on VMs (i.e., guests) with most hypervisors. With VPMC, existing profiling tools can be used inside a VM to profile applications without any modifications. Similarly, in [20] Menon et al. presented a system-wide statistical profiling toolkit for Xen. This work is similar to [7] in that it enables the accurate profiling of guest applications and the hypervisor overhead. Our paper uses VPMC to profile latency-sensitive workloads.

In [7], the authors studied and quantified the increased cost of handling TLB misses in virtualization due to the hardware-assisted MMU using tier 1 applications and a network microbenchmark. Focusing on latency-sensitive workloads, this paper takes a deeper look at the gap between virtual native performance and gives a complete breakdown of the response-time overhead of running latency-sensitive workloads in VMs.

8. Conclusion

This paper investigated the response-time overhead of latency-sensitive workloads, especially request-response workloads, in vSphere. With this type of workload, any overhead that the virtualization layer imposes is exposed directly in the response time with every step of extra operation serialized. This paper analyzed the breakdown of the sources of the response-time overhead observed in such workloads.

Specifically, this paper presented that the virtualization overhead in the response-time of a latency-sensitive workload is not entirely constant as expected. Instead, it also has a variable component that increases with regard to the response time of the workload itself. Using five different request-response workloads, this paper further demonstrated that the constant overhead incurs due to extra layers of virtualizing network I/Os and CPUs, while the cost of using another page table structure with hardware-assisted MMU virtualization is the main reason for the variable part of the overhead.

Acknowledgments

The authors would like to thank Dan Hecht, Garrett Smith, Haoqiang Zheng, Julie Brodeur, Kalyan Saladi, Lenin Singaravelu, Ole Agesen, Ravi Soundararajan, Seongbeom Kim, Shilpi Agarwal, and Sreekanth Setty for reviews and contributions to the paper.

References

1 K. Adams and O. Agesen. “A comparison of software and hardware techniques for x86 virtualization.” In Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct. 2006.

2 Thomas J. Bittman, George J. Weiss, Mark A. Margevicius, and Philip Dawson. “Magic Quadrant for x86 Server Virtualization Infrastructure,” Gartner Group, June 11, 2012.

3 What’s New in VMware vSphere 4: Virtual Networking. VMware, Inc., 2009. http://www.vmware.com/files/pdf/techpaper/Performance-Networking-vSphere4-1-WP.pdf .

4 VMware vSphere: The CPU Scheduler in VMware ESX 4.1. VMware, Inc., 2011. http://www.vmware.com/files/pdf/techpaper/VMW_vSphere41_cpu_schedule_ESX.pdf.

5 Understanding Memory Resource Management in VMware vSphere 5.0. VMware, Inc., 2011. http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf

6 Performance Evaluation of Intel EPT Hardware Assist. VMware, Inc., 2008. http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.

7 Benjamin Serebrin, Daniel Hecht, “Virtualizing Performance Counters.” 5th Workshop on System-Level Virtualization for High Performance Computing (HPCVirt 2011), August 2011.

8 Netperf. www.netperf.org, 2011. http://www.netperf.org/netperf/.

9 VMware vFabric GemFire. VMware, Inc., 2012. http://www.vmware.com/products/vfabric-gemfire/overview.html.

10 Apache HTTP Server Project. The Apache Software Foundation. http://httpd.apache.org.

11 perf: Linux profiling with performance. https://perf.wiki.kernel.org/index.php/Main_Page.

12 Leonard Kleinrock. *Queueing Systems. Volume 1: Theory.* Wiley-Interscience, 1975.

13 A Menon, AL Cox, and W Zwaenepoel. “Optimizing network virtualization in Xen.” In Proceedings USENIX ‘06 Annual Technical Conference.

14 Jeremy Sugerman, Ganesh Venkitachalam, and BengHong Lim. “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor.” In Proceedings of the 2001 USENIX Annual Technical Conference, June 2001.

15 J.Walters, V. Chaudhary, M. Cha, S. Guercio, and. S. Gallo. “A Comparison of Virtualization Technologies for HPC.” In Proceedings of the 22nd International Conference on Advanced Information Networking and Applications (AINA). Pp. 861–868. Okinawa 25–28, Mar., 2008.

16 J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, “Bridging the gap between software and hardware techniques for IO virtualization,” in ATC’08: In Proceedings of the USENIX 2008 Annual Technical Conference, pp. 29–42, 2008.

17 L. Cherkasova and R. Gardner, “Measuring CPU overhead for I/O processing in the Xen virtual machine monitor.” In Proceedings of the 2005 USENIX Annual Technical Conference, 2005, pp. 387–390.

18 Gamage, S., Kangarlou, A., Kompella, R. R., Xu, D. 2011. “Opportunistic flooding to improve TCP transmit performance in virtualized clouds.” In Proceedings of the Second ACM Symposium on Cloud Computing(October).

19 Aravind Menon and Willy Zwaenepoel. “Optimizing TCP receive performance.” In Proceedngs of the 2008 USENIX Annual Technical Conference, June 2008.

20 A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. “Diagnosing performance overheads in the Xen virtual machine environment.” VEE ’05, June 2005.

21 D. Ongaro, A. L. Cox, and S. Rixner. “Scheduling I/O in virtual machine monitors” VEE ‘08, 2008.

22 VMware VMDirectPath I/O http://communities.vmware.com/docs/DOC-11089.

23 J. Buell, D. Hecht, J. Heo, K. Saladi, and H.R. Taheri. “Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications.” VMTJ Summer 2013, 2013.

Analyzing PAPI Performance on Virtual Machines

John Nelson
University of Tennessee
john.i.nelson@gmail.com

Abstract

Performance Application Programming Interface (PAPI) aims to provide a consistent interface for measuring performance events using the performance counter hardware available on the CPU as well as available software performance events and off-chip hardware. Without PAPI, a user may be forced to search through specific processor documentation to discover the name of processor performance events. These names can change from model to model and vendor to vendor. PAPI simplifies this process by providing a consistent interface and a set of processor-agnostic preset events. Software engineers can use data collected through source-code instrumentation using the PAPI interface to examine the relation between software performance and performance events. PAPI can also be used within many high-level performance-monitoring utilities such as TAU, Vampir, and Score-P.

VMware® ESXi™ and KVM have both added support within the last year for virtualizing performance counters. This article compares results measuring the performance of five real-world applications included in the Mantevo Benchmarking Suite in a VMware virtual machine, a KVM virtual machine, and on bare metal. By examining these results, it will be shown that PAPI provides accurate performance counts in a virtual machine environment.

1. Introduction

Over the last 10 years, virtualization techniques have become much more widely popular as a result of fast and cheap processors. Virtualization provides many benefits, which makes it an appealing test environment for high-performance computing. Encapsulating configurations is a huge motivating factor for wanting to do performance testing on virtual machines. Virtual machines enable portability among heterogeneous systems while providing an identical configuration within the guest operating system.

One of the consequences of virtualization is that there is an additional hardware abstraction layer. This prevents PAPI from reading the hardware Performance Monitoring Unit (PMU) directly. However, both VMware and KVM have recently, within the last year, added support for a virtual PMU. Within the guest operating system that provides a virtual PMU, PAPI can be built without any special build procedure and will access the virtual PMU with the same system calls as a hardware PMU.

To verify that performance event counts measured using PAPI are accurate, a series of tests were run to compare counts on bare metal to counts on ESXi and KVM. A suite of real-world applications was chosen to attempt to more accurately represent use cases for PAPI. In general, most users will use PAPI preset events to measure particular performance events of interest. PAPI preset events are a collection of events that can be mapped to native events or can be derived from a few native events. Preset events are meant to provide a set of consistent events across all types of processors.

2. Testing Setup

2.1 Benchmark Suite

Tests performed are taken from the Mantevo Suite of miniapplications [4]. The purpose of the Mantevo suite is to provide miniapplications that mimic performance characteristics of real-world large-scale applications. The applications listed below were used for testing:

- CloverLeaf – Hydrodynamics algorithm using a two-dimensional Eulerian formulation
- CoMD – An extensible molecular dynamics proxy applications suite featuring the Lennard-Jones potential and the Embedded Atom Method potential
- HPCCG – Approximation for implicit finite element method
- MiniGHOST – Executes halo exchange pattern typical of explicit structured partial differential equations
- MiniXYCE- Simple linear circuit simulator

2.2 Testing Environment

Bare Metal

- 16-core Intel Xeon CPU @ 2.9GHz
- 64GB memory
- Ubuntu Server 12.04
- Linux kernel 3.6

KVM

- Qemu version 1.2.0
- Guest VM - Ubuntu Server 12.04
- Guest VM - Linux kernel 3.6
- Guest VM - 16GB RAM

VMware

- ESXi 5.1
- Guest VM - Ubuntu Server 12.04
- Guest VM - Linux kernel 3.6
- Guest VM - 16GB ram

2.3 Testing Procedure

For each platform (bare metal, KVM, VMware), each application was run while measuring one PAPI preset event. Source code for each application was modified to place measurements surrounding the main computational work of each application. Ten runs were performed for each event. Tests were run on a "quiet" system; no other users were logged on, and only minimal OS services were running at the same time. Tests were performed in succession with no reboots between runs.

3. Results

Results are presented as bar charts in Figures 1–5. Along the x-axis is a PAPI preset event. Each preset event maps to one or more native events. Along the y-axis is the difference in event counts from bare metal event counts—that is, the ratio of the mean of 10 runs on the virtual machine to the mean of 10 runs on a bare metal system. Therefore, a value of 0 corresponds to an identical number of event counts between the virtual machine and bare metal. A value of 1 corresponds to a 100% difference in the counts on the virtual machine from on bare metal.

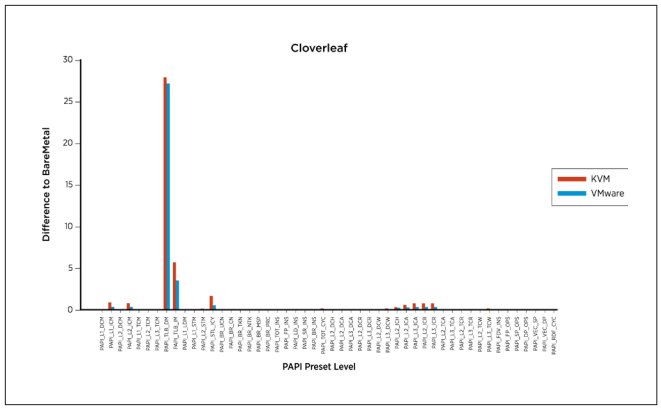


Figure 1. Event Counts on ESXi and KVM Compared to Bare Metal for the sCloverLeaf Benchmark

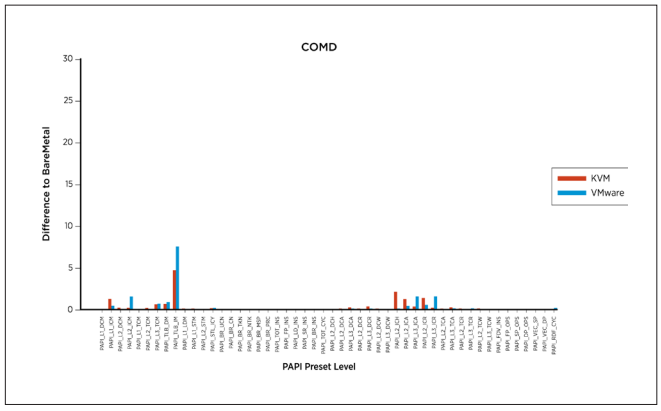


Figure 2. Event Counts on VMware and KVM compared to Bare Metal for the CoMD benchmark.

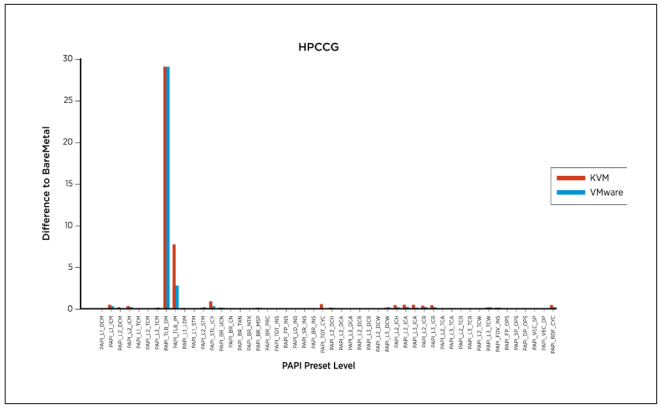


Figure 3. Event Counts on VMware and KVM compared to Bare Metal for the HPCCG benchmark.

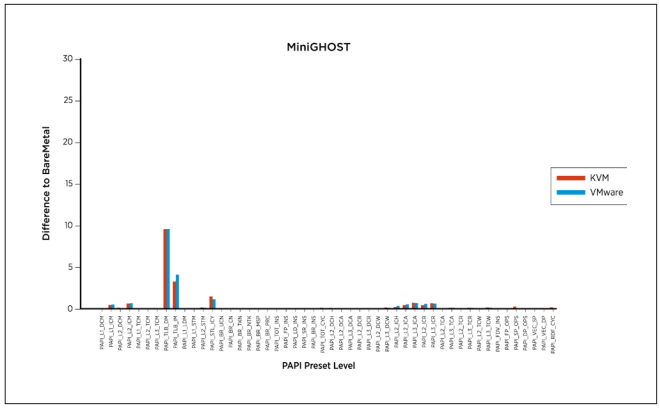


Figure 4. Event Counts on VMware and KVM compared to Bare Metal for the MiniGhost benchmark.

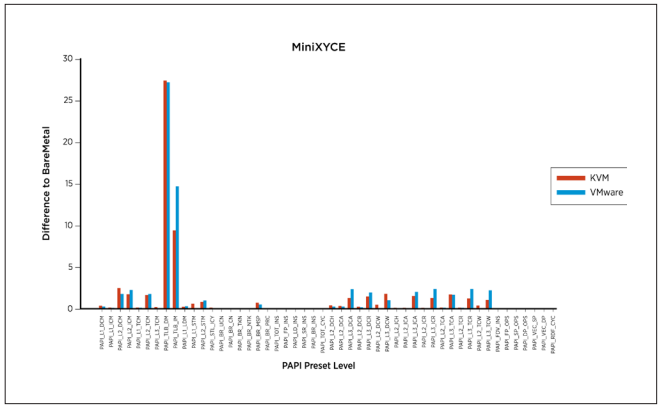


Figure 5. Event Counts on VMware and KVM compared to Bare Metal for the MiniXyce benchmark.

4. Discussion

On inspection of the results, two main classes of events exhibit significant differences between performance on virtual machines and performance on bare metal. These two classes include instruction cache events such as PAPI_L1_ICM, and translation lookaside buffer (TLB) events such as PAPI_TLB_IM. These two classes will be examined more closely below. Another anomaly that has yet to be explained is KVM reporting nonzero counts of PAPI_VEC_DP and PAPI_VEC_SP (both related to vector operations), whereas both the bare metal tests and the VMware tests report 0 for each application tested.

4.1 Instruction Cache

From the results, we can see that instruction cache event counts are much more frequent on the virtual machines than on bare metal. These events include: PAPI_L1_ICM, PAPI_L2_ICM, PAPI_L2_ICA, PAPI_L3_ICA, PAPI_L2_ICR, and PAPI_L3_ICR. By simple deduction, we can see that L2 events are directly related to PAPI_L1_ICM, and likewise L3 events to PAPI_L2_ICM. That is, the level 2 cache will only be accessed in the event of a level 1 cache miss. Miss rate will compound total accesses at each level and, as a result, the L3 instruction cache events appear the most different from on bare metal with a ratio of more than 2. Therefore, it is most pertinent to examine the PAPI_L1_ICM results, because all other instruction cache events are directly related. In Figure 6, we can see the results of the HPCCG tests on bare metal, KVM, and ESXi side by side. As can be seen on the graph, both KVM and ESXi underperform bare metal results. However, ESXi is quite a bit better off with only 20% more misses than bare metal, whereas KVM exhibits nearly 40% more misses. Both have a larger standard deviation than bare metal, but not by a huge margin.

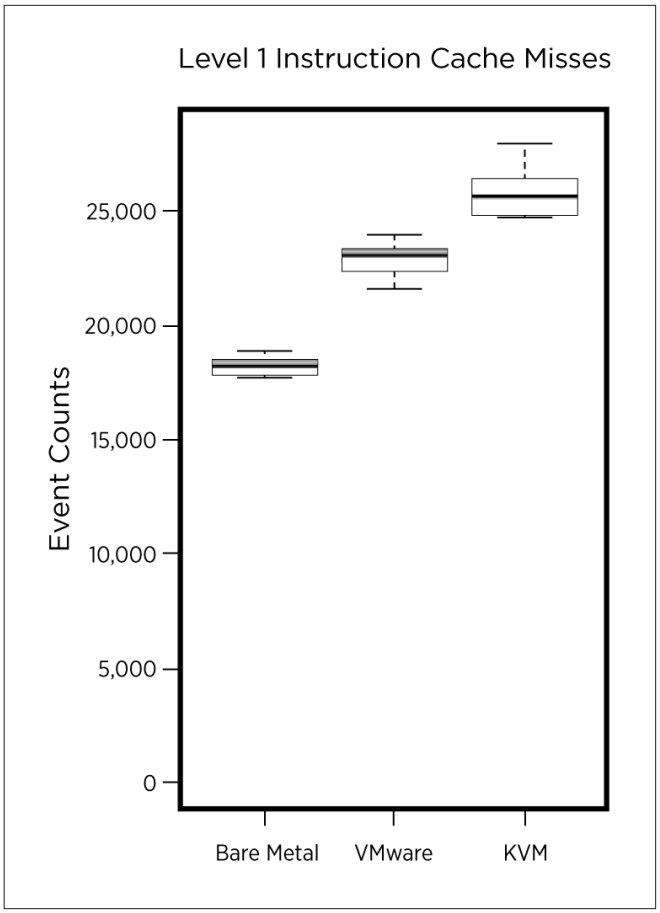


Figure 6. L1 Instruction Cache Miss Counts on ESXi and KVM Compared to Bare Metal for the HPCCG Benchmark

4.2 TLB

Data TLB misses are a huge issue for both KVM and ESXi. Both exhibit around 33 times more misses than runs on bare metal. There is little difference between the two virtualization platforms for this event.

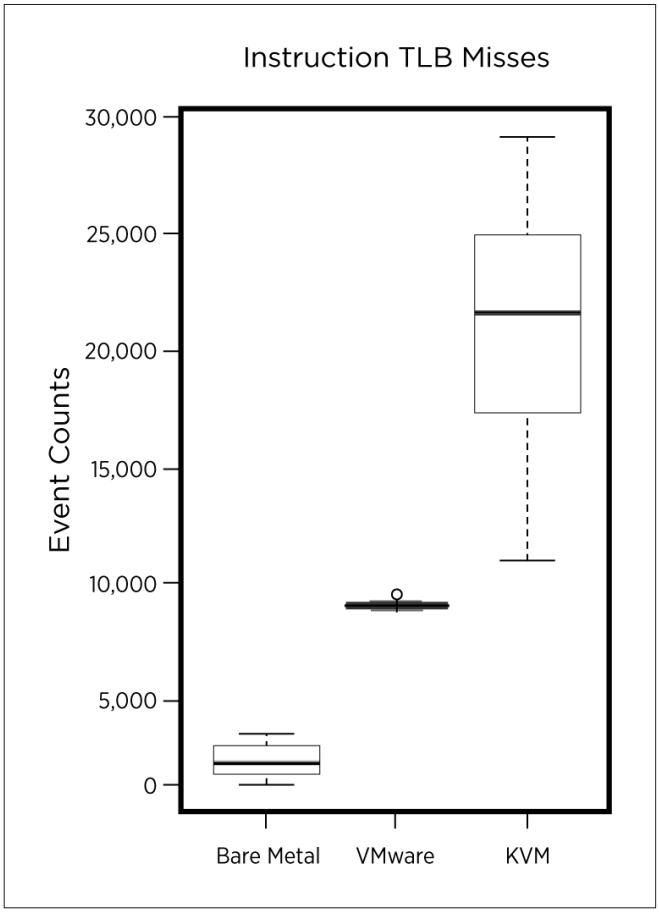


Figure 7. L1 Instruction TLB Miss counts on VMware and KVM compared to Bare Metal for the HPCCG benchmark.

Instruction TLB misses, shown in Figure 7, are also significantly more frequent on both virtualization platforms than on bare metal. However, ESXi seems to perform much better in this regard. Not only does ESXi incur 50% of the misses seen on KVM, but ESXi also has a much smaller standard deviation (even smaller than that of bare metal) compared to KVM's unpredictable results.

4.3 MiniXyce

The results for MiniXyce warrant special consideration. The behavior of the application itself appears much less deterministic than other tests in the test suite. MiniXyce is a simple linear circuit simulator that attempts to emulate the vital computation and communication of XYCE, a circuit simulator designed to solve extremely large circuit problems on high-performance computers for weapons design [4].

Even so, the behavior of the application may have exposed shortcomings in the virtualization platforms. Not only are instruction cache miss and TLB miss events much more frequent on ESXi and KVM, but data cache misses and branch mispredictions are also much more frequent. MiniXyce is the only application tested that displayed significantly different data cache behavior on the virtual machines from on bare metal. This may suggest that there exists a certain class of applications, with similar behavior characteristics, that may cause ESXi and KVM to perform worse than bare metal with regard to the data cache. This may be explained by the fact that virtualizing the Memory Management Unit (MMU) has a well-known overhead [2].

4.4 Possible Confounding Variable

VMware and KVM do not measure guest-level performance events in the same way. One of the challenges of counting events in a guest virtual machine is determining which events to attribute to the guest. This is particularly problematic when the hypervisor emulates privileged instructions. The two main counting techniques are referred to as *domain switch* and *CPU switch*. Domain switch is the more inclusive of the two, including all events that the hypervisor contributes when emulating guest I/O. VMware uses the term *host mode* for domain switch and *guest mode* for CPU switch.

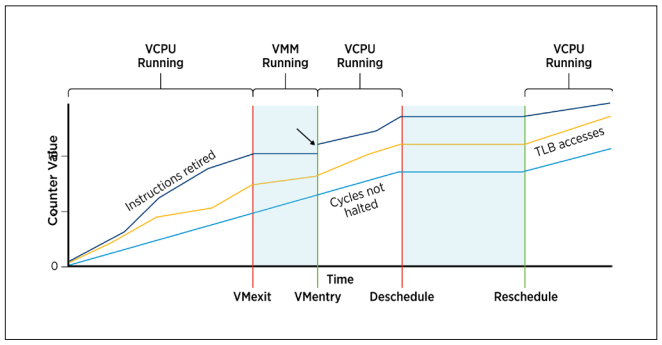


Figure 8. Example Timeline of Virtual Machine Scheduling [1]

Figure 8 shows an example timeline for virtual-machine scheduling events taken from [1]. CPU switch does not include events contributed by the hypervisor. On VM exit, when the hypervisor must process an interrupt (such as a trapped instruction), the performance counter state is saved and then restored on VM entry. Descheduling also causes the save and subsequent restore in CPU switch. Alternatively, the performance counter states in domain switch are saved only when the guest is descheduled and restored on rescheduling. Events are counted between VM exit and VM entry in domain switching. Domain switching, therefore, gives a more accurate view of the effects the hypervisor has on execution, whereas CPU switching hides this effect. However, domain switching may also count events that occur due to another virtual machine.

KVM chooses either domain switching or CPU switching, both of which have downsides as discussed above. ESXi, by default, uses a hybrid approach. Events are separated into two groups: speculative and nonspeculative. Speculative events include events that are affected by run-to-run variation. For example, cache and branch miss-prediction are both nondeterministic and may be related to previous executing code that affects the branch predictor or data cache. Both are affected by the virtual machine monitor as well as all virtual machines that are executed. Therefore, any specific virtual machine cache performance, or other nonspeculative event performance, will be affected by any other virtual machine that is scheduled. That is, the data cache will be filled with data by a virtual machine while it is executing. Further loads to this data will be cached. However, when execution returns to the virtual machine monitor or to another virtual machine, data in the cache may be overwritten by data that is loaded by the virtual machine monitor or another running virtual machine. On the other hand, nonspeculative events are events that are consistent, such as total instructions retired. ESXi will count speculative events between VM exit and VM entry but will not count nonspeculative events. This gives a more accurate representation of the effects of the hypervisor on speculative events while providing accurate results for nonspeculative events that should not be affected by the hypervisor [1].

The difference in how KVM and VMware choose to measure events could potentially skew the results for certain events. It is safe to assume that for the majority of the events tested, the event counts can be trusted, as there is a less than 1% difference between either virtualization platform and bare metal. It has also been shown in [2] that instruction cache and TLB performance overhead is a known problem, so it is likely that PAPI is in fact reporting accurate counts for related events.

5. Related Work

PAPI relies on PMU virtualization at the guest level to provide performance measurements. [3] discusses guest-wide and system-wide profiling implementations for KVM as well as domain versus CPU switch. [1] expands on the domain-versus-CPU-switch distinction by providing an alternative hybrid approach that counts certain events similar to domain switch and other events similar to CPU switch.

A performance study that examined VMware vSphere® performance using virtualized performance counters as one of the data-collection tools is presented in [1]. [1] also provides a discussion of TLB overhead, a result also observed by the study presented in this paper. [5] provides a framework for virtualizing performance counters in Xen, another popular virtualization platform.

6. Conclusions

PAPI can be used within a KVM or VMware virtual machine to reliably measure guest-wide performance events. However, there are a few events that, when measured, either reveal poor performance of the virtualization platform or are possibly overestimated when attributed to virtual machines. Virtual machine performance for the few significantly different events was expected [2]. For the large majority of events, one should expect to see almost identical results on a virtual machine as on a bare metal machine on a lightly loaded system. Future work examining event counts when a system is overloaded by many concurrently running virtual machines is necessary to make conclusions about whether results provided by PAPI are accurate in such a situation.

Acknowledgments

This material is based on work supported by the National Science Foundation under Grant No. CCF-1117058 and by FutureGrid under Grant No. OCI-0910812. Additional support for this work was provided through a sponsored Academic Research Award from VMware, Inc.

References

1. Serebrin, B. and Hecht, D. 2011. Virtualizing performance counters. In *Proceedings of the 2011 International Conference on Parallel Processing* (Euro-Par'11), Michael Alexander et al. (Eds.). Springer-Verlag, Berlin, Heidelberg, 223–233. http://dx.doi.org/10.1007/978-3-642-29737-3_26
2. Buell, J., Hecht, D., Heo, J., Saladi, K., and Taheri, H.R.. Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications. *VMware Technical Journal*, Summer 2013. http://labs.vmware.com/vmtj/methodology-for-performance-analysis-of-vmware-vsphere-under-tier-1-applications
3. Du, J., Sehrawat, N., and Zwaenepoel, W. Performance Profiling in a Virtualized Environment. 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2010), Boston, MA. https://www.usenix.org/legacy/events/hotcloud10/tech/full_papers/Du.pdf
4. Mantevo Project. http://www.mantevo.org
5. Nikolaev, R. and Back, G. Perfctr-Xen: A Framework for Performance Counter Virtualization. Virtual Execution Environments 2011, Newport Beach, CA. http://www.cse.iitb.ac.in/~puru/courses/spring12/cs695/downloads/perfctr.pdf
6. PAPI. http://icl.cs.utk.edu/papi/

Hierarchical Memory Resource Groups in the ESX Server

Ishan Banerjee
VMware Inc.
ishan@vmware.com

Jui-Hao Chiang
VMware Inc.
jchiang@vmware.com

Kiran Tati
VMware Inc.
ktati@vmware.com

Abstract

Modern operating systems specialize in partitioning the physical compute resources of a computer among software applications. Effective partitioning of physical resources enables multiple applications to securely execute on the same physical machine while maintaining performance isolation. In a virtualized environment, a hypervisor partitions physical resources among virtual machines. This enables virtual machines to securely execute on the same machine without affecting one another.

VMware's ESX Server is a hypervisor that provides controls for partitioning memory and CPU resources among virtual machines. ESX implements a hierarchical partitioning of memory and CPU resources using *resource groups*[1]. Resources are hierarchically partitioned based on their placement in a *tree* structure. Resource attributes such as *reservation*, *limit*, and *shares* provide users with fine-grained partitioning controls.

Hierarchical memory resource groups are a powerful tool enabling the partitioning of the physical memory resources of a computer. This enables fine-grained partitioning of compute memory among virtual machines in the data center. Partitioning can be deployed by a user or by automated data center management software.

This article describes the memory partitioning scheme of ESX, provides examples to demonstrate its use, and empirically evaluates its effectiveness.

General terms: memory management, memory partition, memory resource groups

Keywords: ESX, memory resource management

1. Introduction

The operating system (OS) traditionally controls the amount of resources that application software consumes on a single compute node. Modern data centers are equipped with an unprecedented amount of compute resources, such as memory and CPU. Virtualization of compute resources enables partitioning of the resources available on a single compute node in the data center. Effective utilization of the compute resources in a fair and efficient manner is an important task for the virtualization software. The hypervisor [2, 5] is software that virtualizes the physical compute resources from a single compute node. A hypervisor equipped with a flexible resource-partitioning scheme will enable efficient utilization of the compute node.

Consider an example in which a virtualized compute node has 64GB of physical memory (pRAM). This memory is partitioned into two fixed memory partitions: **A** with 32GB of pRAM and **B** with 32GB of pRAM. Consider two virtual machines (VMs)— V1 and V2—with virtual memory (vRAM) size of 24GB each. When V1 is powered on under **A**, it receives 24GB of memory. If V2 is powered on under **B**, it also receives 24GB of memory. However, an attempt to power on V2 under **A** will fail, because the **A** partition has only 32GB of total memory. This simple example highlights how the physical memory resource of a compute node can be partitioned by the hypervisor.

ESX is a hypervisor that provides fine-grained resource controls enabling users to partition physical memory and CPU resources among powered-on VMs [20]. This is done using (a) per-VM resource controls and (b) host-wide partitioning of resources. ESX implements dynamic partitioning of resources at both the per-VM and host-wide levels. A dynamic partitioning scheme allows resources to flow between partitions while the hardware is powered on.

Per-VM resource controls provide *reservation*, *limit*, and *shares* (RLS) attributes for controlling memory and CPU resources made available to a VM. Host-wide partitioning is implemented using *hierarchical resource groups*. This is a software-based dynamic partitioning scheme whereby resources can *flow* from one partition to another, if permitted, based on the RLS attributes of each partition.

This article describes hierarchical resource groups for partitioning physical memory resources, in ESX. ESX also provides similar capabilities for partitioning physical CPU resources. A description of CPU resource-partitioning capabilities is not included in this article. The remainder of this article is organized as follows:

- Section 2 provides information about memory-partitioning schemes in contemporary OSs and hypervisors.
- Section 3 describes hierarchical memory resource groups in ESX.
- Section 4 shows empirical results to demonstrate the effective use of memory resource groups.
- Section 5 concludes the article.

1 Resource **groups** are also known as resource pools in VMware's vSphere products.

2. Related Work

There has been considerable academic interest in developing resource-scheduling techniques for computing resources, for both traditional OSs and contemporary hypervisors, over the past two decades. An important goal was to develop controls for quality of service and performance isolation between resource consumers.

Priority schedulers in OSs assign absolute priorities to resource consumers, which can often be coarse-grained and ad-hoc [7]. Also, the priority assigned to one consumer could affect the resources scheduled for another consumer. *Fair schedulers* [13, 14] were found to be useful for coarse-grained controls but require complex usage collection and fine-tuning. To address these shortcomings, Waldspurger et al. [21] developed Lottery Scheduling, a proportional-share resource scheduler, to provide responsive control over resource scheduling. Stoica et al. [18] demonstrated how a proportional-share resource scheduler could work with realtime and non-realtime requirements.

Hierarchical resource scheduling has also received attention. Goyal et al. [9] show a hierarchical CPU scheduler for the Solaris kernel. It demonstrates how CPU resources can be partitioned, in software, among different application classes. Each application class can subdivide its allocation to its own subclasses.

Researchers have demonstrated *reservation* of compute resources and *limit* controls as a means of implementing quality of service. Bruno et al. [4] use Reservation Domains to guarantee CPU, I/O, network, and physical memory resources to processes. Performance isolation in the IRIX OS has been shown by Verghese et al. [19]. In this work the authors use a software abstraction called Software Performance Unit (SPU) to associate computing resources, such as memory and I/O, with CPUs. The SPU implements the unit of isolation. Researchers have demonstrated performance isolation in virtualization technologies such as Xen[2] and KVM[3]. Gupta et al. [11] show improved VM performance isolation with CPU resource limit controls in Xen Domains using the SEDF-DC and ShareGuard mechanisms.

The Linux[4] kernel and commercial OSs such as Microsoft[5] Windows implement simple priority-based scheduling for CPUs [3]. The Linux out-of-memory (OOM) killer contains a hint of memory partitioning because it terminates processes that consume excessive physical memory. FreeBSD[6] implements OS-level partitioning of execution environments, called *Jail*. This permits applications to execute within a Jail without being able to access data belonging to processes in another Jail. Jails, however, do not provide CPU and memory resource partitioning and isolation between application processes.

Hardware-based resource partitions have been implemented by IBM and HP. IBM's *LPAR* (Logical Partitioning) and HP's *nPar* (Hard Partitioning) use hardware techniques to electrically partition compute resource within a single server [10, 15]. An OS can execute on one hardware partition with exclusive CPU and memory resources assigned to the partition. However, because of physical characteristics within each physical server, there are limited ways to physically partition the hardware. Once a partition is configured, resources cannot be redistributed among partitions while the server is powered on.

Software-based resource partitions augment the hardware partitions. IBM's PowerVM, a virtualization solution, introduced the DLPAR (Dynamic Logical Partitioning) technology for dynamically configuring CPU and memory resources among LPARs on a server. HP's vPar (Virtual Partitions) statically partitions compute resources within a hardware partition (nPar).

Research on VM monitors and server consolidation using hypervisors complemented compute resource management. Bugnion et al. [5] developed Disco, which enables large-scale shared-memory multiprocessor machines to run unmodified commodity OSs. They employed inexpensive software-based virtualization on the IRIX OS to partition hardware resources. Govil et al. [8] demonstrated fault tolerance using Cellular Disco. Techniques for workload consolidation for hypervisors have also been developed. Memory page deduplication [12, 17] reduces the memory footprint of VMs. Live migration [6, 16, 22] of VMs balances workloads across multiple hypervisors.

Software-based resource partitions offer the flexibility of dynamically distributing hardware resources among partitions. ESX implements a software-based resource-partitioning scheme. It draws upon resource-partitioning concepts such as reservation, limits, and shares to provide fine-grained control over the distribution of resources among partitions. This is implemented using hierarchical resource groups.

3. Hierarchical Memory Resource Groups

This section describes the layout and configuration of hierarchical memory resource groups in ESX.

3.1 Resource Tree

Physical memory and CPU resources available to ESX are organized in the form of a tree. This is called the *resource tree*. A single resource tree is used by ESX for organizing both memory and CPU resources. This article discusses the organization and partitioning of memory resources based on the resource tree.

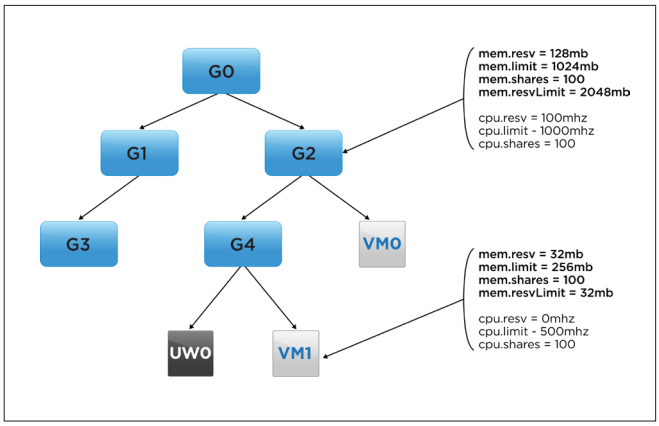


Figure 1. Memory resource groups form a tree-like structure in ESX. Vertices represent resource groups, and edges connect parent groups to their children. Five resource groups are shown in this figure: G0–G4. ESX considers VMs (VM0, VM1) and user-worlds (UW0) as resource groups with the same set of attributes. The square shape represents the special container resource group for VMs and user-worlds.

2 www.xenproject.org
3 www.linux-kvm.org
4 www.microsoft.com
5 www.linux.org
6 www.freebsd.org

Figure 1 illustrates the concept of a resource tree in ESX. The resource tree is a directed graph whose vertices are resource groups. A directed edge originates from a *parent* group and terminates at one of its *child* groups. In Figure 1, vertices G0–G4 are resource groups. ESX also considers VMs and user-worlds (UWs) as a resource group and creates special container resource groups as leafs in the resource tree. VMs and UWs are memory consumers, because they allocate and use physical memory. The resource groups themselves do not consume, store, or hoard memory.

A resource group has four associated memory attributes and four CPU attributes. The memory attributes are *mem.resv*, *mem.limit*, *mem.shares*, and *mem.resvLimit*. These attributes are configurable by the user. They do not change until explicitly altered by the user.

mem.resv indicates the amount of memory that is guaranteed to be distributed to memory consumers placed under that resource group. Memory consumers are typically VMs and UWs. The guaranteed memory is not reserved up front for the consumers. However, ESX will make this amount of memory available by reclaiming an appropriate amount from other memory consumers when needed. For safe operation of ESX, ESX performs admission control on the mem.resv attribute when its value is altered on any resource group or when a new resource group is created.

mem.limit indicates the maximum memory that can be consumed by all memory consumers placed under a resource group. If memory consumers under a resource group attempt to consume more memory, then ESX will reclaim memory from memory consumers under that group. The amount of memory to reclaim from each memory consumer will be determined by ESX based on their RLS attributes and other consumption patterns of those memory consumers. For a resource group, *mem.limit* is always greater than or equal to its *mem.resv*.

mem.shares indicates a relative priority of memory distribution between a resource group and its siblings. Typically, if the total memory consumption below a resource group exceeds its *mem.limit*, then *mem.shares* of its child resource groups come into play. Memory is distributed to its child resource groups based on their *mem.shares*.

ESX can sometimes implicitly increase the value of *mem.resv* at a resource group. The *mem.resvLimit* attribute indicates the maximum value *mem.resv* can have at a given resource group. It acts as an upper bound during this implicit increase. These attributes are available for each resource group, including VMs and UWs.

Figure 2 shows an example of a resource tree instantiated in ESX. The *HOST* resource group is assigned all the physical memory resources available to ESX. For example, on a 64GB ESX server, the *HOST* resource group will have *mem.resv* = *mem.resvLimit* = *mem.limit* = *64GB*[7].

The HOST resource group has four children: *VIM*, *SYSTEM*, *USER*, and *IDLE*. VMware vSphere® products place all powered-on VMs under the *USER* resource group. vSphere does not place non-VM memory consumers under this resource group. The remaining three resource groups are used by ESX for executing UWs, for placing kernel modules, and for other bookkeeping purposes.

7 The actual value may be slightly lower owing to memory pages being set aside for booting ESX.

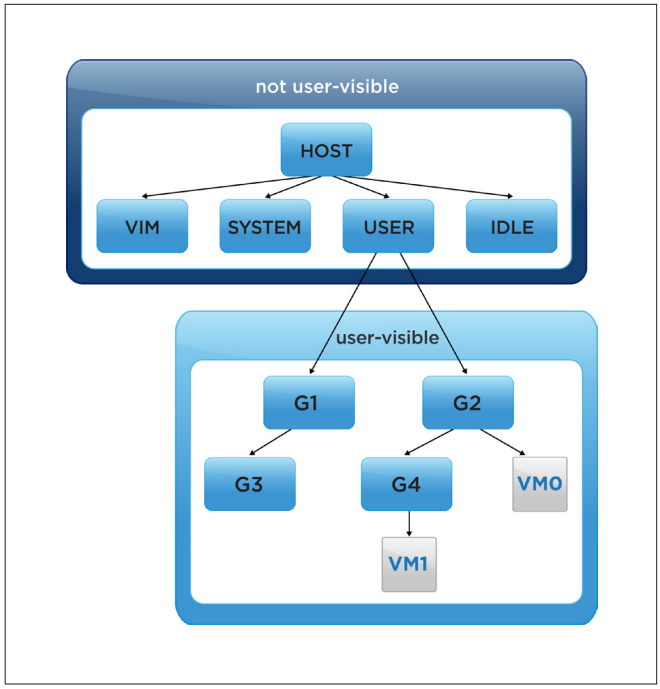


Figure 2. The resource hierarchy below the USER resource group is made visible to the user in vSphere products. The remaining resource groups are used for bookkeeping by ESX and are not visible to users in vSphere products.

These three resource groups can be configured automatically by ESX to contain memory resources to be made available to consumers within them.

For the purpose of simplicity in the remainder of this section, the resource tree model from Figure 1 will be used. In this model, the G0 resource group is equivalent to the HOST resource group in an ESX server. The following subsections describe operations on the resource tree.

3.2 Structural Operations

The resource tree allows structural operations—*add*, *move*, and *delete*—to be performed on it. Figure 3 (a) shows an example of a resource tree with two resource groups, G0 and G1. Figures 3 (b), 3 (c), and 3 (d) show changes being made relative to the immediately preceding figure. All structural operations performed on the resource tree are atomic in nature. That is, if an operation fails, the resource tree will retain the structure and properties that were in effect before the operation was initiated.

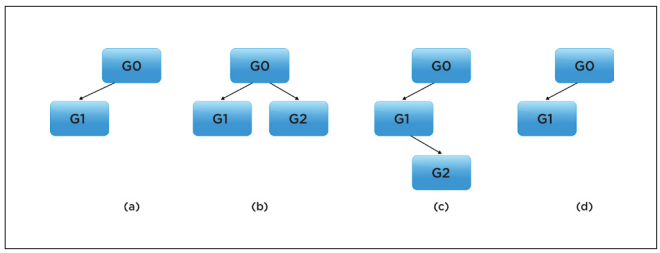


Figure 3. Structural operations on a resource tree: add, move, and delete. (a) Initial state of the resource tree. Each subsequent figure is relative to the immediately preceding figure. (b) G2 is added. (c) G2 is moved. (d) G2 is deleted.

In Figure 3 (b), an *add* operation adds a new resource group G2 to the resource tree as a child of G0. The new resource group will contain all the attributes described in Figure 1. For the operation to succeed, G2 will undergo *admission control* (see Section 3.1).

In Figure 3 (c), a *move* operation has moved resource group G2 to a new location as a child of group G1. The attributes of G2 remain unchanged with this operation. However, for the operation to succeed, G2 will undergo admission control as a child of G1. If admission control fails, then the operation is reverted.

In Figure 3 (d), a delete operation has deleted the resource group G2. Only leaf resource groups can be deleted. The delete operation on a leaf resource group does not fail.

3.3 Attribute Operation and Semantics

This section describes the four memory attributes of resource groups. It also describes how ESX interprets and uses these attributes for admission control and memory distribution among resource groups and memory consumers.

3.3.1 mem.resv

The *mem.resv* resource group attribute specifies the amount of memory that is guaranteed to memory consumers under that resource group. This memory will be made available to the memory consumers when required. It is not stored at the resource group.

Admission control is performed at a resource group G when (1) the value of *mem.resv* is changed at one of its children and (2) a new resource group is added as a child of G. The following condition must be true before the above two operations are declared to be successful.

$$\texttt{parent.mem.resv} >= \sum_{child \in children} \texttt{child.mem.resv} \quad (1)$$

In Equation 1, *children* includes the group being added to G. Admission control is performed when a new group is added to the resource tree as well as when a group is being moved within the tree. The admission control ensures that a parent group always has enough reservation to distribute to its children. The left-hand side of the equation is replaced with *effective mem.resv* when the parent has a configured finite *mem.resvLimit* (see Section 3.3.2).

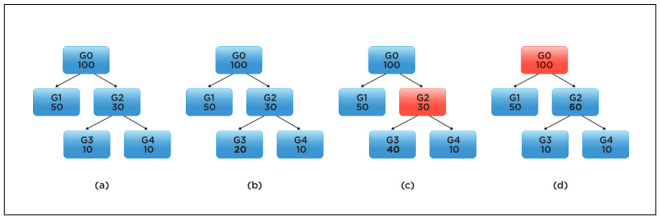


Figure 4. Operation on the **mem.resv** attribute. Resource groups are shown with their names and **mem.resv** values. Admission control with Equation 1 is always performed at each resource group when mem.resv is changed. (a) Initial state of the resource tree. Subsequent figures are relative to this figure. (b) G3 raised to 20 and succeeds. (c) Attempted increase of G3 to 40 fails owing to admission control at G2. (d) Attempted increase of G2 to 60 fails owingG1 admissionG2 control at G0.

Figure 4 shows admission control being performed when *mem.resv* is changed. For simplicity of explanation, it is assumed that *mem.resvLimit* = *mem.resv*. This assumption is not always true in ESX. Section 3.3.2 shows a realistic example in which both *mem.resv* and *mem.resvLimit* are considered during admission control. In Figure 4, resource groups are shown with their names and values of *mem.resv*.

Figure 4 (a) shows the initial state of the resource tree. In this state, Equation 1 is satisfied for every resource group. In Figure 4 (b), *mem.resv* for G3 is raised to 20. This change succeeds because Equation 1 is satisfied by G3's parent, G2. In Figure 4 (c), an attempt is made to raise G3 to 40. This change fails because Equation 1 fails at G2 (shaded). Similarly, Figure 4 (d) shows the failure to raise G2's *mem.resv* from 30 to 60.

3.3.2 mem.resvLimit

When an attempt is made to increase the *mem.resv* value of a resource group G, admission control might fail the operation owing to a failure of Equation 1 at a resource group. This will require users to appropriately configure the value of *mem.resv* at one or more ancestor resource groups of G.

ESX provides an *expandable reservation* scheme to automatically perform this increase at all required ancestor resource groups of G. It is implemented using the *mem.resvLimit* attribute. The *mem.resvLimit* attribute provides ESX with a safe method of automatically and implicitly increasing the value of *mem.resv* at a resource group to satisfy admission control. The *mem.resvLimit* provides an upper bound on how much ESX can implicitly raise the value of *mem.resv*.

Expandable reservation works by permitting ESX to implicitly raise the *mem.resv* of a resource group G up to *mem.resvLimit*. The implicitly computed value is called *effective mem.resv* of G. After doing so, ESX must use the effective mem.resv to perform admission control at G's parent using Equation 1. ESX can continue to move up the resource tree until it reaches the topmost resource group.

Figure 5 shows a resource tree to illustrate the use of *mem.resvLimit*. In Figure 5, each resource group is labeled with its name and values of *mem.resv* and *mem.resvLimit*, separated by /. Figure 5 (a) shows the initial state of the resource tree. In this state, all resource groups satisfy Equation 1. Figures 5 (b), 5 (c), 5 (d), 5 (e), and 5 (f) show changes being made relative to Figure 5 (a). In these figures, resource groups can have different *mem.resv* and *mem.resvLimit* values—unlike Figure 4, in which all resource groups are assumed to have *mem.resvLimit* = *mem.resv*.

In Figure 5 (b), an attempt is made to raise the mem.resv of resource group G3 to 30. Without *expandable reservation*, the admission control would have failed at group G2 using Equation 1, because an additional *mem.resv* of 10 is required at G2. However, *mem.resvLimit* at G2 permits its *mem.resv* to be increased by 10. ESX implicitly attempts to increase *mem.resv* of G2 by 10 to 40. This is permissible at G0 using Equation 1. ESX first internally computes an effective *mem.resv* for G2 as 40 and permits the *mem.resv* of G3 to be increased to 30.

Figure 5 (c) shows that it is possible to configure *mem.resvLimit* with any value. Admission control is not performed when the value of *mem.resvLimit* is altered. In Figure 5 (c), the *mem.resvLimit* of G1 and G2 are set to 80. It might or might not be possible to implicitly raise the *mem.resv* of either resource group in future. That will be known only when an attempt is made to increase their *mem.resv* or the *mem.resv* of one of their children.

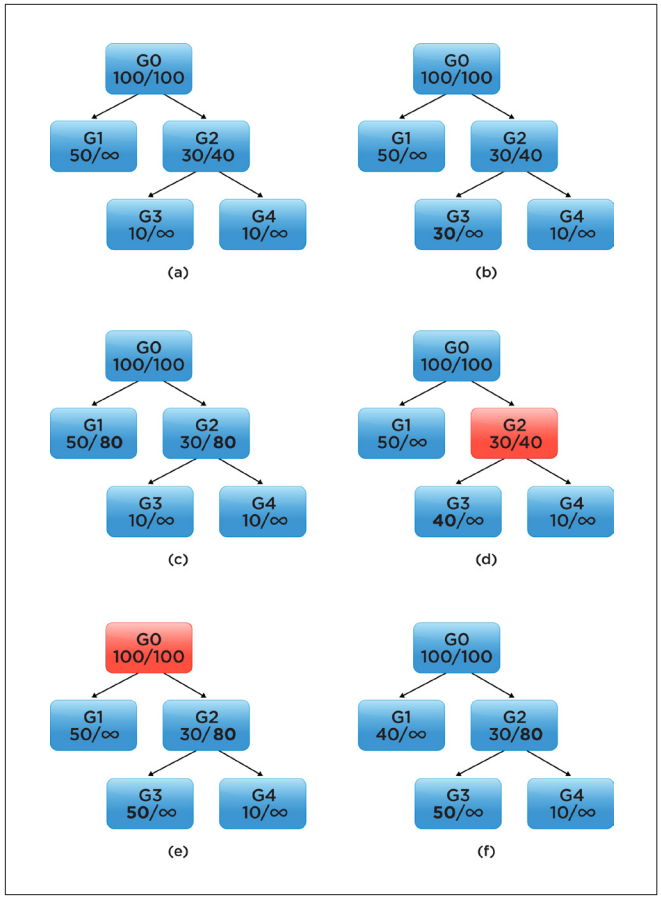


Figure 5. Operation on the **mem.resv** attribute in the presence of **mem.resvLimit**. Each resource group contains its name and (**mem.resv** / **mem.resvLimit**). (a) Initial state of resource tree. Subsequent figures are relative to this figure. (b) Change to G3 succeeds owing to sufficient **mem.resvLimit** at G2. (c) Change to **mem.resvLimit** always succeeds. (d) Change to G3 fails owing to insufficient **mem.resvLimit** at G2. (e) Change to G3 fails owing to insufficient **mem.resv** at G0. (f) Change to G3 succeeds. Why?

Figure 5 (d) shows an attempt to raise *mem.resv* of G3 to 40. As before, admission control using Equation 1 fails at G2, because an additional 20 *mem.resv* is required. However, *mem.resvLimit* of G2 permits its *mem.resv* to be implicitly raised by only 10. Hence, the implicit increase of mem.resv at G2 fails, and so does the admission control.

Figure 5 (e) shows an attempt to raise the *mem.resv* of G3 to 50 after raising the *mem.resvLimit* of G2 to 80. Equation 1 at G2 would require an additional 30 of *mem.resv* at G2 to succeed. This would implicitly raise the effective *mem.resv* of G2 to 60. However, doing so would cause Equation 1 to fail at G0. Owing to this failure at G0, admission control is deemed to have failed.

Figure 5 (f) shows an example in which G1 and G2 are altered, followed by G3. It is left to the reader to determine why the change to G3 succeeds.

These examples show how *mem.resv* and mem.resvLimit work together to reserve memory for resource groups.

3.3.3 mem.shares

When distributing memory to a resource group, ESX works in a top-down manner, starting from the root of the resource tree. ESX takes all the memory at a given resource group and distributes it among its children. ESX first distributes the *mem.resv* memory for each child. This will always succeed at every level, because the parent of a resource group always has enough memory to satisfy the *mem.resv* of all its children (Equation 1). Thereafter, ESX will distribute a parent's memory to its children based on the children's relative memory shares[8].

The relative share of a resource group determines the amount of memory that a resource group will receive from its parent's distribution amount, relative to its sibling resource groups. The *mem.shares* attribute enables the user to configure the relative memory shares of a resource group.

Because *mem.shares* is a relative quantity, its absolute value is not important. When a new resource group is added, the memory distribution among its siblings is automatically adjusted. A user can consider this quantity as a relative priority among sibling resource groups. Figure 6 illustrates the use of *mem.shares*. Figure 6 (a) is the initial state of the resource tree. In Figure 6 (a), the relative priority of resource groups G1 and G2 are indicated with 100 and 200 respectively. Mathematically, the memory distribution of their parent, G0, will be distributed between them in the ratio 1:2, or 1/3 and 2/3 using fractional representation.

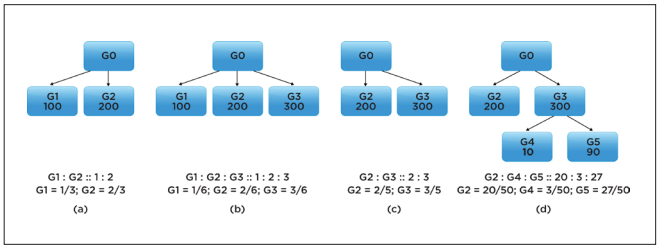


Figure 6. Illustration of the **mem.shares** attribute of resource groups. Resource groups are shown with their names and values of **mem.shares**. Ratio and equivalent fraction of relative shares among leaf-level resource groups are shown. (a) Initial state of resource tree. Subsequent figures are relative to each immediately preceding figure. (b) G3 is added. (c) G1 is removed. (d) G4 and G5 are added. Relative distribution is automatically adjusted in each case.

Figures 6 (b), 6 (c), and 6 (d) are each relative to the immediately preceding figure. In Figure 6 (b), a new resource group G3, with relative shares of 300, has been added to G0. As a result, the relative distributions of G1 and G2 are automatically adjusted relative to G3. Similarly, in Figure 6 (c), G1 has been removed. The relative distributions of the remaining groups are automatically adjusted according to their relative shares. In Figure 6 (d), G4 and G5 have been added to G3. The relative shares of the leaf-level resource groups are shown. This example illustrates that relative shares determine relative priority among siblings. Their absolute value is not relevant at other levels of the resource tree.

3.3.4 mem.limit

The memory limit on a resource group is given by *mem.limit*. This value determines the maximum amount of memory that can be distributed to a resource group. It has a lower bound of *mem.resv*. When distributing memory to a resource group based on its shares, ESX will stop when the total memory distributed to that resource group reaches its *mem.limit*. When value of *mem.limit* on a resource group is configured, there are no admission-control steps to be performed. A resource group can be given any value of *mem.limit* that is greater than or equal to its *mem.resv*.

8 ESX combines other usage factors such as **active**ness with relative shares

3.4 Memory Demand and Distribution

The previous sections described attributes of resource groups: *mem.resv*, *mem.resvLimit*, *mem.shares*, and *mem.limit*. These attributes are used by ESX to translate memory demands from memory consumers in resource groups into memory distribution to those consumers.

Configuration and use of the resource tree does not require knowledge of the configured memory size of a memory consumer. The configured memory size of a VM is the virtual address space of the VM, while for a UW it is the *mmaped* size. VMs and UWs, which are memory consumers, allocate and consume memory from the free memory pool in ESX. The amount of memory consumed by a memory consumer is its *demand*. From time to time and when free memory is low, ESX uses the memory resource tree and the memory usage characteristics, such as activeness, of the memory consumers to determine the appropriate distribution[9] of memory for each consumer. This is the step that translates a memory consumer's demand into its distribution. If a memory consumer is consuming more than its distribution, then the excess memory is reclaimed from it by using a memory reclamation technique [1] such as memory ballooning, memory compression, or hypervisor-level swapping.

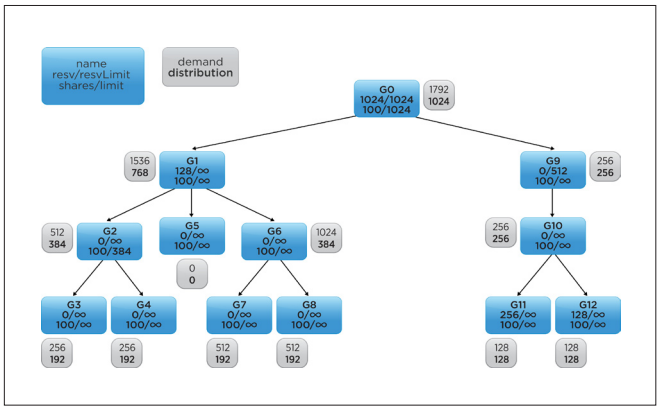


Figure 7. Illustration of memory demand and distribution using a memory resource tree. Each resource group shows its name, **mem.resv**, **mem.resvLimit**, **mem.shares**, and **mem.limit**. The memory demanded from memory consumers and the all resource group attributes are inputs to ESX. Based on these inputs, ESX distributes memory to resource groups, starting from G0. The total physical memory available for distribution at G0 is 1024.

Figure 7 shows an example in which all the memory resource attributes and memory demand from consumers are considered. ESX takes the configuration attributes and memory demand as the input and produces the memory distribution for each resource group as the output. For simplicity, it is assumed that the memory usage characteristics—such as activeness—of all memory consumers are identical.

Figure 7 assumes that memory demands are generated by memory consumers located in the leaf-level resource groups only. Other resource groups are assumed to have no memory consumers directly attached to them. The root resource group, G0, is assigned all the physical memory available to ESX: 1024 units. For this group, *mem.resv*, *resvLimit*, and *mem.limit* are all identical and equal to 1024.

9 Also known as **entitlement** and **allocation target** in vSphere

To compute the memory distribution for each resource group in the resource tree, ESX executes the following steps:

1. Traverse the tree in a bottom-up manner, calculating the total memory demand at each resource group. The total demand at each resource group is the sum of the demands from its children.
2. Traverse the tree in a top-down manner, distributing memory to each resource group based on its resource attributes, total demand, and usage characteristics.

In step 1, ESX computes the demand at each non-leaf resource group. This is simply the sum of their respective children. In step 2, ESX distributes memory to each resource group in a top-down manner.

This section presented hierarchical memory resource groups in ESX. It described attributes that control distribution of memory to resource groups. The next section uses controlled experiments to evaluate the effectiveness of memory resource groups in distributing memory.

4. Evaluation

This section evaluates the effectiveness of hierarchical memory resource groups in partitioning physical memory among VMs.

Specifically, the following are demonstrated:

- Functional evaluation – Three experiments are conducted using simple memory-consuming workloads as follows:
  - **Resv** – Show that *mem.resv* guarantees memory to memory consumers under a resource group.
  - **Shares** – Show that *mem.shares* distributes memory to memory consumers under a resource group in a fair manner.
  - **Limit** –Show that *mem.limit* bounds the maximum memory distributed to memory consumers under a resource group.
- Benchmarks – A complex hierarchical resource tree is used to demonstrate the effectiveness of the *mem.resv*, *mem.shares*, and *mem.limit* attributes.

4.1 Functional Evaluation

The *mem.resv*, *mem.shares*, and *mem.limit* attributes are evaluated using three independent experiments. The evaluation shows that these attributes provide a method to enforce guaranteed memory, fair distribution, and an upper bound of memory distribution to memory consumers.

Experiments are conducted on a 16-core, hyperthreaded, 128GB RAM, 256GB SSD ESX server with a development build of ESX 6.0. All VMs contain an Ubuntu 64-bit OS. VMs are stored on local 900GB, 10K hard drives. VMs in these experiments execute a memory-intensive workload. This workload allocates a specified amount of memory and writes a random pattern into it. It then accesses all the allocated memory in a round-robin manner for a specified time. In these experiments, memory ballooning, transparent page sharing, and memory compression are disabled. The only method of memory reclamation is hypervisor-level memory swapping. In addition, memory distribution to VMs by ESX is based only on the VMs' demand. Other factors, such as activeness of the workload, are disabled[10].

10 Advanced memory configuration option **IdleTax** is set to 0

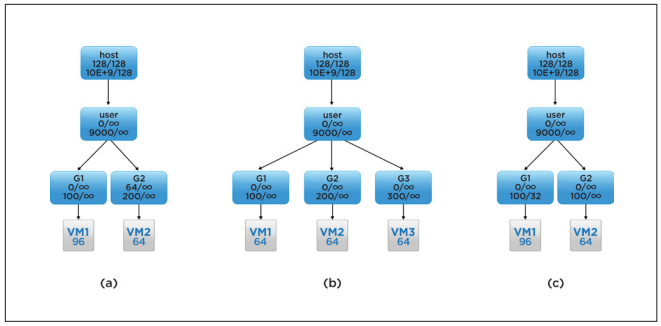


Figure 8. Functional experiments demonstrating the use of (a) **mem.resv** (b) **mem.shares** (c) **mem.limit**

Figure 8 shows the resource trees for the three experiments. The experiments are described below.

4.1.1 mem.resv

Figure 8 (a) shows the resource tree for evaluating the behavior of *mem.resv*. Resource groups are shown with the same format as in Figure 7. In Figure 8, resource groups G1 and G2 are children of *user*. G2 is configured with a *mem.resv* of 64GB. VM1 has a configured virtual memory size of 96GB, while VM2 has 64GB. Based on this resource tree configuration, memory consumers placed under G2 are guaranteed 64GB of memory, irrespective of memory demands from other (namely, G1) resource groups.

In the experiment, VM1 and VM2 are powered on and the memory-intensive workloads, described above, of size 90GB and 60GB respectively, are executed for 1,200 seconds. The total memory consumption, including guest OS components inside the VMs, is about 94GB and 64GB respectively. Figure 9 shows the memory consumption of VM1 and VM2. In Figure 9, the X-axis shows time in seconds, and the Y-axis shows the consumed and swapped memory. It can be seen that VM2 is able to allocate and consume 64GB memory throughout the experiment. VM1, on the other hand, is able to consume the remaining memory. Although VM1 attempts to consume as much as 94GB of memory, it is distributed about 60GB of memory. The remaining demand is met by reclaiming about 30GB of VM1's virtual memory to the swap space. This experiment demonstrates that the *mem.resv* attribute of resource group G2 is able to guarantee memory distribution to its memory consumers, namely VM2.

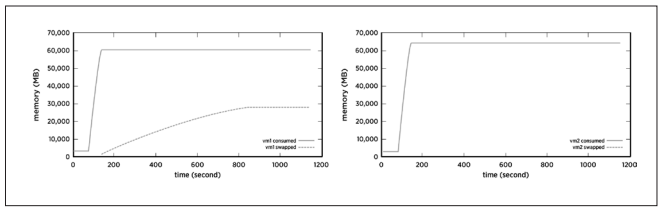


Figure 9. Evaluation of the behavior of **mem.resv**. VM2 is guaranteed 64GB of memory by G2.

4.1.2 mem.shares

Figure 8 (b) shows the resource tree for evaluating the behavior of *mem.shares*. In Figure 8 (b), resource groups G1, G2, and G3 are children of *user*. Their *mem.shares* are 100, 200, and 300 respectively. This means that ESX will attempt to distribute memory to them in the ratio 1:2:3. Each of the three VMs has a configured memory size of 64GB.

In the experiment, VM1, VM2, and VM3 are powered on, and the memory-intensive workload, described above, of size 64GB is executed in each VM. Figure 10 shows the memory consumed by each of the VMs. In Figure 10, the X-axis shows the time in seconds, and the Y-axis shows consumed and swapped memory. It can be seen that the memory consumption of the VMs at steady state is about 21GB, 42GB, and 63GB respectively. About 43GB, 22GB, and 1GB are reclaimed from VM1, VM2, and VM3 respectively. This experiment demonstrates that the *mem.shares* attribute of resource groups is able to distribute memory based on the configured relative shares of the VMs.

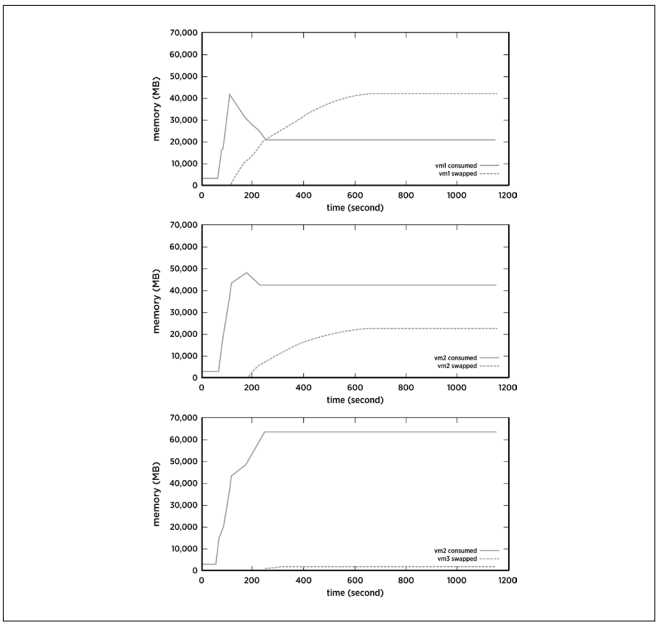


Figure 10. Evaluation of the behavior of mem.shares. VM1, VM2, and VM3 consume memory in the ratio 1:2:3.

4.1.3 mem.limit

Figure 8 (c) shows the resource tree for evaluating the behavior of *mem.limit*. In Figure 8 (c), resource groups G1 and G2 are children of user. G1 is configured with a *mem.limit* of 32GB. VM1 and VM2 are configured with 96GB and 64GB of virtual memory. Based on this resource tree configuration, memory consumers under resource group G1 will be distributed at most 32GB of memory.

In this experiment, VM1 and VM2 are powered on, and the memory-intensive workloads, described above, of size 90GB and 60GB respectively, are executed for 1,200 seconds. The total memory consumption, including guest OS components, inside the VMs total about 94GB and 64GB. Figure 11 (c) shows the memory consumption of VM1 and VM2. In Figure 11, the X-axis shows time in seconds, and the Y-axis shows the consumed and swapped memory. It can be seen that VM1's distribution is limited to 32GB,

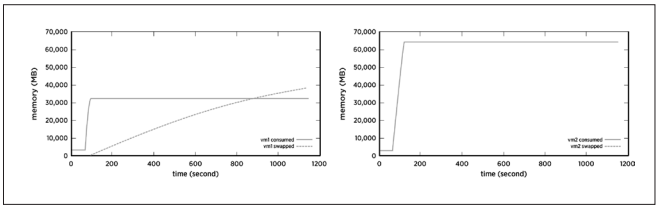


Figure 11. Evaluation of the behavior of mem.limit. VM1 is limited to 32GB by G1.

although it attempts to allocate 94GB. (The experiment is terminated before 96GB is allocated.) The memory consumption in excess of 32GB is reclaimed using hypervisor-level swapping. VM2 is able to consume 64GB of the remaining memory. This experiment demonstrates that the *mem.limit* attribute of resource group G1 is able to limit the memory distribution to its memory consumers, namely VM1.

These three simple experiments demonstrated how memory resource attributes— *mem.resv*, *mem.shares*, *mem.limit*—can be effectively used to control distribution of physical memory to VMs. The next section describes an experiment with a complex resource tree.

4.2 Benchmarks

In this section, an experiment is conducted to demonstrate complex partitioning of physical memory among VMs, using a larger memory resource tree and the *mem.resv*, *mem.limit*, and *mem.shares* attributes.

The experiment was conducted using an ESX server with 64GB of physical memory, 16 cores with hyperthreads spread across 4 NUMA nodes, 200GB SSD (of which 64GB was used as a swap space for VMs), and a 900GB 10K local disk. A development build of ESX 6.0 was used as the hypervisor.

Two different workloads were used in this experiment.

- **SPECjbb05** – A VM with configured memory size of 4GB running 64-bit Ubuntu OS. SPECjbb05[11] with 5GB heap, 8 warehouses and 1,800 seconds of execution time was executed continuously. The steady-state memory demand from this workload VM was observed to be 2,500MB when independently executed with ample memory resources.
- **Kernel compile** – A VM with configured memory size of 4GB running 64-bit CentOS 6. Kernel compile workload was executed continuously. The steady-state memory demand from this workload VM was observed to be 3,080MB when independently executed with ample memory resources.

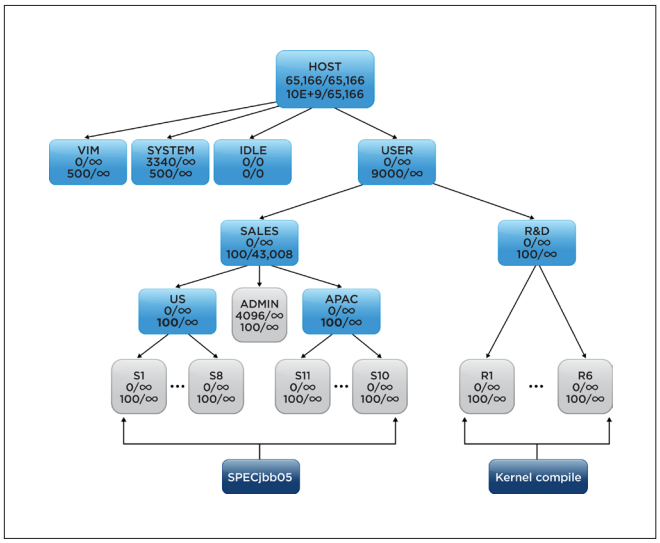


Figure 12. Memory resource tree for demonstrating **mem.resv**, **mem.limit**, and **mem.shares** attributes. The **Sales** resource group has **mem.limit** = 43,008MB. The **Admin** VM has **mem.resv** = 4GB. Relative shares of the **US** and **APAC** resource groups are varied in successive experiments.

11 http://www.spec.org/jbb2005/

The resource tree shown in Figure 12 was instantiated on the ESX server. The *user* resource group had two children: *Sales* and *R&D*. Sales had two children: *US* and *APAC*. The *R&D* resource group contained 6 kernel compile VMs. The *US* and *APAC* resource groups contained 8 and 10 SPECjbb05 VMs respectively. In addition, the *Sales* resource group had a *mem.limit* of 43,008MB. A single SPECjbb05 VM with *mem.resv* = 4GB was placed under *Sales*. All other resource-group parameters were set to the default values. The *idle*, *system*, and *vim* resource groups are shown for completeness.

In Figure 12, four unique categories of VMs are present:

- VMs under the *US* resource group
- VMs under the *APAC* resource group
- The single VM placed directly under the *Sales* resource group
- VMs under the *R&D* resource group

All VMs in each of the above four categories will behave in a similar manner.

The following three hypotheses are made about this resource tree. The results of the experiments will be used to validate them.

H1: The *mem.limit* attribute will limit distribution of memory to the *Sales* resource group.

H2: The *mem.shares* attribute will proportionately distribute memory between sibling resource groups *US* and *APAC*.

H3: The *mem.resv* attribute will guarantee memory distribution to the *Admin* VM.

To simplify the experiments and reduce randomness of execution, memory reclamation using transparent page sharing, memory compression, and memory ballooning were disabled. Also, memory distribution is based solely on demand. Other factors such as activeness are not considered during memory distribution by ESX.

4.2.1 Equal Resource Attributes

In this experiment, ESX was instantiated with the resource tree shown in Figure 12. The relative shares of the *US* and *APAC* resource groups were set to 100 each. VMs were powered on, as shown in Figure 12. The workloads in the VMs were allowed to execute continuously for 12,000 seconds. The VMs were then powered off. The memory distributed by ESX to each resource group and to each VM was recorded at intervals of 1 second.

The total steady-state memory demand under each resource group, calculated from the steady-state demand of each VM, is shown in Table 1 (a) (*total demand*). This is the memory demand generated by the workload and the OS inside the respective VMs.

During execution, ESX distributes memory to resource groups and VMs based on their memory demands and the resource group attributes. Because the *Sales* resource group has a *mem.limit* of 43,08MB, ESX will distribute a maximum of this amount of memory to *Sales*. The remaining memory will be distributed to the *R&D* resource group. The *Sales* resource group will first meet the memory demand of *Admin* and then distribute the rest in equal proportion to the *US* and *APAC* resource groups.

Figure 13 (a) shows the memory distributed by ESX to each resource group. Figure 13 (b) shows the memory distributed by ESX to the *S1, S11, Admin*, and *R1* VMs during the experiment. In Figure 13, the X-axis shows time in seconds, and the Y-axis shows the amount of memory distribution in MB. From this figure, it can be seen that the workloads take about 900 seconds to stabilize. The memory distribution at steady state (chosen as the 6,000 second mark) from Figure 13 (a) is added to Table 1 (a) (*distribution*) for comparing with the memory demands at the respective resource groups.

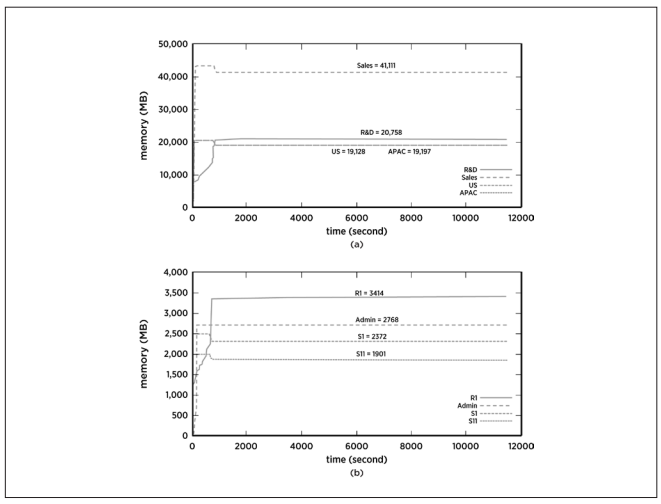


Figure 13. Memory distribution by ESX to resource groups and VMs, based on demand and resource group attributes. Values at steady state (6,000 second mark) are labeled. (a) Distribution to the **Sales**, **US**, **APAC**, and **R&D** resource groups. (b) Distribution to the **S1**, **S11**, **Admin**, and **R1 VMs**.

From Figure 13 (a) it can be seen that hypotheses **H1** and **H2** hold true:

H1 Distribution to *Sales* does not exceed 43,008MB at any time. This shows that the *mem.limit* resource group attribute can be effectively used to limit the total memory distribution to a resource group.

H2 Distributions to *US* and *APAC* always remain in equal proportion. This shows that the *mem.shares* resource group attribute can be effectively used to proportionately distribute memory among resource groups.

From Figure 13 (b), hypothesis **H3** holds true:

H3 Memory demand of 2,500MB by *Admin* is always met with a distribution of 2,768MB. This shows that the *mem.resv* resource group attribute can be effectively used to guarantee memory distribution to a resource group.

From Table 1 (a), it can be seen that

- Distributable memory at the *user* resource group is 65,865MB.
- Distribution to the *R&D* resource group is always more than its demand, because memory is left over after the required amount is distributed to Sales. Memory demands from *R&D*'s VMs will always be met, and they will not undergo memory reclamation.
- Distribution to *Sales*, *US*, and *APAC* is less than their demand. When VMs placed under these resource groups attempt to access their full demand, ESX will reclaim memory from these VMs.
- Distribution to *US* and *APAC* are always in equal proportion. This is shown by D3 ~= D4.

Table 1(b) shows the memory distributed by ESX to individual VMs. It can be seen that

- Memory distributed to VMs under *US* are in equal proportion and are fully distributed, shown by D3 ~= T1. Similarly, for *APAC*, it is shown by D4 ~= T2 and for *R&D*, by D2 ~= T4.
- Memory available at *Sales* is fully distributed, shown by D1 ~= D3 + D4 + T3. Similarly, for *user*, it is shown by D0 ~= D1 + D2.

This section showed an experiment with a complex resource tree and equal *mem.shares* for the *US* and *APAC* resource groups. The following section demonstrates memory distribution when these resource groups have different *mem.shares*.

GROUP	VMs	VM DEMAND	TOTAL DEMAND	DISTRIBUTION
user	–		65,980	61,865 (D0)
Sales	1	2500	47,500	41,111 (D1)
US	8	2500	20,000	19,128 (D3)
APAC	10	2500	25,000	19,197 (D4)
R&D	6	3080	18,480	20,758 (D2)
A				

			S1 2372	S2-S7 2372	S8 2374	Total (T1) 18,982
				S11 1901	S12-S20 1902	Total (T2) 19,019
					Admin 2768	Total (T3) 2,768
R1 3414	R2 3441	R3 3442	R4 3444	R5 3446	R6 3448	Total (T4) 20,635
B						

Table 1. Steady state (6, 000 second mark) memory demand and distribution with **mem.shares** of **US** and **APAC** set to 100. (a) Memory demand from workloads under each resource group (**total demand**) and memory distribution by ESX to each resource group (**distribution**) (b) The memory distribution to individual VMs received from its parent resource group.

4.2.2 Modified resource attributes

A set of two experiments were conducted by assigning different values of *mem.shares* to the *US* and *APAC* resource groups. Each experiment was similar to the one presented in Section 4.2.1. They are described as follows:

150-100: *mem.shares* of *US* and *APAC* were set to 150 and 100 respectively. The *US* and *APAC* resource groups will receive memory from their parent group Sales in the ratio 150::100, or 3:2.

100-150: *mem.shares* of *US* and *APAC* were set to 100 and 150 respectively. The *US* and *APAC* resource groups will receive memory from Sales in the ratio 150::100, or 2:3.

The memory distribution to different resource groups and to different VMs was recorded as before. Figure 14 (a) and (b), respectively, show the memory distribution to resource groups and to VMs in those

resource groups. In Figure 14, the X-axis identifies each of the three sets of experiments; the left Y-axis shows memory distribution in MB. The right Y-axis in Figure 14 (a) shows SPECjbb05 score and kernel compile time in minutes. The data for experiment 100-100 is identical to the one presented in Figure 13.

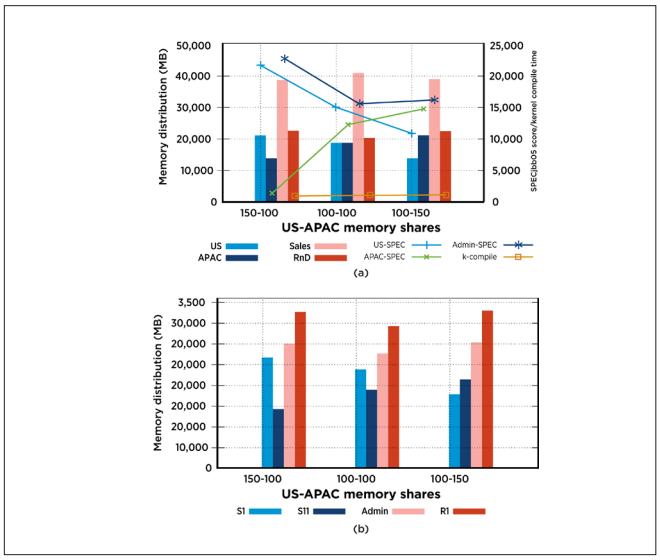


Figure 14. Memory distribution to resource groups and VMs for **mem.shares** of **US** and **APAC** set to 150-100, 100-100, and 100-150. (a) shows memory distribution to resource groups; average SPECjbb05 score for **US**, **Sales**, and **Admin**; and average kernel compile time for **R&D**. Average is taken across all VMs in the respective resource group, across 10 executions during steady state. (b) shows memory distribution to the **S1**, **S11**, **Admin**, and **R1** VMs.

From Figure 14 (a), using the left Y-axis, it can be seen that *US* and *APAC* receive memory distribution in the ratios 3:2, 1:1, and 2:3 for the *mem.shares* settings of 150:100, 100:100, and 100:150 respectively. *This is the key result from of this set of two experiments.* At 150:100 they receive 21,566MB and 14,373MB, at 100:100 they receive 19,128MB and 19,197MB, and at 100:150 they receive 14,419MB and 21,545MB. This shows that the *mem.shares* resource group attribute effectively controls memory distribution to resource groups. The distribution to *Sales* is the sum of distributions to *US*, *APAC*, and *Admin*. The remaining memory is distributed to R&D.

Sales receives a distribution of 38,972MB, 41,111MB, and 39,004MB respectively. The distribution is less than the *mem.limit* of 43,008MB. This might be because, at steady state, the SPECjbb05 workloads in the VMs might have slowed down owing to memory reclamation. This allowed the JVM to execute garbage collection in time to meet the workload requirements. Similarly, *R&D* resource group received a distribution of 22,285MB, 20,758MB, and 22,826MB respectively.

Performance of workloads is shown in Figure 14 (a), using the right Y-axis. From this figure, it can be seen that the kernel compile times for all three configurations are almost constant at 1,000 seconds. This is because the *R&D* resource group receives sufficient memory distribution in all cases. These VMs do not experience any memory reclamation.

The SPECjbb05 score for VMs under the *US* resource group progressively decreases from 21,364 to 14,852 and 10,716. This is because the memory distributed to this resource group, and hence to its VMs, progressively decreases as shown on the same figure. At the same time, the score for VMs under *APAC* increases from 1311 to 12,023 and 14,619. This is because the memory distribution to this resource group, and hence to its VMs, increases. The score for the *Admin* VM decreases from 22,564 to 15,318 and 16,028. Although this VM has full memory reservation and receives its full memory demand, its performance is reduced owing to the growing CPU demands from VMs in the *APAC* resource groups. CPU resources were not altered from the default values in this experiment.

Figure 14 (b) shows the memory distribution to the *S1*, *S11*, *Admin*, and *R1* VMs from the *US*, *APAC*, *Sales* and *R&D* resource groups respectively. *S1* receives 2677MB, 2373MB, and 1785MB respectively from the *US* resource group. Similarly, *S11* receives 1419MB, 1902MB, and 2136MB from *APAC*. *Admin* receives 3014MB, 2768MB, and 3022MB from *Sales*. *R1* receives 3776MB, 3442MB, and 3797MB from *R&D*. Figure 14 (b) shows that VMs placed under resource groups receive equal memory distribution from their respective parent resource groups. This is because the relative shares of VMs are all equal.

Hypotheses **H1**, **H2**, and **H3** also hold true at all times in these two experiments. They demonstrate fine-grained control over memory distribution among VMs by altering the attributes of the memory resource tree.

The experiments conducted in this section show how a complex memory resource tree can be used to effectively partition the physical memory of an ESX server. Resource tree attributes—*mem.resv*, *mem.limit*, and *mem.shares*—provide fine-grained control over the distribution of memory.

5. Conclusion

This article describes hierarchical memory resource groups in ESX. It shows how memory resource group attributes—*mem.resv*, *mem.resvLimit*, *mem.shares*, and *mem.limit*—can be used to dynamically partition the memory resources of ESX among powered-on virtual machines. Memory resource groups are a powerful tool for partitioning hardware memory among virtual machines in a flexible, scalable, and fine-grained manner.

6. Acknowledgment

Hierarchical memory resource groups was designed and implemented in ESX by Anil Rao and Carl Waldspurger.

References

1 I. Banerjee, F. Guo, K. Tati, and R. Venkatasubramanian. Memory Overcommitment in the ESX Server. VMware Technical Journal, pages 2–12, Summer 2013.

2 P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. ACM SIGOPS Operating Systems Review, 37(5):164–177, 2003.

3 D. Bovet and M. Cesati. Understanding the Linux Kernel. O'Reilly & Associates Inc, 2005.

4 J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In Proceedings of the 1998 USENIX Annual Technical Conference, pages 235–246, 1998.

5 E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. ACM Transactions on Computer Systems (TOCS), 15(4):412–447, 1997.

6 L. Cui, J. Li, B. Li, J. Huai, C. Hu, T. Wo, H. Al-Aqrabi, and L. Liu. VMScatter: Migrate virtual machines to many hosts. In Proceedings of the 9th ACMSIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '13, pages 63–72, 2013.

7 H. M. Deitel, P.J. Deitel, and D.R. Chofnees. Operating Systems. Pearson Education, 2004.

8 K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In ACM SIGOPS Operating Systems Review, volume 33, pages 154–169. ACM, 1999.

9 P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, OSDI '96, pages 107–121, 1996.

10 E. Group. Server Virtualization on Unix Systems: A comparison between HP Integrity Servers with HP-UX and IBM Power Systems with AIX. White Paper, 2013.

11 D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. In Middleware 2006, pages 342–362. Springer, 2006.

12 D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, pages 309–322, 2008.

13 G. J. Henry. The fair share scheduler. AT&T Bell Laboratories Technical Journal, 63(8):1845–1857, 1984.

14 J. Kay and P. Lauder. A fair share scheduler. Communications of the ACM, 31(1):44–55, 1988.

15 K. Milberg. IBM and HP virtualization: A comparative study of UNIX virtualization on both platforms. IBM developerWorks, March 20109.

16 M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In Proceedings of the 2005 USENIX Annual Technical Conference, 2005.

17 P. Sharma and P. Kulkarni. Singleton: System-wide page deduplication in virtual environments. In Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC '12, pages 15–26, 2012.

18 I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In Real-Time Systems Symposium, 1996., 17th IEEE, pages 288–299. IEEE, 1996.

19 B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. ACM SIGPLAN Notices, 33(11):181–192, 1998.

20 C. A. Waldspurger. Memory resource management in VMware ESX Server. SIGOPS Oper. Syst. Rev., 36(SI):181–194, Dec. 2002.

21 C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation, page 1, 1994.

22 T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '09, pages 31–40. ACM, 2009.

Improving Software Robustness Using Pseudorandom Test Generation

Janet Bridgewater
VMware, Inc.
janetb@vmware.com

Paul Leisy
VMware, Inc.
pleisy@vmware.com

Abstract

We present two pseudorandom test generators, each designed for testing very different types of software. These generators automatically create a wide range of test cases based on a deep understanding of the features of the software under test. They select from a range of input values most likely to create a relevant test. Those values include fixed, sequential, pure random, and weighted random. The strength of this approach is that it combines the expected with the unexpected and creates complex test conditions not covered by unit tests. We have run these generators, and the tests they have created, over long periods of time and have had success uncovering both interesting and complex bugs. This has resulted in improvements to the robustness of the software we are testing.

General Term: verification

Keywords: black-box, software testing, random testing, system-level testing, product verification

1. Introduction

As CPU hardware becomes more ubiquitous and more capable, the software running on that hardware grows larger and more complex. Testing of that software, which typically involves a multidimensional problem set, presents a challenge that is growing exponentially.

Testing of large and complex software products includes unit tests, integration tests, and system- level tests [1]. A unit test typically checks the correctness of one software module or one isolated feature. Testing several software modules or a group of related features is accomplished with integration testing. Finally, system-level tests are used to test the complete product, including all features that all the software modules implement. Tests written using the *white-box* method (also called *structural testing*) have intimate knowledge of the source code, structures, and private interfaces [2]. They attempt to verify all inputs and paths through the code. Tests written using the *black-box* method (also called *functional testing*) focus on verifying the functionality of the software with no knowledge of how it's accomplished [2]. They rely on a feature-specification document so that the test knows which features to expect and how they function and interact.

One major shortcoming of these tests and methods—especially concerning integration and system tests—is that the number of inputs, number of input values, number of code paths, number of features, and number of feature interactions can be so large that it's impossible to do exhaustive testing in a reasonable time frame. Other directed types of tests that are not exhaustive are typically handwritten and take considerable time and resources to achieve the desired levels of coverage.

In this paper, we present two pseudorandom test generators that automatically create black-box system-level tests. By repeatedly generating and executing test cases that have inputs intelligently formed from random and nonrandom values based on the feature set, more-efficient system-level testing is achieved.

The pseudorandom approach to testing represents a middle ground between exhaustive testing and uniform random testing. An exhaustive test iterates over all possible input values for all inputs. This is often not practical because it takes much too long for a test of this type to finish. A uniform random test picks pure random values for all inputs and runs the test until a good representative sample of input values has been tested. This method can waste time by testing similar inputs that are treated the same by the software under test. Also, special corner cases that depend on a single value might never be generated or might take a prohibitively long time to be realized.

The pseudorandom method of testing uses a mixture of pure random, weighted random, sequential, and fixed values for the inputs. It uses guided randomness, so the test is more efficient in generating the cases with the most coverage possibilities, creating very powerful results in the testing cycle. Pseudorandom testing is therefore a controlled random generation of a series of test cases. It understands the underlying features of the software to be tested and generates test-case combinations that a user or designer might have missed when writing tests by hand. Creating an effective system-level pseudorandom test requires a deep understanding of the product and insight into how various values presented to a range of inputs might or might not create relevant test cases.

Generation of a test case is based on a random seed. All values that make up that test case are derived from that seed. The same test case can be reproduced at any time by using the same random seed. This gives the tests created by the generator a repeatability quality.

The VMware virtual machine monitor (VMM) is part of the hypervisor that virtualizes the x86 guests' instructions, memory, interrupts, and basic I/O operations. The first test generator described in this paper focuses on verifying the instructions virtualized by the VMM. The test cases needed for this area include a mixture of valid and invalid instructions in random order; random and corner-case values for registers; random and corner-case values for memory addresses; and a sequence of target environments. Section 2 describes this test generator in more detail.

The VProbes™ feature that VMware created enables inspection of a virtual machine (VM) and the hypervisor that supports it. To learn more about an issue of interest, the user writes a simple script that is loaded on a running system. The VProbes engine that runs scripts (which are compiled into the internal VProbes language) is the primary focus of the pseudorandom test generator. However, the entire architecture is exercised when running the test scripts, which are combinations of the language tokens and mixtures of random variables. This is described in more detail in section 3.

Based on our experiences with these two pseudorandom test generators, we compiled a list of some common characteristics that make for an effective generator. Section 4 contains this list with some elaboration.

2. The VMware FrobOS Test with Random Instruction Generation

This section describes a program called 54055_randomCode64, which uses a random instruction generator to test the execution of x86 guest instructions by the VMM. VMware uses two methods to virtualize the guest CPU: Hardware Virtualization (HV) and Binary Translation (BT) [3]. In addition, an x86 interpreter is incorporated into the VMM to handle various situations, such as stepping to a safe point or avoiding recursive page faults when a page is marked not present for notification purposes. Testing the VMM's BT and its interpretation of x86 instructions is the focus of this test.

The simple approach to this test is to generate 100 random bytes and attempt to execute them as instructions. This is repeated for many hours. This test runs in a guest along side the FrobOS operating system, which is designed to create and support a general environment for running tests [4].

First, a VM is created, and the FrobOS operating system and the test is loaded into guest memory. Next, FrobOS begins executing and sets up a minimal environment for the test to run in. This includes switching to protected mode, setting up memory mappings, and initializing the interrupt vectors and handlers. FrobOS then hands off execution to the test, which does the final environmental setup. Finally, the test enters a loop, which first stores 100 random bytes into a code page and then branches to that same page. The interrupt handler for faults provided by FrobOS returns to the test inside the loop. After a predetermined amount of time passes, the test exits the loop and ends, and the VM is powered off. The test also provides a custom timer -interrupt handler to detect when the random bytes have formed instructions that cause infinite loops.

The following subsections describe the initial version of the test, some enhancements that we made, experimental results including an example bug that was found, and future work that can be done to improve the test. It is assumed that the reader has some familiarity with the x86 architecture.

2.1 Initial Version of Test

The initial version of this test randomized only the 100 bytes to be executed, keeping the remaining environment and setup fixed. For example, the registers were set to all zero except for the stack pointer. The stack page was cleared to all zeroes. The initial environment was set at the lowest privilege level (CPL = 3). The target environments contained in the Global Descriptor Table (GDT) and Local Descriptor Table (LDT) were left as FrobOS had set them. No result checking occurred, and the test simply ran thousands of random sequences over a long period of time, using the first x86 fault as the iteration exit control. The VMM either successfully handled each case or failed with a software assert. Using the guest execution controls provided by FrobOS, the test was run in two different guests: an HV guest and a BT guest.

2.2 Adding a Reference Model

The first major enhancement we made was to use HV mode as the reference model. When a test case finishes, it's important that the results are verified and match the architecture specification. This can be done by comparing the results of the software (BT and interpreter) with the results of the hardware (HV mode). Specifically, when the test case is run in an HV guest, the random sequence of bytes is run a second time through the VMM's interpreter, and the results of the two runs are compared. Likewise, when the test case is run in a BT guest, the random sequence of bytes is also run through the interpreter and the results compared. In this way, the results in all modes are validated. In this context, the results of the test case are the contents of the x86 registers, the flags, the instruction pointer, and the exception type. Future plans include comparing the contents of memory as part of result checking.

A major complication to using a reference model in this situation is that many behaviors in corner cases of the x86 architecture are model-dependent. For example, some instructions might or might not wrap the address when the memory operand crosses out of the highest page in memory. So the three modes—HV, BT, and interpreter—might all respond in architecturally correct ways but not match one another. We created a table of all these discrepancies inside the test so that the test could automatically detect these cases and skip reporting them as failures.

2.3 Randomizing Registers and Stack

The second major enhancement we made was to introduce randomness to the contents of the registers and the stack page. This vastly improves coverage for data manipulation and address generation that the random instructions might conjure up. It was decided to randomize the contents of registers and the stack page around 80% of the time, leaving them set to zero the other 20% of the time. This enables the corner case of zero to be more frequent than 100% random would normally give.

2.4 Target Environment Initialization

The third major enhancement we made was to initialize the target environments. This helps the random sequence of bytes touch more areas of the architecture. Whenever a far call or far jump instruction is generated by the random bytes, various entries in the GDT and LDT are accessed as the target environment. Initializing these tables to be larger and to contain diverse values opens up more test cases to be generated. Instead of randomizing the entries in the GDT and LDT, we chose a sequence that covers most combinations.

The end result of these last two additions is that random instructions with random register values create random pointers into the GDT and LDT that contain a fixed sequence. Also, random values contained in the stack might be used as pointers into the GDT and LDT if the instruction happens to include a stack operation.

2.5 Collecting Statistics

We added statistics to this test in the form of recording the specific type of x86 fault that occurred after the randomly generated test case was run. Page faults (#PF) on memory accesses were further broken down into operand (oper), instruction fetch (if), stack fetch (stck) and GDT/LDT accesses. Also, the average bytes executed before encountering a fault were tracked. This is a sample of statistics recorded after the test was run for 4 minutes:

```
****** Exit Statistics ***********
 *   oper #PF count 72111
 *   if   #PF count 7680
 *   stck #PF count 147
 *   gdt  #PF count 255
 *   ldt  #PF count 234
 *   (0)  #GP count 16731
 *   int  #GP count 5112
 *   gdt  #GP count 3456
 *   ldt  #GP count 3612
 *        #UD count 7932
 *        #DB count 1398
 *        #DE count 150
 *        #SS count 3
 *        #BR count 27
 *        #BP count 0
 *   gdt  #NP count 105
 *   ldt  #NP count 93
 *       LOOP count 65
 *      TOTAL count 119111
 *      avg bytes executed 4.57798
 *********************************
```

After running the test for 12 hours, we compared the statistics between 64-bit, 32-bit, and 16-bit code. This revealed a deficiency in our 16-bit code initialization. The random instructions had a higher percentage of page faults (#PF) and lower average number of bytes executed compared to 64-bit and 32-bit code. Further investigation revealed that the 16-bit code was executed in a 32-bit address above 64K so that all jumps and calls resulted in a page fault (#PF).

2.6 Adding Debug Facilities

We enhanced the debug capabilities of the test so that it was easy to see the results and easy to narrow down the exact instruction or instructions involved in the mismatch. The test dumps relevant information when the results of the reference model differ from the results of the VMM's interpreter. Parts of the code and stack pages are dumped, as well as the instruction pointer, registers, and fault information. See subsection 2.7 for a sample.

The random sequence of bytes that produces a mismatch usually includes instructions and registers that do not pertain to the failure. The test provides a way to run with specific bytes in the code page, instead of random bytes. Running one or more subsets of the original random bytes enables the critical instructions and register values to be discovered.

2.7 Experimental Results

After the VMM was tested with unit tests and various operating systems with various workloads, we ran the pseudorandom test on it. Over a period of 6 months, which included testing and making enhancements, a collection of 22 corner-case bugs was uncovered. These were all new bugs that were not found by a large collection of handwritten directed tests. A description of one example follows.

For this particular case, the registers were initialized to zero except for the stack pointer, which was set to 0x8a1c800. The page after the stack page (0x08a1d000) was marked not present in the page table entry. The results shown for "run1" are from the HV run, and the results shown for "run2" are from the VMM's interpreter run. This failure was detected after the test ran continuously for approximately 12 hours.

```
FAIL: CPU 0: randomcode.c:782: register esp mismatch

run1.esp(0x8a1d001) != run2.esp(0x8a1cffd)


     eax      ebx      ecx      edx      esp

run1 00000000 00006100 00000000 00000000
                                         08a1d001

run2 00000000 00006100 00000000 00000000
                                         08a1cffd

run1:#PF at rip: 08a1a02d, fault addr 0x08a1d001

run2:#PF at rip: 08a1a02d, fault addr 0x08a1d000


Emitted code is for 32 bit mode

First random code byte is at 0x8a1a019
```

```
0x8a1a000: cd 42 bc 00 c8 a1 08 8e
0x8a1a008: de 33 c0 33 db 33 c9 33
0x8a1a010: d2 33 ff 33 f6 33 ed 39
0x8a1a018: c0 44 69 f8 a8 6d 28 3f
0x8a1a020: 1c 4c 8d 07 b7 61 98 36
0x8a1a028: 00 c0 0f b3 f8 07 7d f8
0x8a1a030: ec eb b2 e5 89 17 cf 73
```

```
0x08a1a019 | 44           INC %esp
0x08a1a01a | 69f8a86d283f IMUL
                          %edi,%eax,$0x3f286da8
0x08a18020 | 1c4c         SBB %al,$0x4c
0x08a1a022 | 8d07         LEA %eax,(%edi)
0x08a1a024 | b761         MOV %bh,$0x61
0x08a1a026 | 98           CBW %eax
0x08a1a027 | 3600c0       ADD %al,%al
0x08a1a02a | 0fb3f8       BTR %eax,%edi
0x08a1a02d | 07           POPSEG %es
0x08a1a02e | 7df8         JGE -0x8 ;0x28
```

The first two lines identify the esp register as the one register that is different between the two runs. The next five lines dump the registers, the rip value from the fault, and the fault address for both run1 and run 2. (To shorten the output and keep it one line, the esi, edi, and ebp registers are not shown here.) The memory dump in hexadecimal format shows the initialization code starting at page offset 0x000 and the random bytes starting at offset 0x019. Finally, the random bytes are shown in disassembled format.

Not all instructions created by the random bytes were essential in producing the bug. Use of the debug tools noted above reduced the core instructions needed to the following sequence. (The "INC eax" was not needed but was added to count the times through the loop)

```
0x08a1a000 | 44           INC %esp
0x08a1a001 | 07           POPSEG %es
0x08a1a002 | 40           INC %eax
0x08a1a003 | ebfc         JMP -0x4 ;0x1
```

We analyzed these results and determined that the reference model, represented by HV mode, read 2 bytes off of the stack and bumped the stack pointer by 4 for each "POPSEG es" executed. The VMM's interpreter, on the other hand, read 4 bytes off of the stack and bumped the stack pointer by 4 for each "POPSEG es" executed.

2.8 Future Enhancements

This random test would benefit from having error handlers that can validate the error, repair the environment, and reexecute the random code. Currently, when a fault occurs, the test case is finished and another sequence of random bytes is generated. This enhancement would bolster testing of nonfaulting paths in the VMM. The error handlers would need to have detailed knowledge of the x86 interrupt mechanisms and the encoding of instructions to ascertain the reason for the fault. In some cases the hardware does not provide enough information to pinpoint the cause accurately. This could be ignored and the next instruction resumed to continue the test case.

Another area that needs attention is the validation of memory pages. Adding checkpointing to the code, stack, and data pages would enable mismatches involving memory to be identified quickly. Currently they are detected only when two instructions (one write, one read) access the same memory location.

3. VMware VProbes Language and Probe Pseudorandom Generation Test

VMware VProbes is a facility for understanding what is occurring in a VM or on the virtual infrastructure on which the VM runs. It is an open-ended tool that enables the user to answer questions about the system. [5]

To use VProbes, you write simple scripts in a C-like language, developed internally at VMware, called Emmett. The script is loaded directly into a running system without the need to compile the script or restart the system. The Emmett compiler creates Scheme-like output called VP. The VP output is processed to implement *probes* into a running system by one or more VProbes engines, a VMkernel module that hosts an output-serialization buffer and a virtual device from which the front end can read the output data.

You can think of a probe as a callback that executes when the control in the observed system reaches a particular code location or when a certain event occurs. When the point of interest or event is reached, the code specified in the script for that probe is executed. There are three classes of probes:

- **Static probes** – These probes correspond to predefined points of general interest in the VMware software (e.g., VMkernel, VMM, or VMX), such as the point of the transmission of a network packet or the point of delivery of an interrupt.
- **Dynamic probes** – These probes represent breakpoints on arbitrary instructions or watchpoints on an arbitrary piece of data in the observed system. Unlike static probes, dynamic probes are implementation-specific. You must have access to, and knowledge of, the VMware source code before you can use dynamic probes.
- **Periodic probes** – These probes trigger periodically during the execution of the observed system.

Placing probes into scripts gives the user visibility into a running system for exploring performance or correctness issues. Most important, VProbes is "safe" in that it ensures that you can never change the state of the system—which includes crashing or hanging the system.

The life cycle of a script in the VProbes architecture (see Figure 1) includes:

- Loading a user script through a command-line interface into a running system—either VMkernel or a powered-on VM
- Compiling the script written in Emmett to the internal VP language
- Loading the VP output into the VProbes engine
- Executing the code in a probe body when a probe point is hit
- Creating data/output
- Unloading the script, leaving the system in the same state in which it started

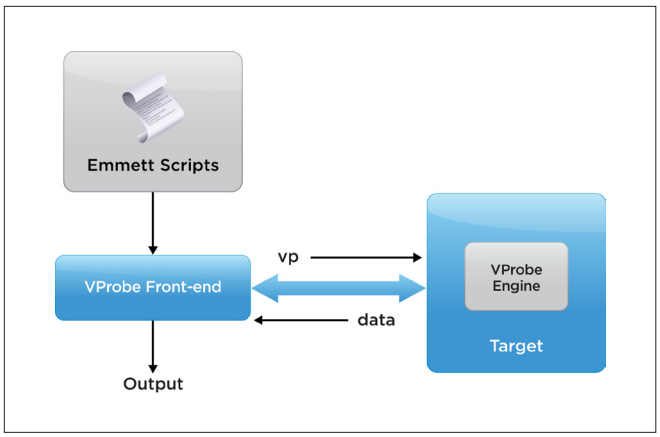


Figure 1. The VProbes Life Cycle

All of these layers need to be tested so that a user writing any script can be **guaranteed** that the script will compile (or return a syntax error), load, and run with any available probe point in the system and produce output. In other words, a user can seemingly create an infinite number of scripts combined with thousands of available probe points. This could be considered the perfect storm for testing. Writing only unit tests for this environment will not touch all pieces in this system. In contrast, designing a pseudorandom generator to create scripts to load and run in a system can give us more thorough testing of VProbes.

This section describes the generator we are using and continuing to enhance to make VProbes robust. It is assumed that the reader has some familiarity with programming languages and general computer-architecture terminology. VProbes concepts and terms are explained herein.

3.1 VProbes Fuzzer Testing

The pseudorandom test designed for VProbes verification is a Python-based application called *fuzzer*. It primarily focuses its test generation on the VP (Emmett-compiled output) language and combination of probe points because the VProbes engine is the heart of VProbes in a running system. However, exercising all pieces in the VProbes infrastructure is included in the fuzzer testing.

Before the VProbes fuzzer test runs, targeted unit tests are run to ensure that the very basics are working. For example, does a script load? If a simple error occurs, the unit test can call it out quickly. After the suite of unit tests runs successfully, the fuzzer can start generating VProbes scripts that cover numerous language symbols and probe combinations to create interesting and more-complex conditions.

A Python wrapper infrastructure is used to initialize and control the test environment. VMware ESXi™ server and hosted platforms Linux or Fusion, as well as local and remote executions are each supported. It determines the platform currently being run on, enables VProbes in the environment and powers on the VMs. On a server, the VProbes architecture can be tested in the VMkernel (the operating-system core of ESXi) and within a powered-on VM. In the hosted environment, VProbes is tested in a powered-on VM. The environment determines what is valid for the generated fuzzer script. A list of probe points specific to the environment is gathered. For example, in the VMkernel alone there can be as many as 84,000 functions to probe. Globals, which are read-only built-in variables providing access to some portion of the virtual hardware and the host state—for example, `NUMVCPUS`, which returns the count of virtual CPUs—also vary between the tested environments. Probes and globals are mined when the fuzzer starts up, and they are kept in a symbol table to select from during test-script creation.

The ability to run the same test code in a variety of environments is a critical benefit for efficiency, usability, and support. A user who knows one test and its user parameters can run it in multiple environments. When new features in the VProbes architecture are developed, test developers can easily incorporate them into an existing test infrastructure that is run in combination with the other features. So we're not just testing the new feature independently, but also the interaction among features. This creates complex test cases and provides efficiency in test development; we're not spending time writing unit tests of the combined features but rather incorporating new with existing features to be randomly combined. For example, a new built-in function for the Scheme-like language is designed and added into the fuzzer generation for testing. It is tested inside a probe, as an operand of an arithmetic expression, in nested expressions and inside a user function. It is tested in all applicable pieces of the VProbes architecture.

3.2. Fuzzer's Use of Randomness

The random number generator that the fuzzer uses is imported from Python's random module. It first uses `random.seed(options.seed)` to start the generator base, followed by instances of `random.randint()`.

The fuzzer uses random-number routines that range from basic generation of integer operands to making all of its selection decisions. The important principle is that the sequence of generated random numbers is always the same based on the seed.

The use of randomness is controlled so that we don't have a generated script of "garbage" but instead one that is syntactically correct. Valid probe points are selected, operands are set up in expressions, identifiers are created, and their type is set. The actual operands are random values and can be the result of a deeply nested expression. If a function has a user register as its operand,

at times the register number is not always between 0 and 15. We skew the selection to "mostly" pick a valid register number so that it makes some sense—but not always, so then we're focusing the testing on what is relevant.

3.3 Fuzzer User Options

A user interface to the test or script generation and execution is a critical piece of the infrastructure. The user should have the ability to narrow down the testing focus and be able to rerun, continuously run (for intermittent failures), or halt when the failure occurs. A variety of knobs can be put into place to specifically control what is in the scripts or how to execute the scripts. An example of script-generation control is that the user can ask to have a `PROFILE` probe randomly placed in every fuzzer script. During VProbes debug, we found that servicing a `PROFILE` probe while other function probes are active can lead to interesting scenarios, so we wanted as many `PROFILE` probes in scripts as possible. Therefore we let the user control this selection of probe. Or, a user can change the default runtime from 4 seconds per script to any amount of time if the script runtime is affecting the results of probe firing. When a user is trying to isolate an environment to debug an error, having these controls from a user interface gives the user more control over the pseudorandom setup.

3.4 Example of Generated Output

Here is one example of test-script-generated VP, the Scheme-like language for internal VProbes execution. It shows the pieces of the language that have been each systematically yet randomly selected and formatted into VP. This script is syntactically correct but doesn't make any real-world sense. The VProbes engine should be able to load this script without error and run it without any dangerous side affects.

The example includes setting up identifiers of integers, strings, and VP data structures (`id0xx`), and a user function (`id004`). It includes predefined globals (`RIP, CR8`), two static probes (`HV_Resume` and `HV_Simulate`), and one periodic probe (`TIMER:109usec`), which were randomly selected, including setting up a random period of 109usec. Built-in functions—`strcmp()`, `timetostr()`, `format()`, and `fetchadd()`—are included in this script, all randomly selected with valid but random operands. The nested expressions inside the probe body are generated here to a depth of 7 expressions; however, depths of up to 50 expressions are valid. This depth is kept track of so that no infinite recursion occurs during script creation.

The end result is that we have increased confidence that if these controlled random scripts can load and execute, then any script a real user writes will be processed correctly.

```
; Initial seed was 1376542692
; Random seed was 490126
; Maximum nested depth 7
(vprobe HV_Resume)
(definteger id000  174936550442)
(vprobe HV_Simulate
   (try (for "b\x05{\x17=?3^9)>WjE]?&\x0" (do )
     (timetostr (cond )) 173935)
      (excdesc))
   (substr (excdesc) CR8 ARG3)
   (excdesc))
(defbag id001 1)
(defbag id002 1 "perdomain")
(defbag id003 1)
(defaggr id009 3 0 "perdomain")
(defstring id010 "&}>\x01\x1a\t\x0esvVH[;C;")
(defaggr id011 3 0 "perdomain")
(definteger id012 131799370 "perdomain")
(defbag id013 1)
(defaggr id014 7 6)
(defun id004 (id005 id002 id006 id003 id007
id008)
   (do (- (strcmp (do (aggr id009 (VMCB_
   EXITINFO2
    VCPUID ARG0) () VMCB_TLBCTL)
    "<%P\x1c\x19\x03IB") (excdesc))
    (strchr (format "%s%s%x%x%u%d%d\n" (do
    "vR\x15i2l\x\'^x") (setstr id010
    PROBENAME)
    (aggr id011 (ESLIMIT R11 DR6) () (cond ))
           APIC_BASEPA LDTR IDTRLIMIT
    VMCB_EXITCODE) (fetchadd id012 (< TSC
    RDI))))
      "\x12+^Y.:\x0cyy\x00\x1c\rSYF[U\\")
   id006
   (excname)
   (cond )
   (format "%x%s%d%u\n" (offatstrcpy GROUPID
   DR6)
   (cond (1 (do "tea!ee"))) (offatseg (cond
   )) (do
   ))
   12131450872661096744
   (getvmcs CSBASE)
   (excname)
   (bagremove id013 (aggr id014 (IDTRLIMIT
   EFER
   VMCB_VAPIC ARG4 RIP TRAR CR4) (id006
   PROBENAME
   PROBENAME id007 PROBENAME id006) RAX)))
```

```
(defstring id015)
(defbag id016 2 "perdomain")
(defbag id017 2)
(vprobe TIMER:109usec
   (excname)
   (timetostr 6551330643392684659)
   (try (clearaggr id011)
      (cmpxchg id012 (aggr id014 (RBP IDTRBASE
    SSBASE VMCS_VMENTRY_INTR_INFO RAX WORLDID
    ARG4) (id015 id010 id015 id010 id010 id010)
    RBX) (&& RFLAGS
      (bagpeek id016 (printf "%d\n" ARG0)))))
   (+ CR2 bagpeek id017 IDTRLIMIT))
   (format "%x%x%d\n" DR7 (logint id000)
      (offatret VMCS_INSTRLEN))
   (do "m\x14*\x1c<qI[5\r")
   "H\"6GA5aqM:O\x08[\x1f]Aj")
```

3.5 Testing Results

Instead of using a reference model for comparison, the VProbes fuzzer relies on script load errors, a monitor crash, a CPU hang, or a compiler error to call out failures when the scripts run. A crash is typically in the form of an assertion violation in the VProbes code.

Here are some real examples of problems that the fuzzer uncovered as new features were being implemented:

Example 1:

An example was generating the built-in `timetostr()` function (generating a human-readable timestamp) with a much-too-large operand:

```
(timetostr 14437534330304277835)
```

This built-in was embedded in a `for` loop found in this generated-script snippet:

```
(vprobe SCSICommandSplit
 (for PRDA_ADDR
    (aggr id001 (TSC_HZ TSC
    GROUPID TSC_HZ PRDA_ADDR RDTSC) (
    id002 id003 id004)
       (strcmp (strerror GROUPID)
       (timetostr 14437534330304277835) ) )
 id005 (do ) )
```

The script ultimately caused a no-heartbeat panic (crash) of the VMkernel. The `timetostr()` built-in becomes very expensive for large (and unrealistic) dates. For dates between years 1900 and 2100 it takes about 1 microsecond. For very large dates, half a million years from now, it takes 2.5 milliseconds. Running `timetostr()` in a loop can take a few seconds and hang the system. To fix this, we put a bound of 252 (year 2112) on the argument of this built-in and increased its internal cost to execute.

This problem was found relatively quickly after `timetostr()` was added into the fuzzer, because operands can be very randomly large and the `for` expression that creates a loop is generated frequently.

As product developers, we must be ready to handle this type of obscure or unexpected operand, because there is no way to predict what a user will put in a script either purposely or via a finger-check.

After a problem found by the fuzzer has been identified and a fix is in place, a small unit test of the case can be created for ongoing verification of the fix.

Example 2:

This random combination of two probes led to a system panic in the VMkernel:

```
(vprobe
  VMK:ENTER:vmkernel.IDTSaveAndFixupGS)
(vprobe VMK:PROFILE:180usec)
```

Firing the profile probe while another VMkernel probe is being serviced needs special attention that was uncovered by this combination. During unit testing, we didn't combine this particular function probe with a `PROFILE` probe of high frequency to test the condition prior to running the fuzzer.

The issue here concerns the following scenario: (1) an interrupt happens, (2) a Non-Maskable Interrupt (NMI) occurs very early during that interrupt (i.e., before the interrupt handler fixes %gs), and (3) a probe is placed early on in the NMI handler (i.e., before the NMI code itself fixes %gs). In this case, the VProbes code crashes while using the invalid %gs, to access VMkernel's Per-PCPU private data area structure (PRDA), causing the system panic.

This failure took some time to hit because the scripts that the fuzzer was generating averaged 1–5 probes each. With the selection pool of tens of thousands of probes and needing a `PROFILE` probe combined with it, it took many weeks combined with some user/test developer intervention to hit this case. After realizing the severity of the problem and examining many generated scripts, we added two modes to fuzzer testing: (1) the ability to add `PROFILE` probes to all scripts and (2) generating scripts that iterate through all available function probes until they are all touched. Now the combination of these two modes enables a `PROFILE` probe and all function probes to execute in the same script within about two hours of running the fuzzer.

Example 3:

The fuzzer now has a mode of generation that iterates through all of the available function probes on the system. This can create a random script with hundreds of probes in it. An Emmett script with more than 600 probes was generated and caused a hang in the system. When a large (but smaller than 32K) script is sent to the VProbes daemon (vprobed) and the compiled VP script is larger than 32K+4K, it caused the daemon to hang. Vprobed was reading only the first 32K of data from the Emmett process, filling up its internal buffer. When it detected the overflow, it waited for Emmett to finish compiling, but Emmett hung because the 4K pipe it used to send the Emmett output to vprobed was filled up—so vprobed and Emmett were both "stuck" waiting for each other.

It's very unlikely that a user will ever create a script with so many probes in it, but uncovering an error such as this reveals a design flaw that enables us to reevaluate the design while fixing it. We have unit tests for "large scripts" and "too many probes," but they tend to stress both at the limit and "+ 1" of the limit. There was no targeted testing for anything so big that it couldn't even be compiled.

When a failure occurs, the script that was generated is saved so that it can be singularly rerun for debug or retesting to verify a fix. We can also regenerate the exact same script, because the fuzzer has the ability to use a base seed (passed in `--fuzzerSeed`) to re-create all follow-on random selections. This is extremely useful and important both in debugging the initial script generation and in verifying a fix. Repeatability is necessary in pseudorandom test creation to ensure that we can "get back" to the exact same scenario. It is a valuable time saver for debug and is "proof" that generation is under control instead being subject to complete randomness.

3.6 Future Enhancements

The current fuzzer code for VProbes testing covers the VP language, the user language (Emmett) compiler, and loading user scripts with the expectation that they'll load without error and run successfully. Enhancements currently in process include loading random scripts that are expected to generate load errors. This approach to generating scripts is accomplished by walking a grammar tree and selecting any random node in the tree that can create recursive patterns of expressions until a terminal is reached. By allowing load errors and expanding the allowable combinations of grammar in the scripts, we're testing many more code paths while ensuring that the system continues to run.

Another enhancement in progress is the speeding up of script creation. If we can generate and execute more scripts in the same amount of time, then our overall coverage is increased.

4. Characteristics of a Pseudorandom Test Generator

Based on our experiences working with a generator, the following are some common elements that we found to be key to making the tests as valuable as possible.

4.1 Unit-Test Product Fundamentals First

In some cases, the pseudorandom test cases are both generated and run on the software product itself. The 54055_randomCode64 program falls into this category. It is important that unit tests are first run on the product so that some degree of confidence is achieved in the test-generation and checking operations. The unit tests should cover common cases of all common operations likely to be used by the generation and checking code.

4.2 Ensure Repeatability in Test Cases and Environment

A pseudorandom test could generate a seemingly infinite number of test cases, each based on a random number, or seed. The seed is used for making decisions and initializing data for the test being created, so it is important that the random-number generator is sound and not skewed toward any set of numbers, is evenly distributed, and generates a large repeatable pool of numbers. The seed is also used to start the exact same sequence of events to re-create the test case that caused an error or verify a fix for that case. When a test case is rerun, the test-case seed is passed to the generator in place of a newly generated seed. All pieces of the test and architecture that could influence the results of a test case must be set up the same as in the failing case—for example, initialized registers, the order of generated instructions, operands used by instructions, and memory or page table initialization. A repeatable environment is best for rerunning the identical test case when failures are being debugged and fixed.

4.3 Focus the Randomness

Every time the pseudorandom test runs, a new set of conditions is generated to run as a test case. When randomness is introduced into test-case generation, there is the danger of testing only error cases that are rarely, if ever, realized in the real world. For example, if there are 1 million ways to create environments and only 10 are actually used and are valid, then creating the test-case environments with a uniform random method results in very inefficient testing. In this situation it probably makes more sense to randomly select from 12 environments—10 that are valid and 2 that are invalid.

This focus, or fine-tuning of the randomness, requires knowledge of the environment and the features being tested and their interactions. A good balance is needed, because taking too much randomness out will also diminish the effectiveness. Some inputs might be best tested using pure random values, and others might benefit from a small range of fixed values.

4.4 Verify Results

After a pseudorandom test case is generated and run, its results must be verified. Verification can be as simple as a pass/no pass or as complex as verifying all outputs and the complete state of the software. One way to accomplish this is to run the pseudorandom test case on a reference model as well as on the product being tested, then compare the results of each run. The reference model could be hardware, a simulator, or autogenerated result tables. Some architectures are so complex that corner cases are not well-defined, and the results are known to vary for each product release. In these cases, a list of these deviations is needed to prevent false-positive miscompares.

Another aspect that needs to be considered is the size of the random test case and how often the results are verified. A failure in the results near the start of a random test might be covered up by running the remaining part of the test case. Keeping the tests smaller or calling out a failure as soon as it occurs eliminates this scenario.

4.5 Collect Statistics

Because the test-case generator always includes some random elements, it's important to collect statistics to verify and fine-tune the coverage of the created tests. If possible, many generated test cases should be analyzed for the areas that they cover. Accumulating these statistics over a period of time makes it possible to determine if the focus of the randomness is doing what it was designed for, and to ensure efficiency for creating relevant coverage during the testing runtime.

It's not always practical to gather statistics at the finest granularity. Sometimes some assumptions must be made that the randomness is covering what it was designed for. For example, if a product has 1,000 invalid values to a particular input, gathering statistics on 10 of them could provide a good level of confidence that the randomness is working as designed. This assumes that the values are uniformly distributed over a known range and are uniformly generated by the test.

Looking toward future work, these statistics could be the basis for introducing a feedback loop. Based on the statistical results, it's possible to adjust "knobs," added to the test generator, to influence focus toward testing features. The gathered statistics would then influence the selection of inputs to the next set of test cases.

4.6 Infrastructure

Creating, running, and checking these tests include several pieces: a pseudorandom test generator, a mechanism for transferring and running the test on the desired platform, and ways to retrieve and check the results. It is an infrastructure that facilitates testing basic and complex cases and is multipurpose in execution. It is beneficial for time-efficiency, comprehensive coverage, and debugging.

5. Related Work

A tool called DART (Directed Automated Random Testing) exists for automatically testing software [6]. It is specifically focused on unit testing using a white-box method. DART examines the source code of the software under test, picks random values for the inputs, and then runs the program concretely. It also runs the program symbolically, calculating constraints on the inputs. The next iteration of the program uses pseudorandom values for the inputs based on the constraints from the symbolic execution. In this way, DART directly targets different paths in the software under test. The pseudorandomness is focused by the symbolic execution of the source code.

In contrast to DART, our test generators are focused on testing features (black box) at the system level. They focus the pseudorandomness by incorporating knowledge of the features into the generators. These two testing tools are complementary—one targeting the structure of the code, the other the implementation of the features.

6. Conclusion

Testing is absolutely necessary during all product phases: design, development, fix verification, and ongoing regression runs. We found that using the pseudorandom method of testing is a valid and efficient way of testing a wide range of software products, from system-level code to compilers, during these phases. Our generators have been critical in finding problems in our complex software and are a good complement to other test methods such as directed unit tests and running real-world system-level workloads.

Acknowledgments

We would like to acknowledge Marco Sanvido and Stuart Easson for their work on the 54055_randomCode64 test. Marco is the original author of the test, and Stuart identified many enhancements in addition to implementing the reference model portion. We would also like to acknowledge Jon DuSaint for his initial coding and groundwork of the VProbes fuzzer.

Many thanks to Rakesh Agarwal, Radu Rugina, and Jerri-Ann Meyer for their guidance in writing this paper and for their reviews.

References

1 Laurie Williams, A (Partial) Introduction to Software Engineering Practices and Methods (2010–2011). http://www.academia.edu/2615624.

2 IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

3 Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam, The evolution of an x86 virtual machine monitor (2010) *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, 2010 Pages 3–18.

4 Stuart Easson, FrobOS is turning 10: What can you learn from a 10 year old? (2012). VMware Technical Journal, Winter 2012. http://labs.vmware.com/academic/publications/FrobOS-vmtj-winter2012.

5 https://communities.vmware.com/community/vmtn/developer/forums/vprobes.

6 Patrice Godefroid, Nils Klarlund, and Koushik Sen, DART: Directed Automated Random Testing (2005). *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Pages 213–223.

Toward an Elastic Elephant Enabling Hadoop for the Cloud

Tariq Magdon-Ismail
VMware Inc.
tariq@vmware.com

Razvan Cheveresan
VMware Inc.
rcheveresan@vmware.com

Andy King
VMware Inc.
acking@vmware.com

Richard McDougall
VMware Inc.
rmc@vmware.com

Mike Nelson
michaelnnelson@gmail.com

Dan Scales
dan.scales@gmail.com

Petr Vandrovec
VMware Inc.
petr@vmware.com

Abstract

A prerequisite to architecting any cloud service is to have an elastic platform that can dynamically respond to the varying demands placed on the service by multiple tenants while meeting their service-level agreements (SLAs). Ideally, such a platform also optimizes for the most efficient use of the underlying infrastructure.

What this means for a framework such as Hadoop is that in order to offer it up as a multitenant, fully isolated service, we need to be able to expand/shrink its cluster size (capacity and footprint) dynamically. However, because the Hadoop compute and data layers are tightly coupled, expanding or shrinking in real time—say in response to a sudden surge in demand—is a challenge.

In this paper we examine the nature of Hadoop and the hurdles that need to be overcome in order to turn it into a well-performing elastic service. We explore some solutions that use existing technologies and provide experimental data that shows that we can achieve our goal with acceptable levels of performance. We further characterize the remaining overhead and present novel solutions to significantly optimize performance even further.

Categories and Subject Descriptors: C.4 [Computer Systems Organization]: Performance of systems – design studies, measurement techniques, performance attributes

General Terms: Measurement, performance, design, experimentation

Keywords: Big data, Hadoop, MapReduce, virtualization, elasticity, multitenancy

1. Introduction

The common approach to virtualizing many applications is to perform a physical-to-virtual (P2V) migration whereby the physical deployment is directly cloned into virtual machines (VMs). Hadoop is no different. Although this is a reasonable first step, there are a few drawbacks specific to Hadoop that make this model less than ideal in a cloud environment, one of which has to do with elasticity. Adding/removing Hadoop nodes to increase/reduce compute resources that are available to jobs is a cumbersome and coarse-grained activity due to the tight coupling between the compute runtime and data storage layers [14]. As we will show in Section 3, this makes it difficult to dynamically scale the virtual cluster to either make use of spare physical capacity or relinquish it to another workload. In other words, the virtual Hadoop cluster is inelastic. To gain elasticity, we need to be able to separate the compute layer from the storage layer so that each can be independently provisioned and scaled. This level of flexibility would enable Hadoop to share resources more efficiently with other workloads and consequently raise the utilization of the underlying physical infrastructure. Moreover, this *separated architecture* would allow for more efficient hosting of multiple tenants, with their own private virtual clusters. It goes beyond the level of multitenancy offered by Hadoop's built-in scheduler and security controls by relying on the hypervisor for much stronger VM-level security and resource isolation guarantees. Furthermore, because each compute cluster would be independent, each tenant could have his or her own version of the Hadoop runtime. These characteristics are all necessary before a very flexible, elastic, and secure Hadoop as a service can be provided.

2. Related Work

Separating Hadoop's compute engine from its data layer has already been discussed in [7] and [9] as a way of achieving flexibility in physical deployments. Although virtualization is considered in both papers, the proposed architectures are in fact hybrids with only a subset of Hadoop's services being virtualized. Depending on the paper, it is recommended that either the Hadoop Distributed File System (HDFS) or MapReduce layer be directly run on native hardware. Furthermore, they both identify the bandwidth limitations of the physical network as a potential bottleneck in achieving separation at scale.

Our approach, on the other hand, drives Hadoop elasticity a level further by virtualizing the entire set of Hadoop services while providing compute and data separation. Moreover, we utilize the strengths of virtual networking coupled with an awareness of Hadoop topology to optimize for performance and scalability.

3. Separating Compute from Data

In typical Hadoop deployments today, the compute and storage engines (known as the TaskTracker and DataNode, respectively) run inside each node, so the life cycle of a node is tightly coupled to its data. Powering it off means that we lose the DataNode, needing to replicate the data blocks it managed. Similarly, adding a node would mean that we need to rebalance the distribution of data across the cluster for the optimum use of storage. This decommissioning/provisioning and the large volume of data transfers involved makes responding to changing resource pressures slow and cumbersome. A further complication is that compute and storage capacity requirements change at very different velocities, and we cannot respond to these different needs independently or efficiently. A potential solution is to scale storage by adding VMs (to add DataNodes) but scale compute by growing/shrinking a VM by adding/removing virtual CPU and memory resources. This would require close coordination among the hypervisor, guest OS, and Hadoop and introduces some complexities that are beyond the scope of this paper. Another potential solution is to separate Hadoop's compute layer from its data layer. This is a more natural solution, because Hadoop inherently maintains this separation via independent DataNode and TaskTracker[1] Java Virtual Machine (JVM) processes that are loosely coupled over TCP/IP.

The primary reason for having co-located DataNodes and TaskTrackers is to maintain compute-data locality. This is important in physical clusters because of network bandwidth limitations and the impact that fetching large amounts of data from a remote node would have on job performance [2]. If we naively placed a DataNode and TaskTracker in separate VMs, we would still be constrained by these limits because data transfers between them would most likely need to traverse the physical network between the hosts in the cluster. But virtual networking between VMs on the same host is free of physical link limitations (because packets travel through memory), and throughput is purely a function of CPU and memory speed. For example, with two Intel Xeon @ 2.80GHz processors running VMware ESX® 4.1, Linux VM-to-VM throughput approaches 27Gbps [13], which is nearly three times the rate supported by the 10Gbps network cards available on the market today.

This fast intrahost networking—coupled with the work being done on virtual topology awareness for Hadoop [11] (making it aware of physical vs. virtual host locality)—opens up the prospect of being able to successfully split compute and data into separate virtual nodes if they are hosted on the same physical machine. An added benefit of this approach is that the operational model is firmly established. It is a widely accepted technique in distributed application architectures in the cloud to add/remove VM instances as a way of scaling the workload.

Note that requiring a hard VM-VM affinity between the DataNode VM and TaskTracker VM would mean that it would not be possible to use VMware vSphere® vMotion® to migrate a compute VM (independently of the DataNode and its associated data) from one host to another. Although vSphere vMotion is a key benefit of virtualization, it is not a necessary feature here. If a compute VM were powered off, Hadoop would automatically restart the terminated jobs elsewhere on the virtual cluster with no effect on service uptime.

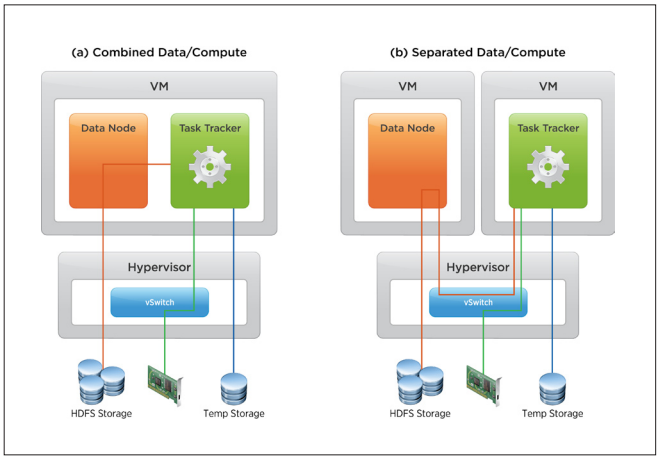


Figure 1. Combined vs. Separated HDFS Data Path

4. HDFS I/O

Figure 1 contrasts the data path that I/O traverses between the combined and separated data/compute architectures. The orange path represents HDFS I/O to the DataNode on the host, and the blue path is temporary I/O. The green path represents I/O destined to another host that must traverse the physical network. In the combined setup (Figure 1a) data packets transfer between the DataNode and a task over a loopback interface within the guest OS. But when JVMs are separated into two different virtual machines (Figure 1b), packets that are sent by Hadoop pass through many more layers before they get to their destination. First, the networking stack in the guest OS processes the message. After the required headers have been added to the packet, it is sent to the device driver in the VM. After the virtual device receives the packet, it is passed on to the virtual I/O stack in ESX for additional processing, where the destination of the packet is also determined. If the packet is destined for a VM on the same host, it is forwarded over the virtual switch (vSwitch) to that VM (orange path in Figure 1b) and not sent on the wire. All other packets are sent to the physical network interface card's (NIC's) device driver in ESX to be sent out on the physical network (green path).

4.1 Experimental Results

To evaluate the impact of separating the Hadoop DataNode from the TaskTracker, we conducted a series of experiments on a single physical node with the following configuration: 8 cores, 96GB memory with 16 disks. The HDFS replication factor was set to 2. We ran the TestDFSIO and TeraSort benchmarks. TestDFSIO is an HDFS I/O saturation benchmark and represents a worst-case scenario in terms of throughput requirements. TeraSort is a more balanced workload with a compute component in addition to HDFS I/O. The results of the experiments are plotted in Figure 2.

1 The TaskTracker daemon is responsible for spawning the task JVMs that run the compute jobs.

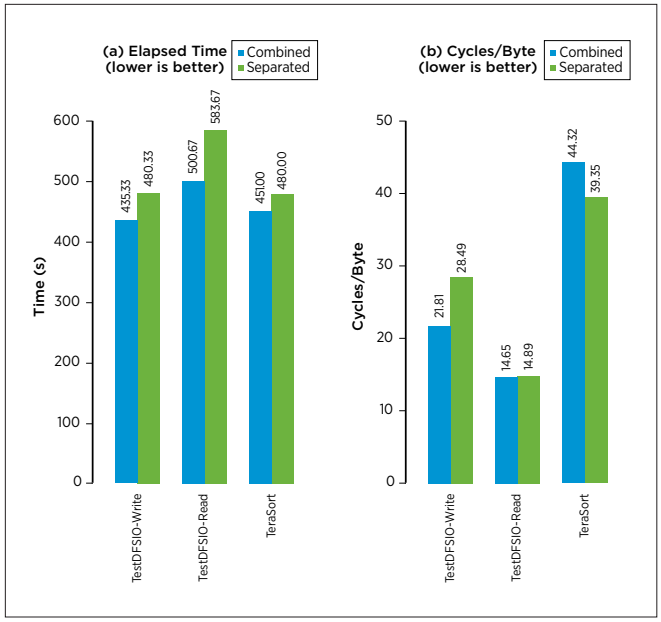


Figure 2. TestDFSIO and TeraSort Performance

Compared to the combined case, separating the DataNode from the TaskTracker causes a 10% and 17% slowdown in TestDFSIO write and read tests respectively, whereas TeraSort regresses by only 6%.

Moreover, looking at overall efficiency (as measured by the total CPU cycles executed per byte of storage I/O), TestDFSIO-Write is affected the most, with a 31% increase in cycles/byte. TestDFSIO-Read is impacted by 2%. The cycles/byte for TeraSort reduces by 11%, which is unexpected. We have yet to determine the root cause of this increase in efficiency.

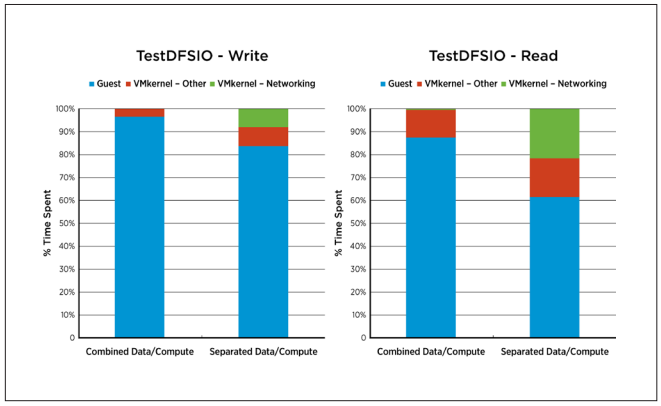


Figure 3. TestDFSIO Breakdown

Figure 3 shows the overhead that is directly attributable to networking, introduced by separation for the TestDFSIO benchmark. Although there is no network activity in the VMkernel (hypervisor) when the DataNode and the TaskTracker are in the same VM, splitting them apart results in 8% and 22% of the total time being consumed by the VMkernel networking stack in the write and read tests respectively. Our measurements have shown that for TeraSort the networking overhead is only about 2–3% and is thus less affected by separation.

In an effort to optimize the network and protocol overhead, we implemented a more efficient VM-to-VM communication channel and evaluated its performance against TCP/IP networking.

4.2 VM-to-VM Channel Implementation

Communication between VMs is via VMware virtual machine communications interface (VMCI) sockets [12] (sometimes called vSockets and also abbreviated as vsock). vSockets presents a POSIX socket API that enables client applications to create networkless channels between VMs on the same physical host. (Note that this applies only to specific versions of VMware ESXi™). It is exposed to user and kernel mode through a new socket family, AF_VSOCK. Addressing is via a specialized structure, `struct sockaddr_vm`, through which clients can specify contexts—VM identifiers, analogous to IP addresses—and ports. Otherwise, the standard socket calls are unchanged, and the semantics are as for UDP and TCP sockets.

Rather than modify the Hadoop stack to use vSockets addressing, we built an interposer library to hook the socket system calls via the `LD_PRELOAD` mechanism. A one-to-one mapping between IP and context is trivially constructed for each VM, and the library uses this mapping to translate between INET and vSocket addresses.

The underlying channel of communication for each vSocket is a shared ring buffer or queue into which data can be enqueued or from which it can be dequeued. Queues are exposed to the guest as primitives via a paravirtual device (VMCI) that also offers mechanisms for signalling the hypervisor and receiving interrupts. Each queue has a header page, with a head and tail pointer, and a number of data pages. Each socket has two such queues to allow bidirectional transfers. Enqueuing involves a copy from the client-supplied buffer (e.g., when Hadoop performs a `send(2)` socket call) and an update of the pointers on the header page. Dequeuing does the reverse, copying out of the queue into the client-supplied buffer. A simple form of flow control, based on queue fullness, is used to determine when to signal the peer that data or space is available in the queue.

Intuitively, queues should be established directly between endpoints for optimal performance. The pages of a queue between two VMs should be shared between those VMs, thereby allowing each side to read and write directly into its peer's memory. Such direct sharing, however, breaks a fundamental tenet of virtualization: isolation. It also prohibits the use of vSphere vMotion, unless VMs are teamed and migrated together. We therefore chose to implement these queues in pairs, with the hypervisor acting as a proxy between them.

A queue is constructed between VM and hypervisor for each endpoint in a channel. For instance, to connect a socket between VMs A and B, one queue is established between A and the hypervisor, and a second queue between B and the hypervisor. Each queue is backed only by the pages of its owning VM. The queues are connected transparently by the hypervisor, which maps the pages backing both queues into the hypervisor's own address space. Isolation is thus ensured. This mapping is performed once at connection time and is permanent, to avoid paying the cost for each data transfer.

A trace is installed on the header page of each queue. As data is enqueued and dequeued by each endpoint of the socket, the head and tail pointers on the queue's header page are updated, causing the trace to be fired in the hypervisor. It can then copy from one queue to another, thereby spanning the gap between the VMs. As an optimization, if the size of the transfer is less than a page, the copy is performed directly by the Virtual Machine Monitor. For larger transfers, the VMkernel performs the copy.

There are clearly some bottlenecks with our approach. There are three copies for each transfer, one of which is the result of ensuring isolation: the extra copy by the hypervisor as it proxies between the source and destination queues. Modern processors exhibit excellent copy performance, but this is still measurably slower than direct access over shared memory. Furthermore, signalling and tracing cause guest exits. Finally, a more subtle problem with the enqueue/dequeue nature of the queues is that it prevents fast transfer of mapped files, as the evaluation in the following section shows.

4.2.1 Evaluation

Figure 4 shows the results of running TestDFSIO with our optimization. As can be seen, elapsed time is on par with the combined mode for the write test and is 4% slower for reads. Although read efficiency did not show a problem to begin with, overall write efficiency significantly improved to being only 8% worse than the combined baseline.

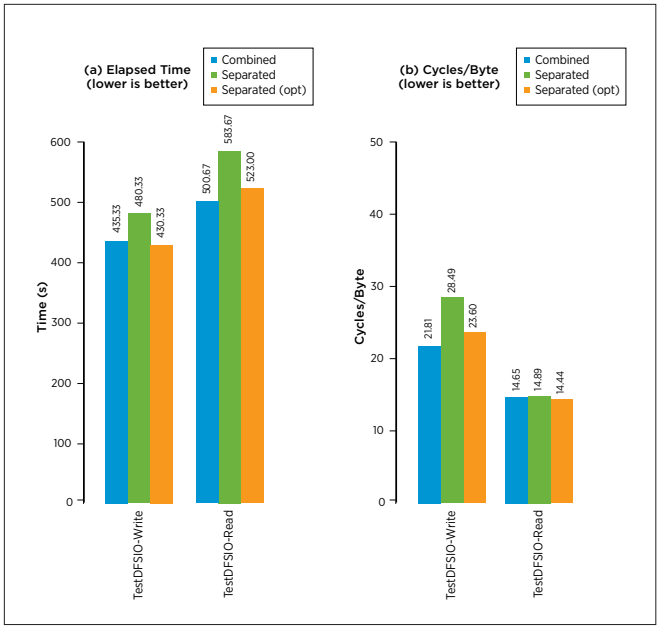


Figure 4. Optimized TestDFSIO Performance

In an effort to understand why TestDFSIO-read was still slower than in the combined case, we profiled the guest OS using `perf(1)`, the result of which is displayed as a flame graph [1] in Figure 5.

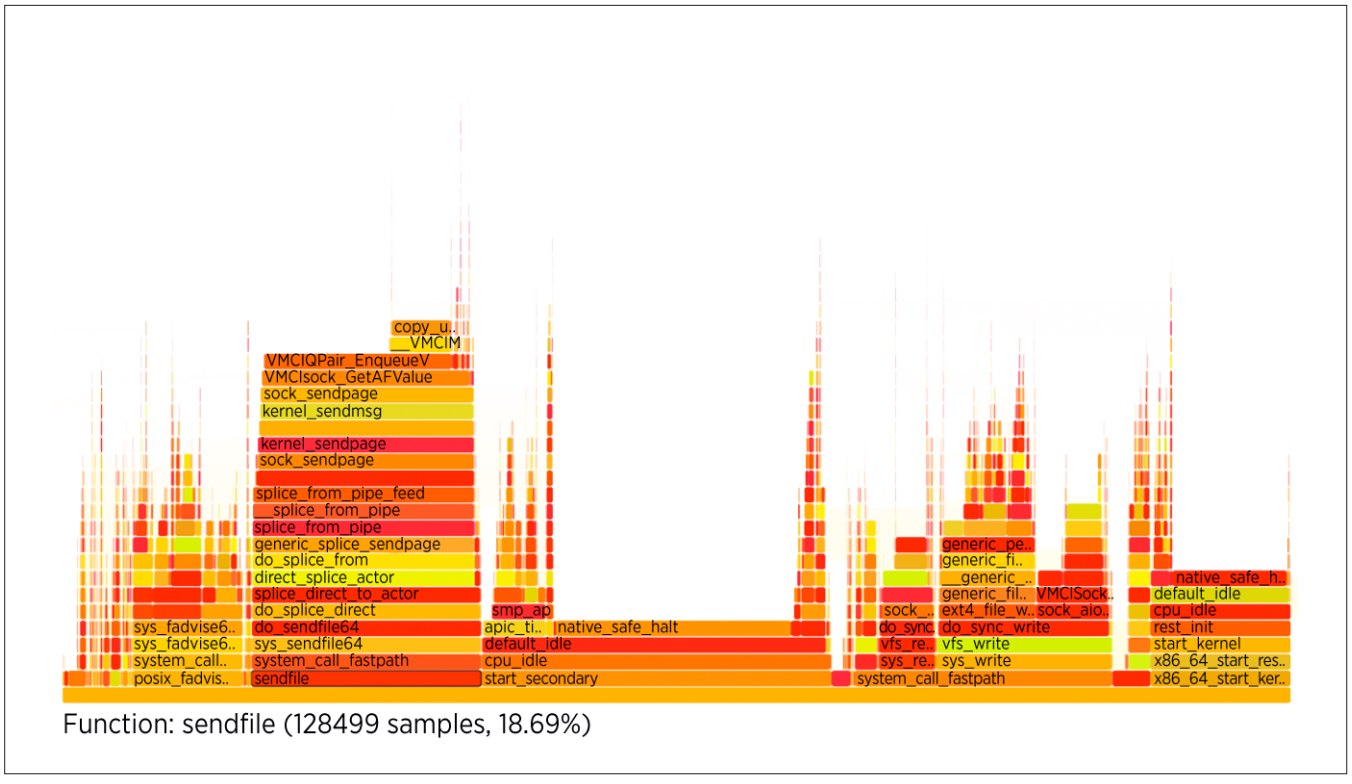


Figure 5. Flame Graph of Guest-OS Profile

The stack to the left of the graph for the `sendfile(2)` call is of most interest due to the number of samples it contains and its stack depth. It accounts for just under 19% of the total time in the profile, and it involves vSockets.

The purpose of `sendfile(2)` is to copy data between two file descriptors (either one of which can be a socket) directly within the kernel. Contrast this with doing separate `read(2)` and `write(2)` calls from user mode, which would require a copy out of kernel into the user buffer for the read and a second copy into the kernel for the write. Performing this in the kernel should avoid at least one of the copies.

The standard INET socket family supports this optimization. For example, when one descriptor is a file and the other a socket, INET can avoid copying the contents of the file into the socket's buffers. Instead, it constructs descriptor lists from the pages backing the file and passes these directly to the lower layers. Because Hadoop transfers large file chunks (64MB), avoiding the copy is a worthwhile optimization.

vSockets is currently unable to implement this optimization. It is bound by its protocol, which relies on the data being present directly in the queue itself. This means that for a file-to-socket transfer—even with the file's pages already visible to the kernel—it must copy the contents of those pages into the queue and vice versa. The upper half of the stack illustrates this. The kernel tries to call `sendpage()`, but vSockets implements no such operation, leading to `no_sendpage()`, which falls back to a regular `sock_sendmsg()`. The transfer is performed as if the separate `read(2)` and `write(2)` calls were made. In future work we plan to investigate supporting `sendfile(2)` and believe that this is all that stands in the way of achieving parity with the combined case.

5. Temporary I/O

Apart from the reads/writes of persistent data to HDFS, over the course of a Hadoop job each task might require temporary storage for saving any intermediate results. The amount of temporary space needed is dependent on the job being executed, and although some jobs might not need any temporary space at all (e.g., compute-only jobs), some could require a constant amount of space, whereas others could require a multiple of the input dataset. As an example, the rule of thumb for TeraSort is that it needs temporary storage at least twice the amount of the HDFS data it operates on, whereas a benchmark such as Pi [5] (a MapReduce program that, as its name suggests, estimates Pi) doesn't use any temporary storage. There is no reliable way to tell a priori how much space will be needed.

The storage allocated during Hadoop cluster deployment can be shared between HDFS and temporary data (temp), or they can each be stored on dedicated drives. Although the Hadoop framework doesn't impose any architectural constraints, using shared local storage space is typically preferred in order to reduce the need to overprovision storage for temp (which leads to storage inefficiencies).

In a native environment, where compute and data nodes are combined, having HDFS and temp on the same ext3/ext4 file system typically does this. The file system is then responsible for allocating and reclaiming storage blocks as files are created and deleted, thereby efficiently sharing space between the two types of Hadoop data files.

However, when the compute nodes are separated from the data nodes in a virtual environment, the most natural way to provide the tasks running in the compute VM with temporary storage space is to attach virtual machine disks (VMDKs) directly to the VM. This is ideal from a performance perspective, because the VMs have direct access to storage via the VMkernel vSCSI layer. However, it suffers from the disadvantage of needing to preallocate storage resources for both temp and HDFS, as mentioned above. The problem is more acute in designing an elastic Hadoop system, for two reasons:

- Although just-in-time provisioning of a new VM (which involves boot disk cloning and vCPU and memory allocation) can be relatively quick, provisioning independent VMDKs for temp is more challenging. We can rapidly provision space by using a sparse disk format, but that comes at the cost of runtime performance. Provisioning a thick disk format, on the other hand—although time-consuming (and antithetical to rapid, on-demand response)—would provide better runtime performance.
- Alternatively, we could preprovision VMs and power them on/off based on demand. The drawback here is that in the powered-off state the VMs are unnecessarily consuming storage space.

5.1 Using NFS Shares for Temporary Storage

An alternative to locally attached storage is to have the DataNode export NFS shares that are mounted by the compute VMs for temp space. This approach has a few advantages worth noting: (a) NFS is a mature technology, and because networking is over the vSwitch, it is not restricted by physical link limitations as mentioned previously; (b) because the DataNode hosts both HDFS and temp, storage provisioning is similar to that of the combined mode whereby the same ext file system hosts and manages the files. However, one of the main disadvantages of using NFS shared storage is the additional networking and remote procedure call (RPC) overhead associated with it.

Figure 1 shows the data path (in blue) when a local disk is attached directly to the compute VM, (Note that the figure shows both the combined and separated cases using local, directly attached temp storage.) When NFS is used to access temp storage from the DataNode, the data path is similar to that of HDFS traffic (shown in green), except that an NFS client and NFS server replace the TaskTracker and DataNode respectively.

Due to the additional layers the data needs to cross, the convenience of NFS shared temporary storage is expected to come at a higher CPU cost than accessing local storage. The following sections evaluate the overhead associated with NFS and introduce a new, extremely efficient solution for creating a temporary storage pool with the same functionality as the NFS solution, but with significantly better performance.

5.1.1 NFS Evaluation

The performance overhead of having Hadoop's mappers and reducers access a shared pool of temporary data over NFS was first evaluated in a small Hadoop cluster deployment under no resource limitations using the test bed configuration in Table 1.

TEST BED CONFIGURATION
Hardware configuration: Server: Dell PowerEdge C2100 with 2x6-core Intel Xeon CPU X5670@2.93 GHz, hyperthreading enabled and 148GB memory. Storage: LSI MegaRAID SAS 9260-8i, default controller cache configuration (WB with BBU) containing 24 local 550GB, 15K RPM SAS drive VMware vSphere: Version 5.1
Hadoop cluster configuration: Cloudera cdh3u4 (hdfs2, mr1) distribution Three-node Hadoop virtual cluster: • 1 master VM - 6 vCPUs, 40GB RAM, 8 146GB VMDKs (Eagerzeroedthick) to host HDFS data, Ubuntu 12.04 LTS • 2 worker VMs – 6 vCPUs, 20GB RAM, 3 – 146 VMDKs (Eagerzeroedthick) to host temporary data, Ubuntu 12.04 LTS

Table 1. Test Bed Configuration Used to Evaluate NFS Shared Temporary Space

The entire Hadoop virtual cluster ran on a single ESX host and was deployed in a data/compute separated configuration. Independent storage pools were used to host HDFS and temporary storage, and the entire storage pool was allocated to the master VM. The master VM ran the Hadoop NameNode and DataNode and exported shares over NFS to serve as temporary storage to the worker VMs. One of the two worker VMs ran the JobTracker and TaskTracker while the other ran only the TaskTracker. Both accessed HDFS and temporary data over a private subnet and a vSwitch exclusively dedicated to handle Hadoop's throughput-intensive traffic. Because the entire cluster was running on a single host, we were able to leverage the high throughput and low latency provided by the vSwitch.

A single test consisted of a TeraGen, TeraSort, and TeraValidate sequence. TeraSort uses a significant amount of temporary space, which makes it an ideal candidate for evaluating the performance of any temp storage solution. TeraGen and TeraValidate, in contrast, hardly use any temporary space at all. We therefore focus our analysis on the TeraSort results.

The graph in Figure 6 highlights the difference in elapsed time for the TeraSort benchmark using temp storage local to the VM from using an NFS-mounted share. We also plot the CPU efficiency (measured in cycles/byte) for accessing temporary data over the two solutions. From the graph we observe that TeraSort performance (elapsed time) is about 7% worse with NFS. Although this result is encouraging, the

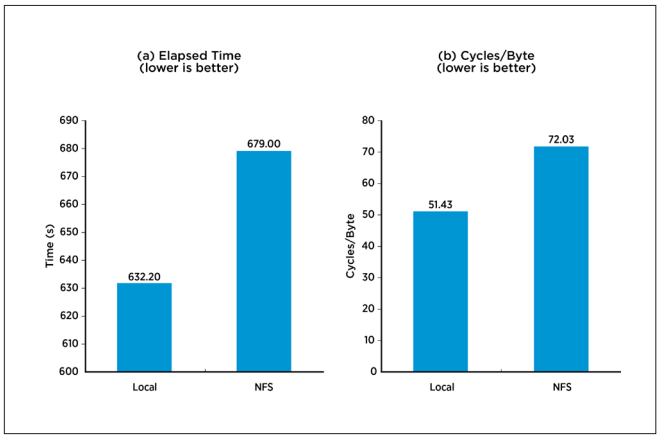


Figure 6. TeraSort Performance Comparison of Local vs. NFS Mounted Temp

40% increase in CPU cost is of concern. This is especially so for CPU-intensive applications for which saturating the CPU could hurt overall performance.

To get a handle on how much of an impact the extra CPU overhead has on job performance, we performed a variant of the above experiments wherein we fully saturated the host. To achieve saturation, the Hadoop cluster was reconfigured and the number of compute VMs was scaled up until CPU utilization was above 90% for both the local and NFS experiments.

Whereas the hardware/software configuration of the scaled test bed used in evaluating the performance of NFS-hosted temp space was similar to the one presented in Table 1, the virtual Hadoop cluster configuration differed as follows: 5 vCPUs instead of 6 per VM; 4 instead of 3 worker VMs with 24GB RAM each (instead of 20GB).

As expected, on a saturated host the performance delta between local and NFS widened to approximately 17%, whereas the CPU overhead was about 4% higher with NFS. To get a better understanding of the nature of the performance difference, we took a look at the different stages in the MapReduce pipeline. From the Swimlanes plot [10] in Figure 7 we observe that the "Shuffle and Sort" phases take longer with NFS. These are the two execution phases of a MapReduce application, where the intermediate results are generated by the map tasks, sorted, and dispatched to the reducers and where temporary storage is utilized the most. One contribution to the slowdown over NFS is the added overhead introduced by the TCP/IP stack and the more layers (compared to local access) that I/O must traverse, as discussed in Section 3.

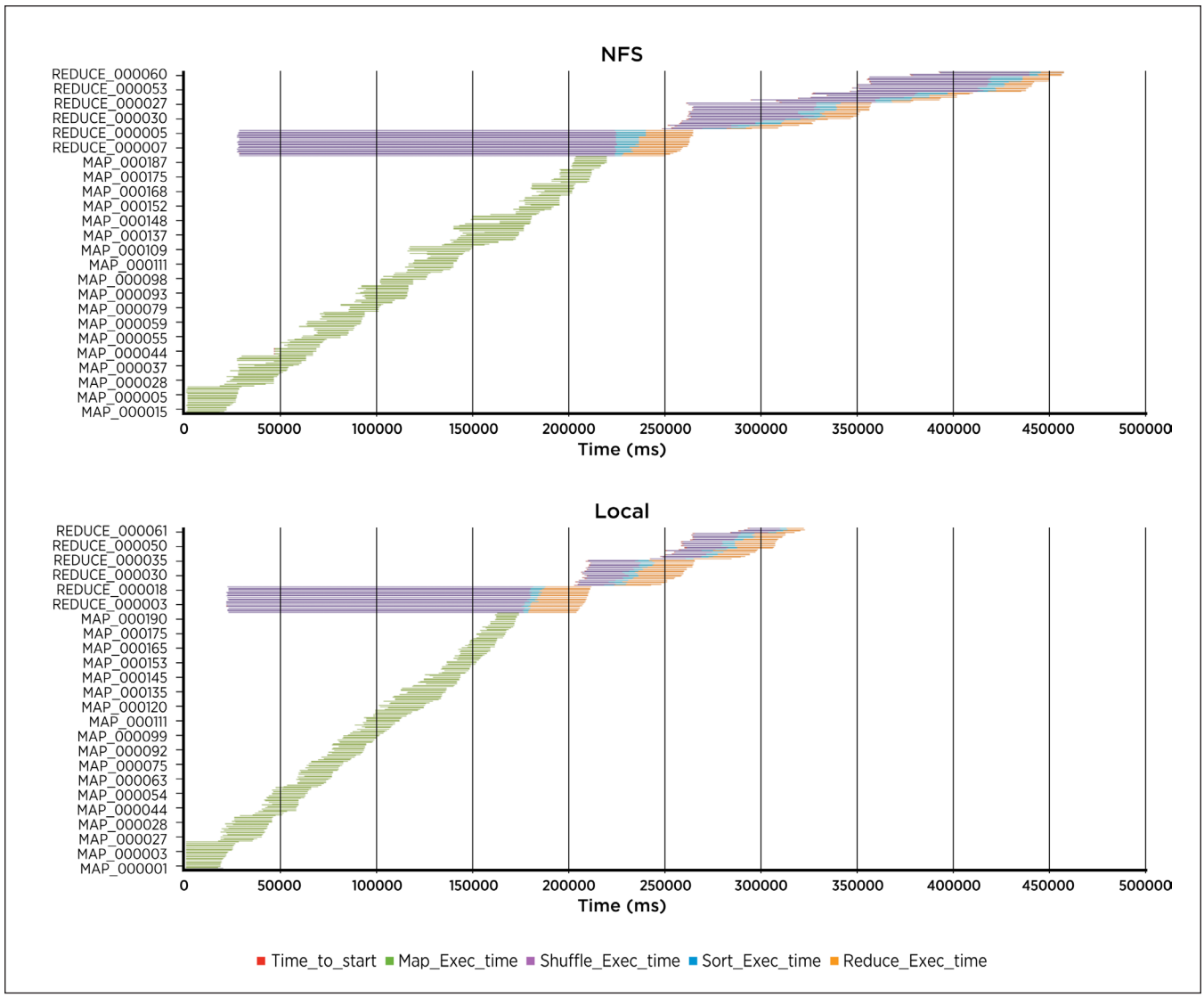


Figure 7. Swimlanes MapReduce Phase Analysis

However, in an attempt to gain more insight we extended our analysis to profiling Java and the guest OS. Using HPROF [3] to profile the Hadoop JVMs, we found that with NFS, 12% of the total execution time of the map phase was spent in `java.io.FileOutputStream.close()` compared to 0.02% in the local case. Similarly, in the reduce phase that same method accounts for 7.5% of the total execution time under NFS compared to 0.04% with local VMDKs. Further analysis revealed that `java.io.FileOutputStream.close()` ultimately results in a call to `nfs_file_flush()` in the guest kernel. To maintain consistency, data cached by the NFS client is flushed (via `nfs_file_flush()`) over the network to the NFS server whenever an application closes a remote file. Ninety percent of the files accessed by the TeraSort job were hosted on NFS and as such, these cache flushes had a significant effect on performance.

An ideal solution to the "temp problem" would be a mix of local storage and NFS with the following attributes:

- Performance and efficiency at least on par with local, direct attached VMDKs
- A single pool of shared storage, provisioned and managed independently of the VMs
- Storage isolation between VMs
- Space allocated on demand and reclaimed when no longer needed

In the next section we present a solution that we call *Elastic TempFS* with these characteristics.

5.2 Elastic TempFS Implementation

Elastic TempFS is implemented as a vSCSI filter module to redirect all accesses to virtual disks storing temp data in compute VMs to a common "pool" file hosted on vSphere VMFS. We use SCSI UNMAP [8] commands from the guest to signal when files are removed, which allows the TempFS module to deallocate space in the pool file whenever temp files are removed.

The basic configuration is that each compute VM has one or more "temp" virtual disks whose accesses go through the vSCSI filter. The vmx file of the compute VM specifies which pool file will back the accesses of each temp virtual disk. The semantics is that the

contents of the temp virtual disks are zeroed out whenever the associated VM is booted. This temp semantics means that the vSCSI filter does not need to persist any of its mapping information to disk (unless needed for overflow), because all temp virtual disks are zeroed out if the entire physical host crashes.

For a particular temp virtual disk, the vSCSI filter tracks all writes to the virtual disk and maintains a list of all contiguous extents that have been written so far. Each time a new extent of the temp virtual disk is written, the vSCSI filter automatically chooses a location in the pool file at which this extent can be stored. The vSCSI filter maintains this mapping largely in memory (possibly overflowing to disk as necessary), so all later accesses to the same extent can be efficiently mapped to the right location in the pool file. All extents associated with a particular VM are automatically deallocated from the pool file whenever the VM is powered off.

In addition, the compute VMs are preferably configured so that they can send down TRIM/UNMAP information to the hypervisor when files are removed from the temp virtual disk. In particular, this can be done easily if ext4 is used as the file system for the temp virtual disks, and the Linux kernel version is 2.6.39 or higher. In this case, any time an ext4 file is removed, an appropriate TRIM command is sent down to the hypervisor. Then the vSCSI filter can deallocate that extent from the pool file, allowing the pool space to be used much more dynamically. Clearly, because temp virtual disks are zeroed on every VM boot, the guest must be configured to properly remake the ext4 temp file systems on each boot. This is a relatively cheap operation if ext4 is used with the proper choice of parameters [4].

ext4 does a good job allocating large files in a small number of (contiguous) extents and also of allocating all small files in somewhat contiguous extents. So—especially for Hadoop workloads (which usually create large temp files)—the vSCSI filter can maintain extent information with a fairly small list of extents.

The large file writes in Hadoop (HDFS and temp) at the virtual SCSI level are 512KB (the largest possible for SCSI). So, we were able to implement a fairly simple allocation policy: For a new extent that is started with a 256K write or larger, we allocate a physical 16MB region, because that extent is likely for a large file and we want to make sure it is contiguous at least in 16MB chunks. For extents started with smaller writes, we can just allocate in a "small file" region in any reasonable way (fragmentation is acceptable). Although there could be more-complicated policies that maintain multiple open regions in the pool file for allocating virtual writes contiguously, we did not investigate them.

5.2.1 Evaluation

To study the performance of TempFS, we used a test bed configuration identical to the one presented in Table 1 on vSphere 5.0, and we followed an experimental methodology similar to the one described in Section 5.1.1.

The graph in Figure 8 highlights the performance difference expressed in TeraSort elapsed time between local and TempFS temporary space while evaluating the extra CPU overhead expressed in cycles/byte for accessing temporary data over TempFS. The data in the graph represents averages over the three executions of the test application.

The results in Figure 8 show that the performance of Elastic TempFS is on par with local temp in terms of both TeraSort application elapsed time and CPU cost.

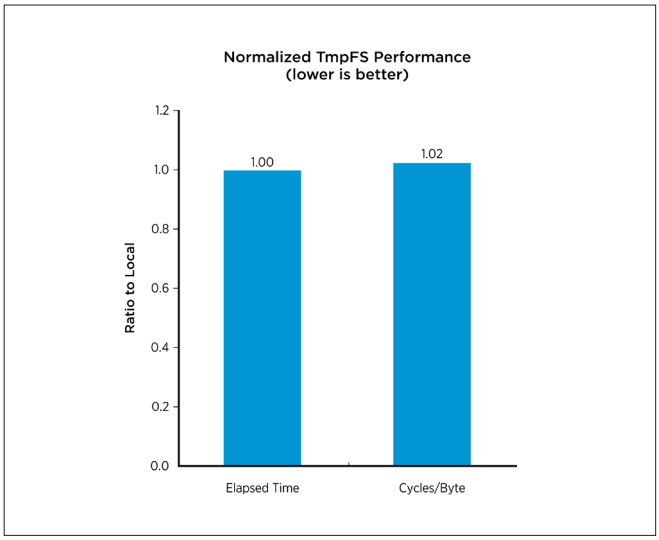


Figure 8. TeraSort Performance Comparison of TempFS Relative to Local, Direct Attached VMDKs for Temp Storage

In summary, TempFS provides us with all the advantages of NFS, but without any of its associated overhead.

6. Conclusion

In this paper we explored the challenges associated with turning Hadoop into a flexible and scalable elastic service and discussed the need to separate the compute and data layers. Whereas traditional methods of separation come at the cost of performance and efficiency, our optimizations and analysis showed that we could utilize intrinsic features of the virtualization platform to achieve the performance levels of the combined case.

In particular, we presented the use of vSockets for intrahost, inter-VM communication, whereby we were able to optimize HDFS I/O performance to be on par with the combined case for the TestDFSIO-Write benchmark and to within 4% for TestDFSIO-Read. Our analysis showed that the remaining overhead for reads is due to the fact that we did not support the `sendfile(2)` system call, and we believe that this is all that stands in the way of achieving parity.

In addition, we presented a new technique—Elastic TempFS, designed to meet the storage needs for temporary MapReduce data—that has the desired attributes of shared, pooled storage and high performance. Although NFS provided for pooled storage with a moderate performance overhead, in our experiments the associated CPU cost was as high as 40%. Elastic TempFS, in contrast, introduced no significant overhead and is equivalent to direct VM-attached, vSCSI storage in terms of both performance and efficiency.

Acknowledgments

We would like to thank Jayanth Gummaraju and the rest of the VMware Big Data team for their contributions as well as Reza Taheri, Jeffrey Buell, and Priti Mishra for their valuable feedback and suggestions.

References

1 Gregg, B. Flame Graphs. http://dtrace.org/blogs/brendan/2011/12/16/flame-graphs/

2 Guo Z, Fox G, Zhou M. Investigation of data locality and fairness in MapReduce. In: MapReduce '12. Proceedings of Third International Workshop on MapReduce and its Applications, 2012, pp. 25–32.

3 HPROF: A Heap/CPU Profiling Tool. Oracle Corp. http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html

4 mkfs.ext4(8) - Linux man page. http://linux.die.net/man/8/mkfs.ext4

5 Package org.apache.hadoop.examples.pi Description. http://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/pi/package-summary.html

6 perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page

7 Porter, G. Decoupling Storage and Computation in Hadoop with SuperDataNodes. In: ACM LADIS '09. Third ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, 2009.

8 SCSI Commands Reference Manual. Seagate Technology LLC; 2010, pp. 211–214. http://www.seagate.com/staticfiles/support/disc/manuals/Interface%20manuals/100293068c.pdf

9 Shafer, J. A Storage Architecture for Data-Intensive Computing. PhD Thesis, Rice University, 2010

10 Tan J, Pan X, Kavulya S, Rajeev G, Narasimhan P. Mochi: Visual log-analysis based tools for debugging Hadoop. In: HotCloud'09. Proceedings of the 2009 Conference on Hot Topics in Cloud Computing; 2009. Article No. 18.

11 Umbrella of enhancements to support different failure and locality topologies. The Apache Software Foundation. https://issues.apache.org/jira/browse/HADOOP-8468

12 VMCI Sockets Programming Guide. VMware Inc.; 2008. http://www.vmware.com/pdf/ws65_s2_vmci_sockets.pdf

13 VMware vSphere 4.1 Networking Performance. VMware Inc., 2009. http://www.vmware.com/files/pdf/techpaper/Performance-Networking-vSphere4-1-WP.pdf

14 White, T. *Hadoop: The Definitive Guide*. Sebastopol, CA.: O'Reilly Media, 2009. pp. 293–396.

vSDF: A Unified Service Discovery Framework

Disney Y. Lam
University of Waterloo
y7lam@uwaterloo.ca

Konstantin Naryshkin
VMware Inc.
knaryshkin@vmware.com

Jim Yang
VMware Inc.
jimyang@vmware.com

Abstract

The use of automatic and self-adjusting service discovery mechanisms can improve availability in service-oriented infrastructures. However, many service discovery protocols cannot be deployed because they require the use of broadcast and multicast traffic, which are prohibited in many networking environments. As a result, many platforms still advertise service information by static means. In this paper, we propose a new service discovery framework, vSDF (VMware Service Discovery Framework). It is a highly tunable service discovery framework that can be used in broadcast, multicast, and even unicast networking environments. In our experimentation, it was found that operating service discovery via unicast means could yield search coverage on a par with multicast and broadcast discovery modes. Moreover, vSDF simplifies service registration for service providers and service search for service requesters. It is also capable of coordinating service registration when multiple service registries exist. On the whole, vSDF is a framework that can improve not only service availability, but also configuration management and code reusability.

1. Introduction

In traditional distributed systems with many service providers and service requesters, a central service registry is commonly deployed, so that service requesters need to know only the location of the service registry in order to access services. In this model, service providers register their services to the service registry, and service requesters query the service registry for these services. Upon a successful query, a service requester can communicate with the service provider hosting the desired service using the information provided by the service registry.

Although this approach can reduce the amount of information service requesters need to know before being able to locate a service, it still requires that service requesters explicitly know the location of the service registry where the service being sought is registered. Moreover, service providers are burdened with knowing the location of the service registry so that they can register their own services and have them be advertised to service requesters.

In most cases, the location of the service registry is provided statically to service providers and requesters in the environment, usually inputted manually by a systems administrator or via an installer script. However, this can lead to reduced service availability, because service requesters are unable to find new services if the service registry changes its location in the network. Furthermore, a service might appear to be unavailable if a service registry cannot remove stale service entries from its database and then provides that incorrect location information to a service requester. Keeping this configuration up to date becomes a burden for personnel managing the system.

In an attempt to solve these problems, many service discovery protocols, such as [5], [9], [13], and [14], have been proposed and implemented. Despite their effectiveness in enabling dynamic discovery of services and service registries, these protocols are unpopular because they primarily use multicast and broadcast traffic [4]. The use of these traffic types can make a distributed environment more susceptible to passive attacks and vulnerable to security breaches overall [2]. Additionally, considerable software integration is needed to deploy these protocols properly.

In this paper, we present vSDF (VMware Service Discovery Framework), an extension and enhancement of Internet Engineering Task Force (IETF) Service Location Protocol (SLP) Version 2. It is a service discovery framework that is a unification of neighbor discovery, random discovery, broadcast discovery, and multicast discovery methods. Due to this amalgamation, vSDF is capable of operating not only in broadcast and multicast enabled environments, but also in networks restricted to unicast traffic. vSDF increases service availability through the use of leases. This mechanism works by requiring services to be periodically reregistered with designated service registries while having service registries systematically remove stale entries. To make the process of service discovery in unicast-based discovery modes more efficient, we added a collaboration feature, so that participants can send clues about where other vSDF agents might be located. We provide APIs so that service providers can easily publish their services and have service agents register services on their behalf. Overall, the use of vSDF can increase service availability in service-oriented infrastructures, reduce configuration-management overhead for systems administrators, and increase code reusability for systems developers.

The software design of vSDF enabled us to quantitatively compare various discovery modes and their search performance. In our experimentation, we found that even with no collaboration among agents, unicast discovery mode achieved—via breadth-first search of the neighbor graph with a limited hop count—search coverage close to that of broadcast and multicast discovery. When collaboration is enabled on vSDF agents, unicast discovery achieved by randomly contacting other potential agents was able to find 24% more agents than without collaboration.

Moreover, both random and neighbor unicast discovery modes were able to achieve search performance similar to that of broadcast and multicast discovery modes.

The contributions of this paper are threefold. First, we designed and implemented a framework that unifies service discovery methodologies found in academic papers, network device discovery mechanisms, and service discovery protocols, with many other tunable features. Second, we provided the proper APIs for service publishing so the framework can be easily integrated into many service-oriented infrastructures. Third, we provide experimental results that will enable engineers to select the best service discovery methods and modes based on quantitative results, rather than qualitative results that have been provided in most other publications, such as [1], [4], and [11].

The rest of our paper is structured as follows. First, we introduce a number of works that are relevant to service discovery. Then, we present the design of vSDF. Next, we provide experimental results to demonstrate the capabilities of the framework. This is followed by a discussion of security and configuration concerns. Finally, we describe future work and conclude.

2. Related Works

Four main families of discovery protocols pertain to service discovery in networking environments: neighbor discovery, random discovery, broadcast discovery, and multicast discovery. Although neighbor discovery and random discovery are typically used for network topology discovery rather than service discovery, they can be easily adapted to perform service discovery and are thus discussed as well.

2.1 Neighbor Discovery

Neighbor discovery protocols (a.k.a. one-hop discovery protocols) are typically used to discover connected network devices at the data link layer.

In general, multicast data link layer protocol data units are sent out to every connected port of a device to request neighbor information. Some examples of neighbor discovery protocols include Cisco Discovery Protocol (CDP) and Link Layer Discovery Protocol – Media Endpoint Discovery (LLDP-MED). In CDP, data link layer multicast frames are sent and received by switches to retrieve platform information, device capabilities, and address information of neighbors [7]. In LLDP-MED, the protocol is used by voice-over-IP (VoIP) phones to obtain the VLAN that it is supposed to belong to, as opposed to using Dynamic Host Configuration Protocol (DHCP) [7].

One-hop discovery protocols can be easily extended to multihop discovery protocols via depth-first or breadth-first search—that is, having a neighbor ask its neighbors for information and so forth. This technique is used by network device collectors to construct a network's topology and is commonly used in industry.

Currently, conventional neighbor discovery features are not available for discovering neighboring hosts from a host. However, in IPv6, Neighbor Discovery Protocol (NDP) will be usable for both networking devices and hosts to discover connected neighbors [12].

2.2 Random Discovery

Random discovery methods have been proposed by academics to solve problems in many areas, ranging from biology to finance to computer science. These algorithms typically involve use of random walks—walks in which a neighbor forwards a message received to another neighbor at random, to discover an underlying graph.

In computer networking, random walks have been shown to be capable of topology discovery via unicast transport. However, this topology discovery method performs worse than broadcast and multicast discovery protocols in terms of time, mainly due to lack of intelligence in choosing neighbors to follow. In [3], it is shown that the search performance of topology discovery can be improved through the use of biased random walks. (i.e., a distance metric is used to choose a neighbor to continue the walk).

2.3 Broadcast Discovery

Broadcast discovery discovers services in a bounded area via the use of broadcast messages. In this discovery mechanism, messages are sent on a one-to-all basis. It is commonly used in ad-hoc mobile networks.

Bluetooth Service Discovery Protocol (SDP) is an example of a broadcast-based service discovery protocol. Bluetooth SDP enables mobile services on devices to be discovered in Personal Area Networks (PANs). On Bluetooth-enabled devices, both a SDP Server and SDP Client are simultaneously executed. Service applications register their services to SDP Servers. These SDP Servers keep a database of services that their hosting devices can provide and broadcast this information to other SDP Servers periodically. Essentially, SDP Servers within a PAN are vertically replicated service databases. When a client application on a device seeks a service, a SDP Client is used to locate the service desired. The SDP Client broadcasts a discovery message that is responded to with a service reply from one or more SDP Servers [13].

To enable broadcast discovery protocols in wired LANs, network devices must enable routing of IP directed broadcasts. By default, this is a feature that is disabled on most network devices [10].

2.4 Multicast Discovery

Many multicast-based service discovery protocols exist. Most of these protocols operate in a similar manner: They send probes to all hosts in the same Internet Group Management Protocol (IGMP) group to discover a target. To provide adequate breadth, we survey three protocols: Domain Name System – Service Discovery (DNS-SD) using Multicast Domain Name System (mDNS), Simple Service Discovery Protocol (SSDP) in Universal Plug and Play (uPnP), and SLP (Service Location Protocol).

2.4.1 DNS-SD Using mDNS

Multicast Domain Name System (mDNS) is defined by RFC 6762 and is used in Apple Bonjour [6]. It uses the same messages, APIs, and operational flows as Domain Name System (DNS). However, mDNS is different in that it enables users to resolve a hostname by sending multicast messages on the local network, as opposed to contacting a dedicated DNS server. DNS-SD is defined in RFC 6763 [5]. It uses mDNS messages to discover other services in a local network.

In DNS-SD, every participant in the local network is considered a peer. Each participant keeps track of DNS records that it knows and replies to another peer if it knows how to resolve a DNS request. Because conventional DNS servers tend to be more geographically distant than peers within a local network, DNS-SD can be used to reduce propagation delay in hostname resolution. For service discovery, DNS-SD hosts keep track of services, in the form of SRV DNS entries, and interact the same way they would with regular DNS entries [6].

2.4.2 SSDP in uPnP

SSDP is the multicasting discovery protocol used in uPnP to discover printers, devices, and network gateways. SSDP follows the same APIs and operational semantics as HTTP. However, SSDP's HTTP messages are transferred over UDP. and HTTP request messages are multicast over the network. The protocol uses dedicated service registries, called control points, to keep track of services in the network. Root devices are responsible for registering services to control points. In a multicast channel, control points can make themselves discoverable by either actively advertising their location or passively waiting for a device to do a search query over the network. When a service becomes unavailable, the root device should revoke the service from the control point using a HTTP goodbye message. To handle service updates in a network, version numbers are used [14].

2.4.3 SLP

SLP is an IETF standard that is defined in RFC 2608 [9]. It is a multicast-based protocol used for discovering printers in CUPS (formerly called the Common Unix Printing System) and for seeking VMware ESXi™ servers hosting desktop virtual machines (VMs) in VMware Horizon View™ Client.

Agents in SLP are run as separate processes on a host. There are three types of agents in SLP: Directory Agent (DA), Service Agent (SA), and User Agent (UA). The two mandatory agents in SLP are SA and UA. SAs advertise service information, and UAs request service location. The DA is an optional agent in SLP that is used to aggregate service advertisements from SAs and reply to UA requests.

If no DA is present in the network, UAs must multicast service requests or receive SA advertisements in order to obtain service location information. When a DA is present, both the UA and SA communicate with the DA as proxy. In order for a DA to announce its presence in the network it either sends multicast DA advertisement messages (active DA discovery) or replies to multicast service requests by SAs and UAs (passive DA discovery). DA location can be given to SAs and UAs statically or via DHCP.

SLP increases service availability through the use of leases; DAs remove expired entries from its database, so SAs are required to reregister services back to a DA.

The protocol assumes that there is never more than one DA entity in the network. When there are multiple DAs, the RFC suggests that users of SLP can choose either vertical or horizontal replication schemes for the service database. Furthermore, it does not enforce the type of database to deploy.

3. Design

vSDF is a prototype framework written in Java. It contains approximately 6,000 lines of code. Because vSDF is an extension of SLP, many of the basic components of SLP have been retained. These attributes are explained in more detail. vSDF's design has been enhanced so that it can better support service discovery when used on a unicast basis and when multiple service registries exist. The high-level details of these features are presented in this section.

3.1 Components

Like SLP, vSDF has three types of agents: Directory Agent (DA), Service Agent (SA), and User Agent (UA). DA acts as the service registry, SA as the service registrant, and UA as the service requester. In a situation in which the participants all know one another, UAs can send SRV_RQST (service request) messages to a DA and receive SRV_RPLY (service reply) messages in reply. SAs send SRV_REG (service registration) messages to a DA and receive SRV_ACK (service acknowledgment) messages in reply. This is shown in Figure 1.

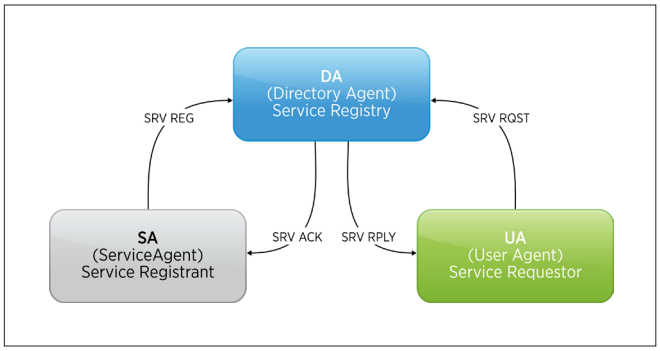


Figure 1. vSDF Components

3.2 Messages

MESSAGE TYPE	DESCRIPTION
SRV_RQST	Request for service location
SRV_RPLY	Reply with service location
SRV_REG	Service registration
SRV_DEREG	Service deregistration
SRV_ACK	Acknowledgment of service registration
SRV_NACK	Acknowledgment of service deregistration
DA_PREDICT	Prediction of a DA's location
DA_AD	Advertisement of a DA's location
DA_RQST	Request for a DA's location

Table 1. vSDF Messages

In vSDF, messages are considered to be in one of two modes: discovery or unicast. In discovery mode, messages are used specifically to find new service registries or services. The

technique that is used to transport messages in discovery mode is tunable and is explained in further detail in Section 3.6. Unicast mode messages are used to manage service registration and service lookup.

The message types are shown in Table 1. With the exception of DA_AD and DA_REQUEST, the messages are all transported in unicast mode (i.e., using unicast UDP or TCP). DA_AD and DA_REQUEST can also be transported via unicast mode but are typically used in discovery mode to facilitate discovery of agents in the network.

3.3 Agent Function Design

In general, each of the agents was designed to concurrently handle four functions. The first handler is used to receive and process unicast messages. The second does the same for discovery messages. The third is used to send out periodic advertisement messages. The fourth performs management tasks. The function of the management handler depends on the agent. For DAs, it includes checking its network configuration and removing expired entries from its database. For SAs, it includes checking for new services published, old services unpublished, and renewing services with DAs. For UAs, it includes removing expired service entries that have been found. UAs also run an additional thread that accepts service requesters' search queries.

3.4 Leases

Leases are required for service registrations in order to improve service availability. The exact numerical value of the lease is given by the service provider. DAs systematically remove expired entries from their service databases, and SAs periodically reregister expiring entries with DAs. This functionality is implemented in the management threads of the agents. Additionally, each DA that an agent finds is registered in the agent's database as a service and also has a lease.

3.5 Advertisement Modes

An agent can discover services or service registries using either passive or active advertisement mode. In vSDF, at least one of the DAs, SAs, or UAs must be in active advertisement mode in order for service discovery to succeed. The modes are described in this section.

3.5.1 Passive Advertisement Mode

In passive advertisement mode, an agent waits to receive discovery messages to learn about other agents in the network. The advertisement handler in the agent is not executed in this mode.

3.5.2 Active Advertisement Mode

In active advertisement mode, agents directly send out advertisement messages to assist with discovery. The operational semantics and advertisement frequency are discussed in this section.

3.5.2.1 Operational Semantics

The messages sent depend on the agent's role, as shown in Figure 2. For DAs, they directly send out DA_AD messages for other agents to receive. SAs and UAs send out DA_RQST messages so that they can be replied to with unicast DA_AD messages. In both cases, when a DA is found by an agent, the agent sends back a SRV_ACK.

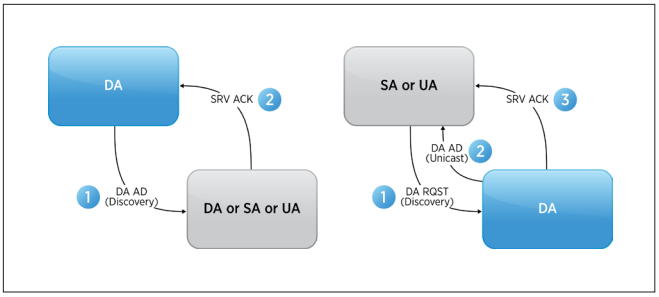


Figure 2. Active Advertisement Mode (Left: DA Active, Right: SA or UA Active)

3.5.2.2 Advertisement Frequency

Active advertisement mode has two phases—initial and continuous—corresponding to two different advertising frequencies at which an agent sends out advertisement messages. Thus, three additional parameters— `initPhase`, `initInterval`, and `continuousInterval`— must be provided to the agent. During the first `initPhase` ms, advertisements are sent out every `initInterval` ms. After initPhase ms have elapsed, advertisements are sent out every `continuousInterval` ms. An agent reverts to the initial phase depending on its role: DAs revert when the network configuration has been changed, and SAs and UAs revert when no DAs have been discovered. This is implemented to reduce unneeded overhead after a vast majority of agents have been discovered.

3.6 Discovery Modes

Discovery modes dictate how discovery messages are sent in a network so that unknown agents can be found. In vSDF, four discovery modes were implemented: broadcast discovery, multicast discovery, n-hop neighbor discovery, and random discovery.

3.6.1 Broadcast Discovery

Broadcast mode was created to closely emulate the broadcast methods used in Bluetooth SDP that are discussed in Section 2.3. In broadcast discovery mode, it is required that a list of subnets defining network boundaries be provided to the agent being executed. Additionally, network devices routing vSDF messages must be configured to allow inter-VLAN broadcast messages to be exchanged (i.e., enable IP directed broadcasts). When a discovery message needs to be sent, this mode sends the message across all the inputted subnets' broadcast addresses. Broadcast discovery mode uses UDP exclusively to transfer messages.

3.6.2 Multicast Discovery

Multicast discovery mode transfers discovery messages in a similar fashion to DNS-SD, SSDP, and SLP (mentioned in sections 2.4.1, 2.4.2, and 2.4.3 respectively). In multicast discovery mode, no additional input to the vSDF process is necessary. At runtime, the vSDF agent joins an IGMP group. When a discovery message is sent, it forwards it to all participants in the same group via UDP. In order to support service discovery across multiple VLANs on the network, a Rendezvous Point must be configured on devices that use Protocol Independent Multicast – Sparse Mode. Furthermore, to decrease network overhead, IGMP snooping should also be enabled on network devices connecting vSDF participants.

3.6.3 n-Hop Neighbor Discovery

In 2.1, neighbor discovery protocols used for network devices were briefly surveyed. These one-hop protocols can be easily extended to perform multihop service discovery. Because the aggregated neighbors list of all hosts in a network forms the topology graph, it is also capable of discovering all the services in a network. They are beneficial in networks that do not support multicast or broadcast traffic, because they can be used in unicast restricted environments over either TCP or UDP.

In vSDF, an agent requires a list of neighbors in terms of a network layer overlay as input. The agent also requires a hop limit. The hop limit, in our case, is the number of vSDF agents a discovery message can traverse before being dropped—much like the Time to Live (TTL) field in IP packets at the networking layer. When an agent receives a discovery message in n-hop neighbor discovery mode, it processes the message, decrements the hop limit, and forwards the message to its neighbors if the hop limit is nonzero.

3.6.4 Random Discovery

In section 2.2, we introduced neighbor topology discovery solutions that make use of biased random walks. In the case of service discovery, random service discovery methods can be further simplified. This is primarily due to the fact that network bounds determining the search space are well-defined for local network service discovery; that is, subnet and/or VLAN information is known. In this mode, the systems administrator provides a list of subnets to a vSDF agent as input. The agent continually sends discovery messages to random hosts in the subnets provided until a contacted host responds to a discovery message. Because this mode uses TCP as the underlying transport protocol, vSDF knows to move on to contacting the next random host when the TCP connect fails. Moreover, this mode is unicast-based and makes exclusive use of TCP.

3.7 Service Publishing

For ease of integration, two publishing interfaces are provided: `String publish(Service srv, String zone, Endpoint ep, long lease);void unpublish(String regId);srv` is the service identifier; `ep` is endpoint of the `srv`; `zoneTag` is the zone ID pertaining to which DA a service wants to register to; `lease` is the amount of time before renewing a service.

The service publishing process was designed so that when a service provider needs to register a service, it publishes it to a file using the service publishing APIs, as shown in Figure 3. The service provider receives a registration identifier that it can use later to deregister with. The management handler in a SA pulls services from the service file and registers them on the service providers' behalf.

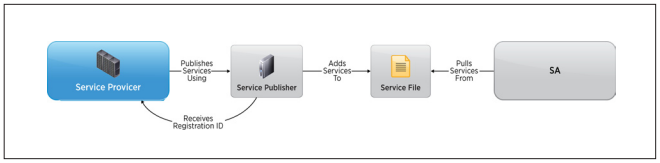


Figure 3. Service Publishing in vSDF

`zoneTag` and `lease` are optional arguments. If no `lease` is specified, then the service publisher assumes infinite lease. If no `zoneTag` is specified, then a zone will be chosen by the SA when it adds the service, as described in Section 3.8.

3.8 Coordinating Multiple DAs

To provide scalability to many services, multiple DAs can be deployed in the network. In vSDF, services can register not only with their service identifier and endpoint information, but also with a zone tag. The zone tag corresponds to a DA in the network. If no zone tag is provided, then selection of a zone tag is done by the SA registering the service. We leverage hierarchical address schemes so that the SA will select the DA that is most proximal to the service endpoint—that is, according to the maximal IP prefix match.

3.9 Collaboration

Collaboration in a vSDF agent enables it to send predictions about agents other than itself that it has knowledge of. The ability for agents to collaborate in vSDF can be very useful in discovery modes that are unicast-based.

For example, this feature is useful when an agent in unicast discovery mode contacts an agent that is not the DA sought, which is shown in Figure 4. When a DA_AD message is received on an agent, it can send back DA_PREDICTION messages about DAs that it knows about. When an agent receives a DA_PREDICT message, it sends a unicast DA_RQST message to the predicted DA location. If the prediction is correct, then the target DA sends back a unicast DA_AD message, advertising its location.

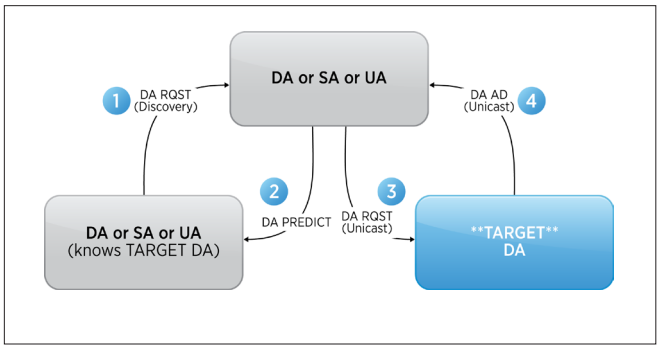


Figure 4. Collaboration in vSDF Agents

Because SAs and UAs need to know about the DAs that are on the network, they can easily send back DA_PREDICT messages. As for DAs, they need to know about other DAs in the network, in order to provide predictions. This can be achieved through passive collection of DA_AD messages from other DAs.

In vSDF, if a DA prediction is incorrect, the source is not notified because it can create unnecessary message overhead. Instead, the source knows that its prediction is incorrect because the database entry pertaining to the prediction times out and is not renewed.

4. Experimentation

vSDF's software design can be easily leveraged to demonstrate different discovery modes. In this section, we explain the test bed used and experimental results obtained.

4.1 Test Bed

To demonstrate our framework's capabilities, we tested vSDF over a VMware data center network by deploying 10 VMs within the local network, over three physical VMware ESX® servers, under one top-of-rack switch.

Each of the VMs has the following specifications:

- Ubuntu 12.04.2 Minimal Server OS
- 1 vCPU with 1 virtual socket and 1 core per socket
- 2048MB RAM
- 40GB Thick Provision Lazy Zeroed Hard Disk
- 1 E1000 Network Adapter

We enable Layer 3 broadcast and multicast traffic when multicast and broadcast discovery modes are run respectively, and restrict the network to run solely on unicast traffic when neighbor and random discovery modes are employed. For neighbor discovery mode, we generate a neighbor graph at random.

4.2 Experiment Design

To provide meaningful comparative results, we conducted experiments in which 30 DA processes and 1 SA process were executed, measuring how long it took the SA to discover each of the DA processes. The obvious evaluation criteria for our experiments were the percentage of DA instances that the SA could discover and what the rate of DA discovery was. In this paper, we formally define search coverage as the percentage of DA instances the SA could discover. The variables that we looked at include advertisement mode and the unicast or multicast service discovery protocol used.

Every experiment was conducted by starting all 30 DA processes, distributed evenly among the 10 machines, at approximately the same time and with the identical configuration—except in neighbor mode, in which each agent DA process had an independently generated set of neighbors. Neighbor selection was done once for each agent (both DA and SA) and was kept the same for all neighbor discovery experiments. Each agent was given a 10% chance of knowing about each of the machines that hosted agents. As a result, three agents were not configured to know about any machines, and one of the agents was allowed to know about 5 machines.

When all of the DA processes were running, a single SA was started on one of the machines and allowed to run for 300 seconds. This SA tried to find the DAs. Every time the SA established contact with a DA process, it logged the location of the DA and a timestamp of the contact. After the experiment, timestamps were normalized to the time since the SA started. Repeated contacts with the same DA were reduced to include only the earliest data point for each DA.

4.3 Experimental Results

The following subsections present the experimental results for various modes. The first set of experiments focused on search coverage in each of the service discovery modes without collaboration enabled. The second set of experiments focused on the impact of collaboration on the search coverage of each service discovery mode.

4.3.1 Service Discovery Modes

This first set of experiments was conducted to compare vSDF's unicast discovery modes—that is, n-hop neighbor discovery and random discovery—to the other conventional multicast and broadcast discovery modes without the use of collaboration.

From our experiments, both multicast and broadcast service discovery modes were able to achieve 100% search coverage regardless of which agents are active or passive. After 300s of deployment, neighbor discovery was able to achieve 90% search coverage in DAs active, SA passive mode and 100% in DAs passive, SA active mode. The performance difference in neighbor discovery can be explained by the asymmetric nature of the neighbor relationship. In active DA, passive SA mode it is explained by the three DAs that did not have any other agents in their known neighbors. These three agents were never able to find another agent to forward their announcements to the SA. In the case of passive DA, active SA, the three machines were able to get SA announcements relayed to them by other agents. In contrast, random discovery was able to achieve only 73% search coverage in DAs active, SA passive mode and 93% coverage in DAs passive, SA active mode. This is shown in Figures 5 and 6.

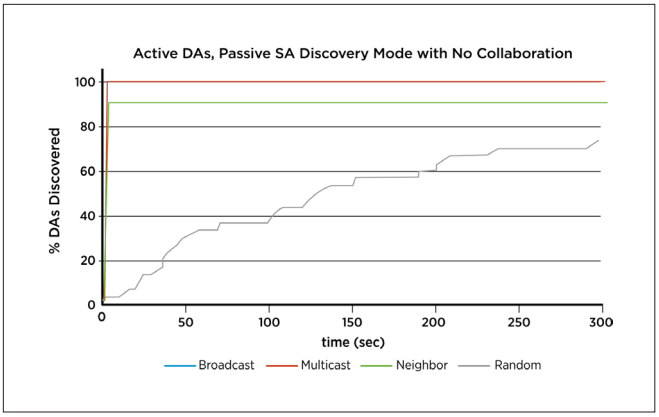


Figure 5. Active DAs, Passive SA Discovery Mode with No Collaboration

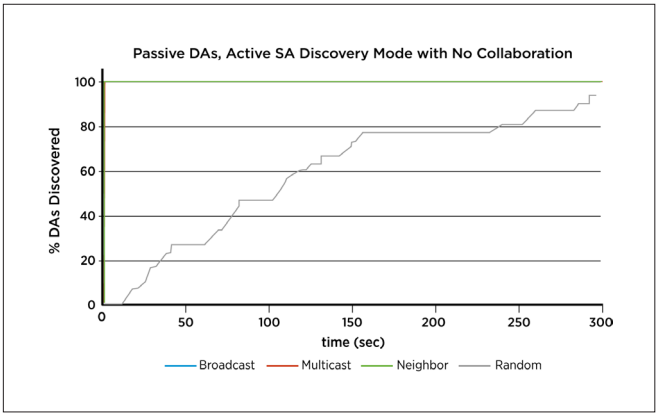


Figure 6. Passive DAs, Active SA Discovery Mode with No Collaboration

Despite the promising search coverage, the rate of random service discovery mode is, in reality, quite slow. When the trend lines in Figures 5 and 6 are observed, it can be seen that the other three service discovery modes— broadcast, multicast, and neighbor— reach their maximal search performance within a second of deployment, whereas random mode was still finding additional members at the end of the 300s experiment run. This is further emphasized in Figures 7 and 8, which show that random mode found few or no DAs within the first 10s.

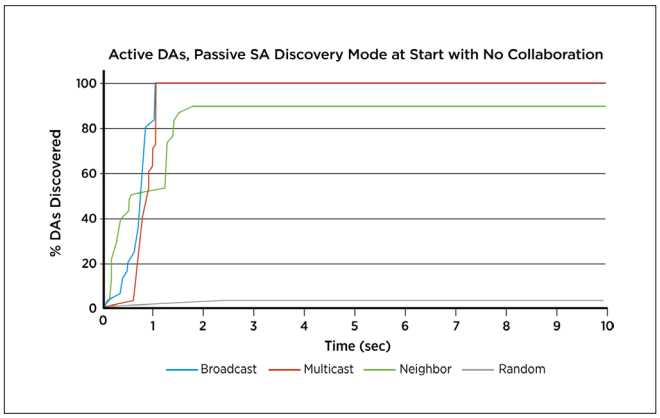


Figure 7. Active DAs, Passive SA Discovery Mode at Start with No Collaboration

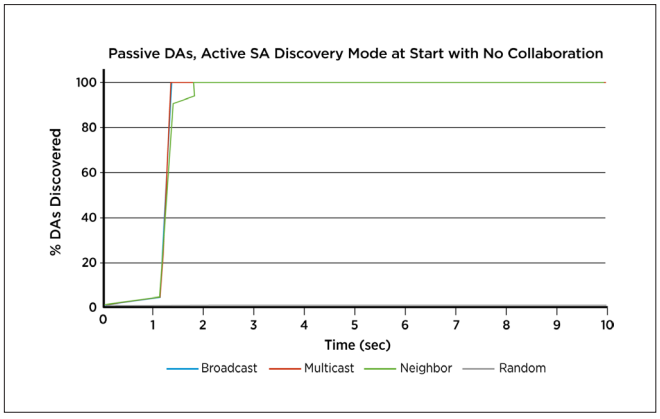


Figure 8. Passive DAs, Active SA Discovery Mode at Start with No Collaboration

4.3.2 Collaboration

In the second set of experiments, we enable collaboration among vSDF agents to study its impact on service discovery. Then, we compare it to the results obtained in 4.2.1. We were interested in learning whether collaboration was helpful in increasing number of targets discovered in unicast discovery modes.

In this set of experimental results, both multicast and broadcast service discovery modes were once again able to achieve 100% search coverage regardless of which agents are active or passive. When search performance in the first 300s of deployment in Figures 9 and 10 is observed, it can be seen that neighbor discovery was able to achieve 90% search coverage in DAs active, SA passive mode and 100% in DAs passive, SA active mode—unchanged from the results without collaboration in section 4.3.1. However, search coverage for random discovery in active DAs, passive SA mode improved from 73% without collaboration to 97% with collaboration. In contrast, random discovery in passive DAs, active SA mode fell from 93% to 83%. We speculate that this drop occurred because experiments were not run simultaneously, which led to the random-number generator producing different values.

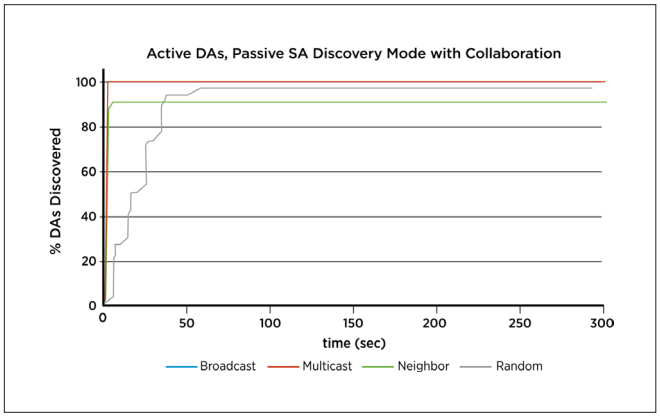


Figure 9. Active DA, Passive SA Discovery Mode with Collaboration

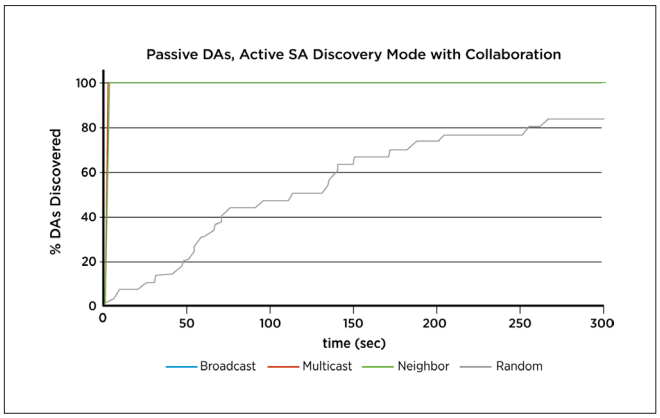


Figure 10. Passive DAs, Active SA Discovery Mode with Collaboration

These results demonstrate that search coverage can be improved with collaboration. In Figure 8, passive DA, active SA mode without collaboration was unable to find a single SA in the first 10s. However, with collaboration, it was able to find 6% of DAs (see Figure 12). The rate of search also improves for random discovery in active DAs, passive SA mode, when the trend lines in Figures 7 and 11 are

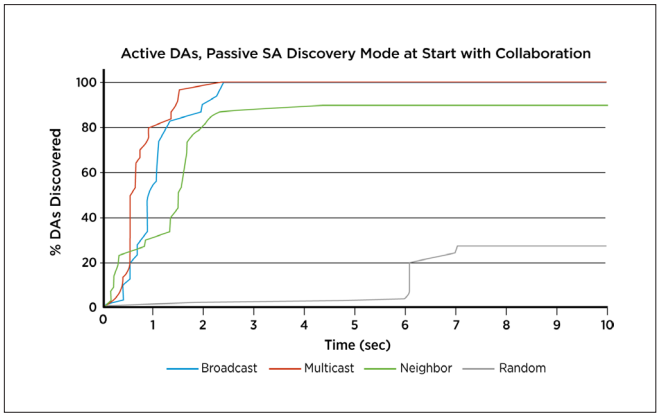


Figure 11. Active DAs, Passive SA Discovery Mode at Start with Collaboration

compared. It was able to find only 3% of DAs within the first 10s without collaboration, and an astounding 27% with collaboration.

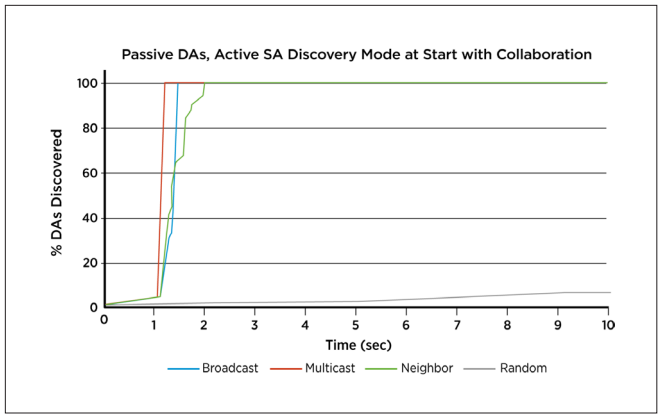


Figure 12. Passive DAs, Active SA Discovery Mode at Start with Collaboration

From these results, it is clear that search in passive DAs, active SA mode achieves better search performance and search rate than active DAs, passive SA mode, regardless of whether collaboration is enabled. This is because in passive DAs, active SA mode, the SA is actively sending vSDF messages in order to search for DAs rather than waiting to be contacted by DAs.

Overall, these experimental results present interesting implications for service discovery protocol deployment. First, unicast discovery modes are capable of finding a majority of DAs in an environment, although search performance might not necessarily be 100%. This property makes unicast discovery mode useful in environments where DAs are horizontally replicated and only one replica needs to be discovered. Second, collaboration can be enabled to improve both search speed and coverage in unicast discovery modes.

5. Discussion

In this section, we further discuss some aspects of vSDF. These topics include security and configuration.

5.1 Security

The nature of service discovery involves finding hosts that are not previously known; this can lead to many security vulnerabilities. The use of multicast and broadcast messages to discover services can lead to rogue-agent and man-in-the-middle attacks. A common scenario for rogue-agent attacks involves a malicious rogue host passively eavesdropping for service discovery protocol messages and then compromising service discovery agents that it finds. Because many service discovery protocols do not authenticate agents [4], malicious hosts can easily forge the identities of other benign agents to launch man-in-the-middle attacks.

When vSDF is deployed, the use of its unicast-based discovery modes can greatly reduce the probability of a rogue-agent attack. Because messages are not directly sent everywhere in the network, a rogue vSDF agent cannot eavesdrop as easily to obtain sensitive location information.

A way to prevent man-in-the-middle exploits is through the use of an authentication agent. In this mechanism, DA_AD messages advertise the location of an authentication agent instead of the location of the DA. An agent wanting to contact a DA first must contact the authentication agent to be authenticated, prior to communicating with the DA. In this case, an explicit assumption is that the authentication agent is trusted and cannot be forged. This mechanism coupled with end-to-end encryption of unicast mode messages can reduce the probability of a man-in-the middle attack.

5.2 Configuration

A significant portion of the utility of vSDF comes from how configurable it is. It can take advantage of hints provided to it to optimize performance. An administrator can give an agent doing neighbor discovery a list of possible peers. By using this list, neighbor discovery can distribute information more efficiently. Because the list is treated as a hint, virtually no delay is caused by incorrect hints in this list.

Neighbor discovery is also helped by the optional collaboration feature. When collaboration mode is enabled, agents that do not normally respond to information requests tell other agents about DAs. DAs also have additional behavior in collaboration mode: They store information about agents not part of their deployment, allowing them to act as additional lookup caches. When this mode is enabled, information propagates faster. If this faster propagation is not needed, generally in cases in which directory agent locations do not change often, this mode can be disabled on a per-agent basis. This slightly reduces the network load for user and service agents and reduces the memory footprint of the directory agent.

As an additional optimization, unicast messages can be transported via either TCP or UDP. We expect a typical installation scenario to involve agents running on machines with a reliable, high-bandwidth connection. In such installations, UDP will be very reliable. Because the system is designed to tolerate packet loss, setting unicast to UDP reduces the network footprint of the system.

In addition to these minor optimizations, an administrator can make one major optimization: setting agents to passive or active. An active agent is one that actively tries to find other agents in the system by advertising itself to other agents. A passive agent waits for advertisements sent by active agents. The more agents in a system that are active, the faster discovery will occur, but the amount of network traffic overhead will also increase. Obviously, in the extreme case that no agents are active, no discovery will occur. Active mode can be further configured by an administrator to adjust how often messages are sent out during initialization, how often they are sent out during steady state, and how long the system should remain in initialization before transitioning into steady state.

6. Conclusion and Future Work

In our future work, we would like to make some improvements to the framework itself. When IPv6 is widely deployed, vSDF's n-hop discovery mode should be updated so that it uses IPv6 NDP [12] to discover neighboring hosts, rather than be provided by a systems administrator. Furthermore, we would like to extend the framework so that it includes different DA selection models to better accommodate backups, replications, and so on.

vSDF is a powerful framework because it is a unification of discovery mechanisms that are used conventionally in distributed systems and computer networks and proposed by academia. This flexibility enables it to be deployed in nearly all network environments. The use of leases can increase service availability, and the simplicity of code integration can increase code reusability. In our experimental results, we showed that unicast discovery modes can be used as an effective replacement for broadcast and multicast ones if 100% search coverage is not required. Moreover, collaboration among vSDF agents can help improve the percentage and rate at which services are found. Overall, vSDF is a service discovery framework that can be deployed regardless of network environment restrictions.

7. Acknowledgments

We would like to thank to rest of the Cambridge Common Services Team for their assistance with the design and implementation of vSDF: Vijay Appadurai, Len Livshin, Margarita Miranda, Jun Zhang, and Jeff Zhou. We would also like to thank the members of the Cambridge Technical Operations team for assistance in setting up a test bed to conduct our experiments.

References

1 Reaz Ahmed et al., "Service discovery protocols: A comparative study," *Proceedings of IM*, pp. 15–18, May 2005.

2 Tom Bachert. IPv4 Multicast Security: A Network Perspective.]. http://www.sans.org/reading-room/whitepapers/networkdevs/ipv4-multicast-security-network-perspective-246.

3 Roberto Beraldi, "Service discovery in MANET via biased random walks," *Proceedings of the 1st International Conference on Autonomic Computing and Communication Systems*, Brussels, Belgium, 2007.

4 Christian Bettstetter and Christoph Renner, "A comparison of service discovery protocols and implementation of the service location protocol," *Proceedings of the 6th EUNICE Open European Summer School: Innovative Internet Applications*, 2000.

5 S. Cheshire and M. Krochmal. (2013, February) DNS-Based Service Discovery. http://tools.ietf.org/html/rfc6763.

6 S. Cheshire and M. Krochmal. (2013, February) Multicast DNS. http://tools.ietf.org/html/rfc6762.

7 Cisco Systems, Inc. (2002, March) Configuring a Rendezvous Point. http://www.cisco.com/en/US/docs/ios/solutions_docs/ip_multicast/White_papers/rps.html.

8 Cisco Systems, Inc. (2006) LLDP-MED and Cisco Discovery Protocol. http://www.cisco.com/en/US/technologies/tk652/tk701/technologies_white_paper0900aecd804cd46d.html.

9 E. Guttman, C. Perkins, and J. Veizades. (1999, June) Service Location Protocol, Version 2. http://tools.ietf.org/html/rfc2608.

10 Juniper Networks. (2010) Understanding IP Directed Broadcast for EX Series Switches. http://www.juniper.net/techpubs/en_US//junos/topics/concept/ip-directed-broadcast-ex-series.html.

11 Choonhwa Lee and Sumi Helal, "Protocols for service discovery in dynamic and mobile networks," *Proceedings of the International Journal of Computer Research*, vol. 11, no. 1, pp. 1–12, 2002.

12 T. Narten, W., E., and H. Soliman. (2007, September) Neighbor Discovery for IP version 6 (IPv6). http://tools.ietf.org/html/rfc4861.

13 Karl Saville. (2004) Introduction to the Bluetooth Service Discovery Protocol. http://homepages.inf.ed.ac.uk/group/sli_archive/slip0304_b/resources/com/karl/sdp/sdp_intro.html.

14 UPnP Forum. (2008, October) uPnP Device Architecture 1.1. http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf.

Notes

Notes

Notes

vmware®

VMware Academic Program (VMAP)

The VMware Academic Program advances the company's strategic objectives through collaborative research and other initiatives. The Program works in close partnerships with both the R&D and University Relations teams to engage with the global academic community.

A number of research programs operated by VMAP provide academic partners with opportunities to connect with VMware. These include:

- Annual Request for Proposals (see front cover pre-announcement), providing funding for a number of academic projects in a particular area of interest.
- Conference Sponsorship, supporting participation of VMware staff in academic conferences and providing financial support for conferences across a wide variety of technical domains.
- Graduate Fellowships, awarded to recognize outstanding PhD interns at VMware.
- VMAP Research Symposium, an opportunity for the academic community to learn more about VMware's R&D activities and existing collaborative projects.

We also support a number of technology initiatives. The VMAP licensing program (**http://vmware.com/go/vmap**) provides higher education institutions around the world with access to key VMware products. Our cloud technology enables the Next-Generation Education Environment (NEE, **http://labs.vmware.com/nee/**) to deliver virtual environments for educational use. New online education providers, such as EdX, use VMware solutions to deliver a uniform learning environment on multiple platforms.

Visit **http://labs.vmware.com/** to learn more about the VMware Academic Program.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

http://labs.vmware.com/academic