VMTorrent: Scalable P2P Virtual Machine Streaming

Joshua Reich
CS Dept., Princeton University
Princeton, NJ, USA
jreich@cs.princeton.edu

Oren Laadan
CS Dept., Columbia University
New York, NY, USA
orenl@cs.columbia.edu

Eli Brosh
Vidyo
Hackensack, NJ, USA
eli@vidyo.com

Alex Sherman
Google
New York, NY, USA
asherman@google.com

Vishal Misra, Jason Nieh, and Dan Rubenstein
CS Dept., Columbia University
New York, NY, USA
{misra,nieh,danr}@cs.columbia.edu

ABSTRACT

Clouds commonly store *Virtual Machine* (VM) images on networked storage. This poses a serious potential scalability bottleneck as launching a single fresh VM instance requires, at minimum, several hundred MB of network reads. As this bottleneck occurs most severely during read-intensive launching of new VMs, we focus on scalably minimizing time to boot a VM and load its critical applications.

While effective scalable P2P streaming techniques for Video on Demand (VOD) scenarios where blocks arrive in-order and at constant rate are available, no techniques address scalable large-executable streaming. VM execution is non-deterministic, divergent, variable rate, and cannot miss blocks. VMTORRENT introduces a novel combination of block prioritization, profile-based execution prefetch, on-demand fetch, and decoupling of VM image presentation from underlying data-stream. VMTORRENT provides the first complete and effective solution to this growing scalability problem that is based on making better use of existing capacity, instead of throwing more hardware at it.

Supported by analytic modeling, we present comprehensive experimental evaluation of VMTORRENT on real systems at scale, demonstrating the effectiveness of VMTORRENT. We find that VMTORRENT supports comparable execution time to that achieved using local disk. VMTORRENT maintains this performance while scaling to 100 instances, providing up to ***11x*** speedup over current state-of-the-art and ***30x*** over traditional network storage.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

Keywords

BitTorrent, Cloud Computing, File Systems, On-Demand, P2P, Swarming, Virtual Appliances, Virtual Machines

1. INTRODUCTION

Traditionally, *virtual machine monitors (VMMs)*[1] have run *virtual machines (VMs)* off locally stored VM images. By combining host-level virtualization with modern networking and data center facilities, the *cloud computing* paradigm has enabled a wide range of new applications for VM technology. Both virtualized public (*e.g.,* Amazon EC2, Microsoft Hyper-V cloud) and private (*e.g.,* Cisco UCS, IBM Cloudburst, Oracle Exalogic Elastic Cloud) *Infrastructure as a Service (IaaS)* clouds have enabled businesses to move their operations from applications (*e.g.,* compute node, webserver, user desktop) running on dedicated hardware to shared infrastructure on which *virtual servers*, *virtual appliances*, and *virtual desktops* run. Doing so enables more efficient hardware utilization, easier management, and quicker failure recovery.

However, to realize these benefits, VMs may often need to be launched on machines that have not been pre-loaded with a copy of the corresponding VM image. By far the most common cloud setup stores VM images remotely on either *storage area network (SAN)* or *network-attached storage (NAS)*, as either *primary* or *secondary* storage Systems utilizing the network for primary storage employ a remote file system abstraction to stream VM images directly from network storage (*e.g.,* Amazon EC2/EBS). Systems utilizing the network as secondary storage, must first download a VM's complete virtual disk image (*e.g.,* OpenStack Glance, Amazon EC2/S3) to local primary storage before the VM can run. In either case, attempts to launch large numbers of fresh VM instances run straight into a network bottleneck.

1.1 The Network Bottleneck

Depending on the VM, and whether network storage is primary or secondary, hundreds of MB to several GB of VM image reads must pass over the network to launch even a single fresh VM instance. In the private cloud, this problem has proven sufficiently common to earn the moniker "boot storm". Likewise, users of OpenStack, open source software for building private and public clouds, have reported that "scale limits are due to simultaneous loading rather than total number of nodes" [16]. In response, at least one recent developer proposal has been made to replace or supplement VM launch architecture for greater scalability [14]. Moreover, with the advent of spot-pricing schemes that encourage the unpredictable launch of large temporary VM deploy-

[1]Also known as *hypervisors*.

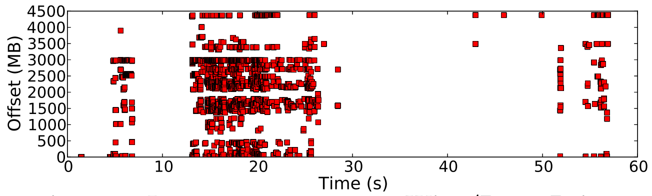


Figure 1: Image access pattern. Win7/PowerPoint.

ments and cross-cloud VM ensemble migration, this issue will likely become more significant for the public clouds[2].

Commercial solutions have addressed this network bottleneck to storage by throwing hardware at it: over-provisioning SAN/NAS, adding flash based storage [2], solid state drives [1], and other in-network hardware caches. However, often this approach becomes prohibitively expensive at current scales, and cloud sizes look to only increase in the future.

1.2 VMTorrent

vmTorrent aims to ***scalably minimize time needed for cloud-based VMs to boot and load critical applications***. Realizing this goal will improve both virtual desktop user experience and virtual server/appliance deployment.

vmTorrent addresses this challenge by recognizing its similarity to that of data distribution. The networking community has developed highly effective scalable peer-to-peer (P2P) techniques for both bulk data download (*e.g.,* BitTorrent) and streaming strictly-ordered data (*e.g.,* Video on Demand). However, the space in between these two extremes has been left open. No techniques exist for scalably streaming partially structured data such as that shown in Figure 1, which plots the clustering of disk accesses in both space (y-axis) and time (x-axis) for a representative run of a PowerPoint application on a Windows 7 VM. This is, perhaps, because up until recently, there has been little application for such generalization. Cloud-based virtual machine image execution provides that motivation.

We provide the first P2P approach to the general problem of streaming large files whose access patterns are not strictly structured like video. vmTorrent leverages the structure of individual VM images[3] in a straightforward way to radically increase the number of VMs which may be efficiently launched on a given hardware configuration. vmTorrent applies to both IaaS public and private clouds, Virtual Desktop Infrastructure (VDI), and is VMM agnostic.

1.3 Contributions

In this paper, we make the following contributions:

1. Decouple P2P-delivery from stream presentation: Instead of requiring applications purpose-written for P2P data streams, vmTorrent makes the P2P data stream transparent to applications by using the same file system abstraction as traditional remote file systems - allowing almost any application to easily use P2P-streamed data (Section 2.1).

2. Profile-based image streaming: We introduce novel mechansims enabling effective VM image streaming by prefetching pieces based on *VM image profiling* and utilization of *piece selection* policies that balance cache misses against swarm efficiency (Section 2.2).

3. Model profile-based P2P image streaming: We express the expected playback pattern of a given VM image in terms of a small set of parameters - including network speed, image size, and number of instances launched - providing a concrete foundation for discussing design choices and extrapolating performance (Section 3).

4. vmTorrent **prototype:** Implemented and deployed on two different hardware testbeds. We measure performance on a variety of VM/workload combinations, using up to 100 physical client peers. We find vmTorrent delivers up to an **11x speedup** over a standard P2P approach that does not incorporate profiling and a **30x speedup** over traditional remote file system approaches. vmTorrent provides equivalent performance to that of local disk execution for all workload sizes (Section 4).

[2]Regrettably, we cannot currently quantify the size and scope of this problem, as cloud providers are reluctant to provide access to their data due to its commercial value.

[3]vmTorrent naturally handles VM images comprising guest OS and applications. However, our techniques require modification for persistent VM images that diverge over time. While many, if not most, VMs are either nonpersistent or non-divergent, we provide discussion of how divergent images may be handled in Section 2.3.

2. SYSTEM DESIGN

vmTorrent's design incorporates two novel features:

1. It decouples the P2P-delivery from data stream presentation mechanism - allowing hypervisors to access this data stream as if it were stored locally.

2. It introduces novel profile-based prefetch - allowing VM images to be scalably streamed with a P2P swarm

To decouple the delivery from the data stream, vmTorrent utilizes a *custom file system server (FS)* that effectively virtualizes the VM image. Through a user-level file system front-end, FS provides the appearance of a completely local VM image, while servicing VMM reads and writes by connecting to a network backend (Section 2.1).

This decoupling allows us to make that network backend one which utilizes P2P techniques for scalable download. However, current P2P techniques, such as bulk download and Video on Demand (VOD), are not well suited to VM image streaming. Thus our *custom P2P manager (P2PM)* incorporates novel piece selection policies, based on VM image access *profiling*, in order to serve the FS (Section 2.2).

Figure 2 illustrates the operation of the system. As the VM starts to execute, (1) the VMM tries to access the disk image in response to the guest's virtual disk accesses. If a block is not yet present (a cache miss), then (2) FS requests that block from P2PM. Meanwhile, (3) based on incoming FS requests, P2PM fetches and, based on VM image *profiles*, prefetches image pieces. (4) Concurrently, P2PM uploads local pieces to other swarm members in response to their requests. P2PM stores each piece locally and (5) passes the file blocks comprising this piece to the file server as they are requested by the VMM (6). The file server handles incoming write requests using *copy-on-write* to local storage or memory, while P2PM retains the unmodified original block to share with peers.

vmTorrent instances are intended to run as close to the VMM as possible, replacing the primary data store. To integrate with a cloud orchestration platform, the hypervisor must be redirected from the former VM image location mounted on primary storage to the FS mount which connects both to network storage (via a dedicated vmTorrent seed instance deployed on/near the network storage) and other peers. The only requirement for applications using

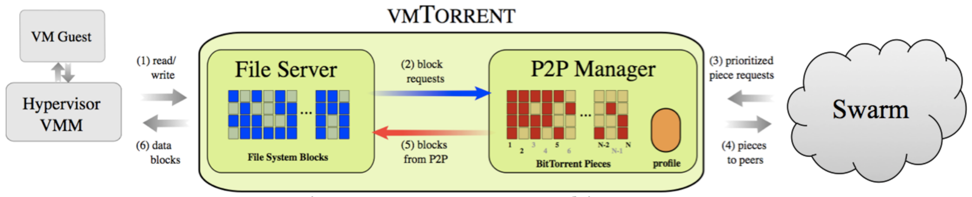


Figure 2: vmTorrent architecture

vmTorrent is that they tolerate occasional high-latency operations on files - currently non-uncommon for accesses through traditional remote file systems.

2.1 File Server

The vmTorrent FS is responsible for providing read and write access to the guest VM's disk image. FS operates similarly to traditional remote file systems: the FS root directory is mounted in a designated location, making its files visible to the system. All disk accesses performed to the sub-tree under that mount point are intercepted by the operating system and delegated to FS. In this way FS enables a standard file system view that is indistinguishable from that of other common remote file systems such as NFS and CIFS, and with similar latency properties - making vmTorrent both fully transparent to the VMM and inter-operable with the SAN/NAS compatible hypervisors used in today's clouds.

When a new guest VM is launched, FS creates a placeholder file for that VM's disk image, which from the VMM's view of the file system is indistinguishable from a complete local copy of the image. However, initially this file is empty; its content is gradually streamed in from the P2P network. As soon as the local placeholder is present FS can begin to serve requests to access the data - even if no data is yet present. For requested blocks that have been prefetched (cache hits), the server will respond immediately. However, if the VMM suffers a cache miss, attempting to access blocks that are not yet present, FS will issue *demand requests* for these blocks to P2PM which will stall until the needed blocks have been received from the swarm.

2.2 P2P Manager

To support smooth VM execution, P2PM needs to provide low-latency delivery of requested pieces to FS. This requires both proportionally many cache hits and minimally delayed network delivery on cache misses.

2.2.1 Applicability of P2P VOD

P2P VOD streaming [18] addresses the first of these two (proportionally many cache hits) by dividing its bandwidth between pieces needed urgently and random pieces from elsewhere in the video. Since pieces of a video are almost always accessed sequentially, it is trivial to determine the active *playback window* (those pieces needed immediately to ensure smooth playback) when the video is encoded with a constant bit rate (CBR).

However, this sequentiality proves a double-edged sword. Should requests be restricted to only pieces in the playback window, all VOD peers starting at the same time will request the exact same small set of pieces from the seed(s). In such a case, there will be little *piece diversity* amongst peers, meaning that they have little to share with one another. Hence, *swarming efficiency* (the effective upload rate offered by the swarm) - and thereby scalability - drops dramatically. To avoid this and ensure sufficient piece diversity, VOD techniques add random piece requests. There is relatively little downside to devoting a fraction of bandwidth to random prefetches, as all pieces in the video stream will eventually be used.

2.2.2 The Challenge of VM Image Streaming

P2PM pursues a similar strategy. However, VM image streaming faces issues CBR video streaming does not.

• ***Sparsity***: the large majority of a VM image may never be accessed in a complete execution cycle (see Table 2). Thus, randomly downloaded pieces will likely never be of use, the bandwidth used to obtain them being entirely wasted.

• ***Stochasticity***: VM image playback varies dynamically and sometimes unpredictably as different workloads execute. We have found that the set of blocks accessed and the order of access varies from run to run - even when executing the same workload on the same machine. This makes predicting the VM's "playback window" challenging.

• ***Rate variability***: VM image access rates vary drastically during the course of execution. Sequences for boot, login, and application startup consist of one or more highly-intense spikes in read request rates, interspersed by periods of almost no image I/O activity. This is as, if not more, challenging than optimizing P2P variable bit rate (VBR) video delivery - a problem that is currently not well understood.

• ***Execution sensitivity***: P2P VOD can tolerate piece delivery failure of individual blocks and still continue correct playback. However, correct VM image execution requires every requested piece be obtained and thus must stall on even a single cache miss.

These factors combine with one another to pose a much greater challenge that that of CBR video streaming. Sparsity and stochasticity together imply that a successful VM image streaming system must predict future accesses and do so well - as false positive mispredictions will most likely never be used. Stochasticity combined with execution sensitivity makes cache misses likely and expensive (compared to the unlikely and individually un-impactful VOD missed deadlines). Rate variability requires the development of new analytic models and, combined with execution sensitivity, forces the optimal P2P piece selection policy to vary dependent on the expected upcoming access rate from the current playback point.

Perhaps counter-intuitively, if the expected upcoming access rates will exceed that of the network, then it may be better for a peer to randomize its requests - since it will have to stall anyway - and increase the swarm capacity. Conversely, when upcoming access rates are slightly lower than that of the network, the optimal strategy may be to exploit the swarm to obtain pieces within the upcoming playback window instead - since potentially costly stalls may be avoided.

We begin addressing this wide set of challenges by first introducing simple *profiling* techniques for playback window prediction (Section 2.2.3) and playback window randomized piece selection policies to balance swarm efficiency against cache misses (Section 2.2.4).

2.2.3 *Profiling*

The first goal of profiling is to mitigate image sparsity by identifying which pieces are highly unlikely ever to be accessed and blacklist them from ever being prefetched. Since the majority of pieces are never accessed, only those that have been seen in at least one profiled run will be entered into the profile. The second goal of profiling is support calculation of the VM playback window, a pre-requisite for effective prefetching.

In this work, we took a straightforward approach to building a profile. Each profile built was specific to both a VM image and a workload. For each image/workload pair, we ran the workload one or more times from boot through shutdown. Through FS, in each run we tracked which image pieces were accessed and when. For each piece that showed up in at least one run, we averaged the times at which it appeared and ordered the pieces accordingly from earliest to latest. The profile produced comprised this ordered list of rows, each row containing average appearance time, piece index, and proportion of runs in which that piece appeared.

The advantages of this approach are several.

- These profiles are inexpensive/quick to construct. A single-run profile can be created in the time it takes to execute the workload once and can support reasonably accurate playback window prediction. Multi-run profiles help further.
- Such profiles are compact. Both single and multi-run profiles created this way are of negligible size (<512KB before compression), especially when compared to that of the corresponding VM images. This compactness facilitates either pre-loading or distributing many profiles on the fly.
- They can produce very low overhead playback window predictions. Utilizing a single profile, the playback window will simply be the list of pieces highest in the profile that have not yet been either used or cached.
- They can be used as building blocks for more sophisticated prediction. The playback window prediction component may track which of several profiles best matches the current execution pattern and uses the current best match, or some weighted combination of the top several matches.
- Finally, they are conceptually simple - both facilitating manual inspection and serving as a baseline for future work.

This approach does have clear limitations. The playback window predictions we produce are not optimal. Moreover, potentially significant information is lost in the averaging process. Future work might be able to preserve more of this information without increasing the profile size too dramatically. Finally, these profiles are specific to each image/workload combination. More general profiling techniques may analyze common access sequences across workloads, or utilize guest file system level information to generalize across similar VM images.

2.2.4 *Piece Selection*

The goal of the vmTorrent piece selection policy is to balance the need to (1) *minimize cache misses* and (2) *maximize swarm efficiency.*

In this work, motivated by execution sensitivity, we have chosen to give cache misses absolute priority over prefetches (although prefetches already in-progress will not be pre-empted). To provide some piece diversity, we utilize a *window-randomized* selection policy that picks one of the first k pieces in the predicted playback window to request. This window size k is a tunable parameter that attempts to balance the urgency of pieces against the need for sufficient peer diversity. If a large k is chosen, pieces fail to be prefetched in the order of predicted use, resulting in expensive cache misses. On the other hand, the choice of an overly small k will damage swarm efficiency, leading to scalability problems since too little piece diversity will be achieved. Consequently, as the number of peers decreases or the workload diversity increases, the optimal k will tend to be lower (since the pressure on centralized servers is lower and the relative piece diversity in the network greater) and vice-versa.

2.3 VMTorrent's Place in the Cloud

vmTorrent targets a very specific problem - getting those parts of the VM image needed by VMMs to those VMMs with minimal delay. vmTorrent handles writes through copy-on-write to a local cache residing within the FS front-end. These changes disappear unless copied back across the network by some other mechanism. vmTorrent's design focuses on scenarios in which a single unchanging (or infrequently changed) image needs to be rapidly scaled up and down. Examples of such images include standardized software development environments, front-end servers for traditional tiered web applications, pre-configured compute nodes, or standardized user desktop environments in which transactional user data and customizations are stored in a traditional network file system which the guest VM is pre-configured to access [10].

However, this choice by no means restricts vmTorrent's applicability to only such scenarios. By tracking those blocks that have been modified and saving these along with other image metadata to the SAN/NAS, we can support data write-back within the VM while still preserving P2P fetch for those majority of unchanged blocks in the VM image (particularly those encoding the boot sequence and applications). We can further extend our technique using mechanisms such as Content-Based Block Caching [21], along the lines described in [31].

3. MODEL

The primary goal of this section is to capture the essential dynamics of P2P VM image streaming. We aim to thereby explain our design and experimental results, extrapolate vmTorrent's performance outside of our experimental setting for sensitivity testing, provide performance bounds, and rigorously define our objective function.

A VM image streaming system's designer has (at least) two critical choices - that of *piece selection policy* ψ and *distribution model* ϕ. The selection policy specifies the order and timing of piece requests originating from a given peer - clearly critical in determining whether playback will progress or stall at any given moment. The distribution model determines which nodes provide upload capacity. One centralized source will run out of upload capacity as the number of peers grows, while the raw upload capacity of P2P distribution scales with the number of peers. Of particular interest is that the selection policy may effect the comparative performance of different distribution models. Section 3.1 presents two simplified policy models, while Section 3.2 considers fully centralized and fully decentralized distributions.

These policy and distribution model choices combine with parameters drawn from the cloud setting and particular VM image to form a complete description of the VM image streaming scenario. These parameters are: the *service wait time* between peer request and receipt of first download bit (W), network speed (r_{net}), piece size (S), and the ideal VM image playback function $M(t)$ (in bytes accessed by time t) modeling the total image size accessed by hypervisor executing on a fully memory-cached image[4]. Given these, our goal is to express the ***average playback process*** $P(t)$ of a vmTorrent instance (*peer*).

With P, we can now rigorously define the objective specified in Section 1 as

$$\min_{\psi} \{\ P_{\psi}^{-1}(A) - M^{-1}(A)\ \} \tag{1}$$

where A is the total number of bytes accessed from boot through workload completion.

3.1 Policy Model

For the sake of simplicity, we consider only strict demand fetching and in-order profile based prefetching in this section. These scenarios are covered, respectively, by Sections 3.1.1 and 3.1.2 which express $P(t)$ in terms of the quantities defined above and one additional term, the *effective peer download rate* r. r describes a peer's effective download as a function of r_{net}, W, S, and n.

3.1.1 Demand

We begin by considering the policy ψ_{demand} where peers only fetch blocks as they are needed. The playback that occurs in a very short time-span will either be the number of bytes a fully-memory cached version would achieve from the current playback point y_t, or, if the effective peer rate was insufficient to sustain this, the amount that could be streamed in this time

$$P(t+dt) - P(t) = \min(M(y_t+dt) - M(y_t), rdt)\,. \tag{2}$$

The current playback point itself is determined by first calculating the number of bytes played back, which is simply $P(t)$. We then compute the corresponding time y_t that a fully-memory cached version would have accessed that same number of bytes by inverting M

$$y_t = M^{-1}(P(t))\,.$$

Equation (2) can be easily manipulated to express $P(t)$ in differential form (playback rate)

$$\Rightarrow \frac{P(t+dt)-P(t)}{dt} = \min(\frac{rdt}{dt}, \frac{M(y_t+dt)-M(y_t)}{dt}) \tag{3}$$

$$\Rightarrow \frac{dP}{dt}(t) = \min(r, \frac{dM}{dt}(y_t))\,. \tag{4}$$

3.1.2 In-Order Profile Prefetch

To incorporate the impact of prefetching, we add a buffer to our model. In any period during which the access rate is lower than the effective rate, content will be buffered. In periods where the access rate is higher than the effective rate, buffered accesses can make up (some) of the difference. Denoting the buffer as $B(t)$, we can express the buffer's evolution as a continuous-time Lindley process

$$B(t+dt) = [(B(t)+rdt) - (M(y_t+dt) - M(y_t))]^{+}\,. \tag{5}$$

We then adapt the demand model shown in equation (4) to include the extra content added from the buffer $B(t)$ to that available from the network

$$\Rightarrow \frac{dP}{dt}(t) = \min(r + \frac{B(t)}{dt}, \frac{dM}{dt}(y_t))\,. \tag{6}$$

Since dt is infinitesimal and the buffer size cannot be negative, we may re-write Equation 6 as the more intuitive

$$\frac{dP}{dt}(t) = \begin{cases} \frac{dM}{dt}(y_t) & \text{if} \quad B(t) > 0 \\ \min(r, \frac{dM}{dt}(y_t)) & \text{if} \quad B(t) = 0\,. \end{cases} \tag{7}$$

The reader will note that this model describes a simplified situation in which (a) demand fetches always take precedence over prefetches, (b) prefetched pieces are always immediately utilizable (*i.e.,* perfect prediction). Assumption (a) accurately reflects our choice (Section 2.2.4) to deterministically prioritize demand over prefetch. We make assumption (b) as modeling the dynamics of prefetched pieces that cannot be immediately utilized becomes quite complicated. Consequently, the playback function produced by this model will provide an upper bound on actual performance.

3.2 Distribution Model

Sections 3.2.1 and 3.2.2 provide approximations for r as for the centralized and P2P scenarios, respectively.

3.2.1 Centralized

We begin by assuming the server splits available bandwidth r_{net} evenly among the n peers. For a server, the service wait time W combines two potential factors: network round-trip time (RTT) and the time until the serving peer has a piece ready. This latter component represents server congestion and grows very slowly in its first phase after which it spikes sharply [39]. Accordingly, the time needed for a single peer to obtain a single piece is simply the service wait time plus the time for the download to complete

$$t_{piece} = W_{cs}(n) + S/\frac{r_{net}}{n}\,. \tag{8}$$

Assuming that the hypervisor will not issue new read and write requests while it blocks on those previously issued[5], the effective peer download rate r is

$$\frac{S}{r} = t_{piece} = W_{cs}(n) + \frac{S}{\frac{r_{net}}{n}} \Rightarrow r = \frac{r_{net}S}{n(\frac{r_{net}}{n}W_{cs}(n)+S)}\,. \tag{9}$$

3.2.2 P2P

Assuming that the network provides full bi-sectional bandwidth allowing any pair of peers to exchange data at r_{net}, by the pigeonhole principle each peer should be able to download at rate r_{net}. Additionally, peers each maintain a fixed number of connections with other peers which does not grow with n. Consequently, unlike in the server case, the wait

[4]It is trivial to obtain M during profiling so long as the profiled hardware matches the streaming hardware, although, as for profiling, a new M must be produced for each VM/workload pair.

[5]In practice, the hypervisor may issue several read requests simultaneously. However, these requests are for consecutive byte ranges (thus falling within the same one or two pieces) the vast majority of the time in our experience.

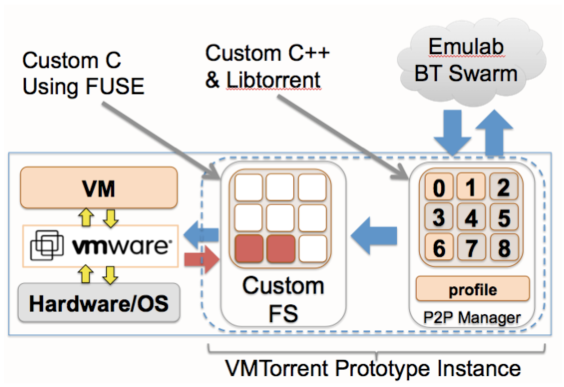


Figure 3: vmTorrent prototype.

time W will not depend directly on n. Here W is an inverse function of piece diversity d: the higher d, the shorter the wait until one of the fixed number of peers has that piece available. However, d itself is a function of n, t and the fetch/prefetch policy ψ. As in modeling VOD, as the number of peers increases, the time needed to achieve high piece diversity under a given fetch/prefetch scheme ψ will become increasingly significant [29]. Thus the time to download one piece from the swarm is

$$t_{piece} = W_{p2p}(d(\psi, n, t) + \frac{S}{r_{net}} \qquad (10)$$

giving the effective swarm rate

$$r_{swarm} = \frac{r_{net}S}{r_{net}W_{p2p}(d(\psi, n, t)) + S}. \qquad (11)$$

However, our architecture does utilize a dedicated network storage seed. Consequently, the rate a vmTorrent peer in swarming mode should expect is the swarm rate plus the rate given in Equation (9)

$$r = \frac{r_{net}S}{r_{net}W_{p2p}(d(\psi, n, t)) + S} + \frac{r_{net}S}{n(\frac{r_{net}}{n}W_{cs}(n) + S)}.^{6} \qquad (12)$$

This equation provides an analytic basis for our decision to utilize a window-randomized piece selection policy (which decreases W_{p2p}) instead of an in-order piece selection policy.

4. EXPERIMENTAL EVALUATION

We implement a prototype version of vmTorrent for Linux hosts and conduct extensive experimental evaluation. Our foremost aim is to determine vmTorrent's efficacy in providing quick and scalable VM distribution and execution. Our primary assessment metric is *completion time* $P^{-1}(A)$: the time that it takes to execute a VM/workload.

4.1 Experimental Setup

Our prototype, shown in Figure 3, comprises a user-space file server tightly integrated with a P2P client. This implementation is capable of transparent operation with any VMM/guest VM combination, and without requiring any changes to the VMM or guest VMs.

Our file server builds on *bindfs* [3], a FUSE [35]-based file system for directory mirroring. The server provides a virtual file system that stores a locally modifiable copy of the guest VM image. While a user-level file server potentially introduces performance penalties over that of a kernel-based implementation, we show this overhead is sufficiently low for our purposes (Section 4.3).

Our prototype's P2P component is built on top of the *libtorrent* 0.15.5 [25] library. Like most BitTorrent clients, libtorrent optimizes its piece selection policy to maximize file download throughput. Since our goal is to minimize the delay for downloading particular pieces needed for the VM execution, we modify libtorrent's default rarest-first piece selection policy to support the low-latency prioritized piece download vmTorrent requires (Section 2.2.4).

The majority of our results come from deployment of vmTorrent on the University of Utah's Emulab network testbed [40]. Each experiment consists of multiple hosts and an initial server which stays in the system for the duration of the experiment. The hosts are selected from a pool of 160 "d710" machines at the University of Utah's Emulab installation. Each host was equipped with a 64-bit 2.4 GHz Quad-Core Intel Xeon X5530 CPU, 12 GB RAM, and a Barracuda ES.2 SATA 3.0- Gbps/7200-RPM/250- GB local disk. The hosts run 64-bit Ubuntu 10.04.1 LTS with a modified 2.6.32-24 Linux kernel provided by Emulab. The hosts use VMware Workstation 7.1.0 build-261024 as the VMM. All experiment control and logging are conducted on a separate control network interface.

As this testbed only provides for 100 Mbps LAN connectivity, slow for today's production environments, we supplement these results with ongoing experiments from Princeton's recently deployed VICCI (1 Gbps) testbed [32]. The VICCI testbed consists of 70 Dell R410 PowerEdge servers, each with 2 Intel Xeon X5650 CPUs and 48GB RAM. However, unlike Emulab, VICCI does not provide bare-metal access. Instead vmTorrent instances run in virtual containers and have access to only a fraction of the overall system resources. Likewise network capacity is shared communally among the simultaneous experiments of multiple investigators and there is no separate control network.

4.2 Methodology

To study the system's scalability, we vary the number n of concurrent physical machines hosting vmTorrent instances (peers) from 2 to 100. For each set of parameters, we present the results averaged over several trials.

We consider the performance of the four scenarios produced by combining piece selection policies {`demand`, `profile`-based prefetch} and distribution models {`centralized`, `P2P`}, plus that of image access from local spinning disk.

1. `cs_d`: `centralized` distribution, `demand` policy.
2. `cs_p`: `centralized` distribution, `profile` policy.
3. `p2p_d`: `P2P` distribution, `demand` policy.
4. `p2p_p`: `P2P` distribution, window-randomized `profile` policy (window size $k = 300$).
5. `local`: Pre-distributed on local spinning disk.

Comparing `p2p_p` against `local` is helpful in providing insight as to whether and how well a given cloud setup can support virtualized execution. However, our main comparison is `p2p_p` against `cs_d`, and `p2p_d`, which represent, respectively, the standard and current state-of-the-art solutions in this domain *vis-a-viz* scalability (Section 5).

Running trials at scale for centralized distribution took the lion's share of our hardware time. Since our model pre-

[6] Note that for $n = 1$, $d = 0$ (since there are no other peers) implying $W_{p2p} = \infty$, $r_{swarm} = 0$, and $r = \frac{r_{net}S}{r_{net}W_{cs}(1)+S}$.

Workload	OS Type	Description
Boot-Shutdown	All	Boot, login, and shutdown.
Latex	Linux	Compile 30-page Latex document, view result in PDF viewer.
DocEdit	Linux	Create new OpenOffice document save, reopen, edit, spell-check.
PowerPoint	Windows	View PowerPoint slide-show.
Multimedia	Windows	Play 30 second music file.

Table 1: Workloads.

VM	Size	A (access to complete workload)	%
Fedora	4.2 GB	320 MB- 360 MB	8-9%
Ubuntu	3.9 GB	235 MB- 400 MB	6-10%
Win7	4.3 GB	295 MB- 350 MB	7-8%

Table 2: Virtual machines.

dicts that `cs_d` and `cs_p` converge as n increases - an intuitive result as opportunity for prefetch becomes negligible when server bandwidth is saturated, we chose to run only `cs_d` for all scaling experiments, as it is by far more widely-deployed than `cs_p`.

We use two profiles for our experiments per VM/workload combination: a small profile and a large profile. The small profile uses just one sample run. It simply lists the order in which blocks are accessed for the first time in the sample run. The large profile is built using 1000 runs. As both the small and large profiles are of negligible size (<512K), we pre-distribute them.

For simplicity, our evaluation focuses on non-persistent VM sets, but this by no means restricts VMTORRENT's applicability. Section 2.3 provides further discussion. For the same reason, we run identical workloads on each peer as we wished to focus on scalability, not playback-window prediction. That said, our results indicate that the greater piece diversity produced by running mixed workload sets, might well offset the increased cache-miss rates of using a less accurate profile (see Sections 4.4 and 4.5).

Unless otherwise noted, we normalize our results with respect to M the fully memory-cached playback function. M provides a theoretical best-case scenario as neither network nor disk I/O is incurred (Section 3).

4.2.1 VMs and Workloads

For our experiments we used a set of workload scenarios designed to simulate common short VDI user workloads. Table 1 lists the usage scenarios for our experiments. These scenarios represent different user activities on desktop virtual appliances. Each benchmark consists of first booting the guest VM, then executing a script that performs a desired workload by mimicking user actions and finally shutdown the guest VM. To automate the execution of these benchmarks, we configured guest VMs to auto-login once booted, and then execute a script that selects the appropriate workload to run.

While the set of workloads we could explore in this work were necessarily limited, we believe the results produced apply to both longer VDI workloads, as well as those run on virtual servers and virtual appliances. For each of these, as we will shortly see, the boot and login sequence poses the greatest challenge in both latency and volume of image access. Consequently, shorter workloads provide the most challenging test with respect to our objective function. VMTORRENT's performance on longer workloads should only exceed that of shorter ones. Moreover, the workloads we examine (*e.g.,* Latex compile, playing a music file) possess the same essential characteristics of those involved in launching virtual servers and appliances: boot and load critical applications. Likewise, the VDI workload set contains both I/O-intensive interactive and CPU/RAM-intensive batch workloads, corresponding to those seen by virtual servers and appliances (*e.g.,* interactive webserver workload, batch processing workload).

In Table 2, we list VM size, the range of A (the total number of bytes accessed from boot through workload completion) across the workloads considered, and percentage of A in total image size.

4.3 Baseline Performance

We begin our investigation by exploring the performance that can be expected for a single peer, $n = 1$. Our goals in this section are to (1) check our model predictions using quantitative evaluation on actual experimental parameters and (2) provide intuition as to how 100 Mbps and 1 Gbps network rates are likely to affect baseline performance by examining the bounds produced by our model.

For $n = 1$, there is no distinction between distribution method (as there are no peers to share with). Indeed in this case both our centralized and P2P distribution models predict the same effective rate

$$r = \frac{r_{net}S}{r_{net}W_{cs}(1) + S} \tag{13}$$

(refer to Equations (9), (12) and Footnote 5).

Accordingly in this sub-section, the only design variable is selection policy. We examine the policies `demand` and in-order `profile`. We utilize in-order, instead of window-randomized, prefetching here as we are concerned with bounding performance and in-order provides the our best prediction mechanism.

4.3.1 Experimental Parameters

As mentioned above, $n = 1$ and $r_{net} = \{100\,Gbps, 1Mbps\}$. M and A were collected during profiling for each combination of VM and workload. The piece size S used for all experiments was 16 KB, which leaves only $W_{cs}(1)$ undefined.

As discussed in Section 3.2.1, W_{cs} has two components, a network-delay term and a server congestion term. Since there are no other peers and the server is dedicated, this second term is a small constant Q representing the base time needed for the server to processes a request

$$W_{cs}(1) = RTT_{net} + Q\,. \tag{14}$$

Our implementation provided Q in the range of 5 x 10^{-4} s. The measured network RTT at 100 Mbps averaged roughly 5 x 10^{-4} s as well. For 1 Gbps networks, bit rates should be 10x larger than at 100 Mbps, yielding $RTT_{1\,Gbps} = 5 \text{ x } 10^{-5}$ s. Substituting into Equation (14) fully defines r for both 100 Mbps and 1 Gbps.

4.3.2 Model Correspondence

With r defined, we can utilize M and A to quantitatively evaluate Equations (3) and (7), respectively, to predict P for policies `demand` and `profile` at 100 Mbps and 1 Gbps.

Figure 4 plots the *cumulative delay* - difference in actual playback and memory-cached playback $(P(t) - M(t))$ - (y-axis) against time (x-axis), for one sample VM/workload combination: an Ubuntu VM running the latex compile and view workload (a mix of batch and interactive operations). `demand` plots are identified with triangle markers,

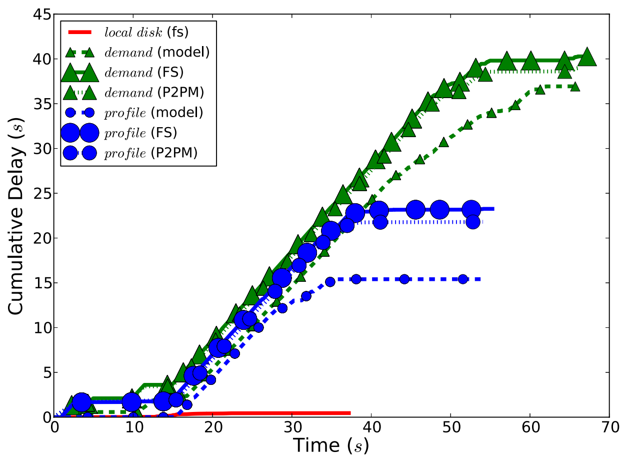


Figure 4: Cumulative delay vs. time: Ubuntu/Latex.

while **profile** plots are identified with circle markers. The model-predicted curves are those with the smallest respective marker size.

The first item to note is that the model appears to more tightly bound **demand** than **profile**. The second is that while the both predicated curves under-estimate the delay seen experimentally (largest marker curves) by several seconds, the curve shapes themselves match quite closely.

This first observation can be easily explained by recalling that **profile** cannot predict the playback window perfectly. As Equation (7) assumes perfect prediction, it is no surprise that **profile**'s actual performance is further from that indicated by the model, than that of **demand** for which no such discrepancy exists.

With respect to the second observation, the difference between measurements and model predictions could have originated from additional delays introduced either on the client-side, in the network, or on the server. To provide some visibility, we instrumented VMTORRENT to track both the delays seen by the FS and P2PM components respectively. If delay was being introduced by VMTORRENT itself (as opposed to delays introduced due to CPU/RAM sharing between VMTORRENT and the hypervisor), this is where it would appear.

The medium and large marker symbols denote delays respectively seen by FS and P2PM in Figure 4. Examining these, it is immediately apparent that overhead induced inside of the VMTORRENT implementation can be seen in the small gap between these curves by the end of execution, but also, that this overhead is relatively modest for a research prototype (∼2 s). For **demand**, the unexplained difference between model and P2PM is roughly another 3 s, which we find more than tolerable for a 66 s run (being ∼5% error).

4.3.3 Access Patterns and Performance Bounds

Having validated our model's single peer predictions against experimental results, we now use that model to provide insight into the differing performance produced from 100 Mbps and 1 Gbps networks. Figures 5-7 plot model-predicted VM playback (execution) progress (y-axis) over time (x-axis). Along with the memory-cached playback M, we plot both **demand** and **profile** policies for both network rates. We produce one plot for each VM, all of which run the same Boot-Shutdown workload.

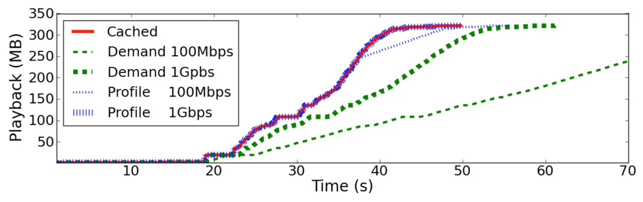


Figure 5: Fedora Playback, 100 Mbps,1 Gbps.

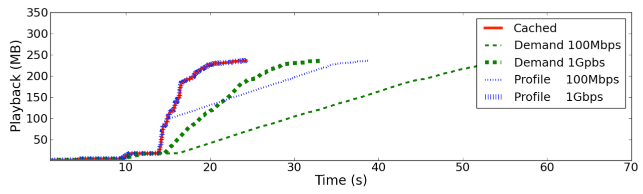


Figure 6: Ubuntu Playback, 100 Mbps,1 Gbps.

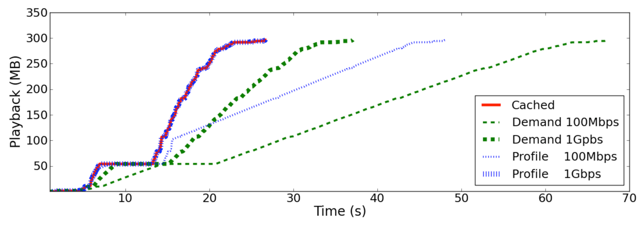


Figure 7: Win7 Playback, 100 Mbps,1 Gbps.

Several features in these plots are noteworthy. Without profiling, even a 1 Gbps network provides insufficient bandwidth for **demand** to stay close to local caching. However, with profiling a 1 Gbps download rate can enable **profile** to achieve essentially ideal performance (despite misprediction). Finally and most interestingly, the relative performance of **demand** at 1 Gbps and **profile** at 100 Mbps depends on M. When M spikes sharply early-on and then flattens, as it does in the boot section of both the Ubuntu and Win7 VM images, there is relatively little opportunity for prefetching to fill the buffer before the spike, after which it just plays catch up on cache misses. Thus the dominant performance factor becomes the rate at which cache misses can be filled, giving **demand** at 1 Gbps a distinct advantage. Conversely, when M increases more gradually and at a latter time - as it does for the Fedora VM - a large playback buffer can be prefetched, and if prefetch is reasonably accurate it trumps the 10x difference in network speed.

4.4 Scalability

We now proceed to our primary concern, scalability. We show the results of running: **p2p_p**, **p2p_d**, **cs_d** and **local** where a number of clients attempt to execute a VM workload. We find that **p2p_p** both performs with roughly the same efficiency as **local** and that it scales far better than the respective current standard and state-of-the art solutions **p2p_d** and **cs_d**.

All experiments consist of a dedicated seed with a complete copy of disk image cached in memory and a set of peers running VMTORRENT instances.

4.4.1 100 Mbps

We begin by exploring performance on a 100 Mbps network. Figure 8 plots mean normalized run time $P^{-1}(A)/M^{-1}(A)$ (y-axis) against the number of peers (swarm size) (x-axis) **p2p_p** with a large profile (large circle marker), p2p_p with a small profile (small circle), p2p_d (square), cs_d (triangle) and local (dashed line). Additionally, model curves are plotted with decreasing width bands for cs_d, p2p_p, and p2p_d. Each sub-figure corresponds to a combination of VM and workload[7] described in Section 2.2.3.

Examining the cs_d plot lines, clear scalability issues have already arisen by $n = 4$ at 100 Mbps and appear to grow linearly with the swarm size, resulting in run times 40-70x greater than the ideal memory-cached playback time (A) by $n = 100$. Contrastingly, p2p_p run times for $n = 100$ are only two to three times A.

The effect of utilizing randomized prefetch in combination with P2P swarming is likewise undeniable. Naive use of P2P by p2p_d scales significantly better than cs_d, but far worse than p2p_p, which benefits from far greater piece diversity. The performance advantage of p2p_p is best measured in orders of magnitude: **4-11**x better than p2p_d and **16-30**x better than cs_d.

Profile size (and implicitly prediction accuracy) appears to be a secondary factor in determining scalability. Both small and large profiles perform essentially the same with respect to scaling, indicating that as n grows large, ensuring piece diversity is the key to scalability, not perfect playback window prediction. This result is also encouraging in that it demonstrates a small amount of profiling can go a long way.

We observe the small gap between p2p_p and local on both the Ubuntu and Win7 VMs across all workloads. On the Fedora VM this is even more pronounced with VMTORRENT significantly outperforming execution from local. This occurs for the same reason noted in Section 4.3.3: Fedora's boot-and-login access pattern is very prefetch-friendly, giving p2p_p a leg up on local. Moreover, here the effect is multiplied since p2p_p peers have long period in which to prefetch. While prefetching p2p_p randomizes piece requests which increases effective bandwidth, and thereby scalability, as predicted by Equation 12. However, when p2p_p hits the catch-up phase, only demand requests are made (due to our design decision to give cache misses absolute priority over prefetches), reducing the piece diversity on which P2P scalability relies. With its relatively long and gradual access curve, the Fedora VM maximizes p2p_p's scalability compared to both p2p_d and cs_d (Note the large gap in Figure 8(a) between p2p_p and p2p_d/cs_d).

Finally, we turn to our model predictions. All model parameters needed for the model curves have been defined in Section 4.3.1, save one - W. For cs_d, we note that even 100 nodes should be well below the server congestion threshold. Thus we set

$$W_{cs}(n) = RTT_{net} + Q\,.$$

For p2p_d, our intuition suggests that piece diversity will decline proportionally to the increase in swarm size

$$W_{p2p}(\texttt{demand}, n) = RTT_{net} + Qn$$

while for p2p_p, this decline should be less than proportional, but higher than logarithmic (since randomized prefetch

[7]We ran into problems automating large scale experiments on Windows 7 for both cs_d and p2p_d. The reason for this is that Windows 7 contains an undocumented auto-login timeout that we could not disable. Consequently, when delay exceeded a certain threshold (roughly 3x memory cached), auto-login would fail causing the need for manual intervention, which was not feasible on large test populations and skewed results for p2p_d at $n = \{4, 8\}$.

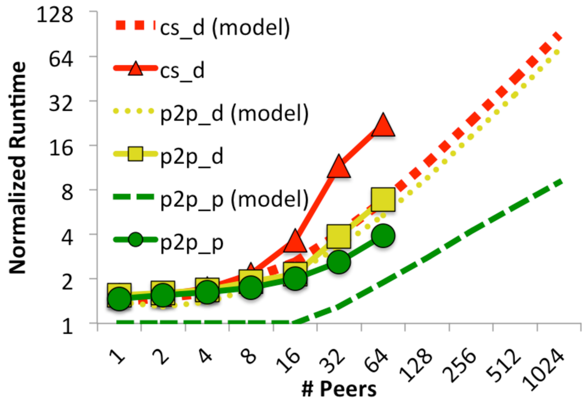


Figure 9: Normalized runtime vs. swarm-size (1 Gbps).

is strictly superseded by cache misses, limiting opportunity to generate piece diversity)

$$W_{p2p}(\texttt{profile}, n) = RTT_{net} + Q\sqrt{n}\,. \qquad (15)$$

we can see that these simple functions capture the scalability dynamics reasonably well, although for $1 < n < 16$ piece diversity is clearly over-estimated by Equation (15) - as seen by the discrepancy in p2p_p model and observed across all three graphs.

4.4.2 *1 Gbps*

We ran 1 Gbps experiments on the VICCI testbed. Figure 9 plots normalized playback $(P^{-1}(A)/M^{-1}(A))$ against peer size for the Ubuntu VM/Boot-Shutdown workload. Examining the cs_d plot lines, scalability issues become apparent by $n = 16$ at 1 Gbps (4x the number of peers for which scalability became an issue at 100 Mbps).

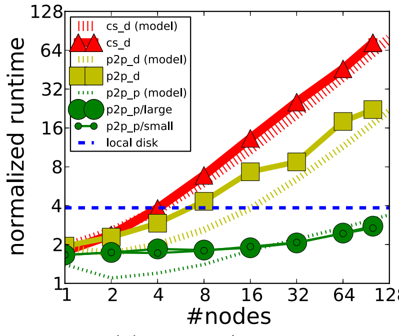
Here we see that while the model curves capture the rough shape of scalability, there are significant discrepancies. The first major discrepancy is that p2p_p's actual performance is roughly 1.5x slower than predicted by the model (although here constraint $P(t) \leq M(t)$ decreases the impact of Equation (15)'s early over-estimation of piece diversity). Secondly, cs_d plots sharply higher than expected.

A study of the logs suggests this is caused by several peculiarities of VICCI not possessed by Emulab. Recall from Section 4.1 that VICCI utilizes virtual containers that share both hardware and network resources among multiple investigators. Apparently, VICCI also suffers from irregularities in its CPU assignment algorithm, lowering performance and exacerbating already variable testbed behavior.

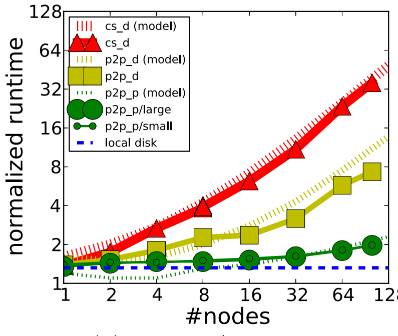
Consequently, given that we do see a correspondence here in shape and scaling, we conclude that our model curves provide a reasonable indicator of 1 Gbps performance. Based on this, we provide extrapolations for up to 1024 nodes and find that p2p_p again provides significantly more scalability than current approaches. However, at $8A$ this performance leaves significant room for improvement. Thus a main challenge for future work is to develop new policies that support better swarming efficiency - likely by relaxing our restriction that cache-misses take absolute priority over prefetches.

4.5 Swarming Efficiency

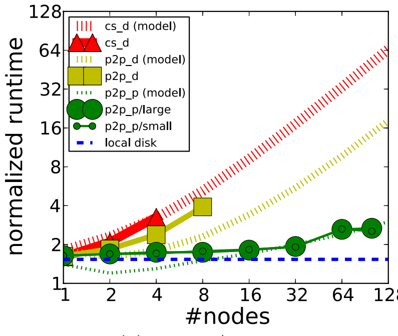
We now study the swarming efficiency on the Ubuntu VM running the Boot-Shutdown workload at 100 Mbps. We measure swarming efficiency for p2p_p directly by plotting



(a) Fedora/Latex.



(b) Ubuntu/DocEdit.



(c) Win7/Music.

Figure 8: Normalized run time vs. swarm-size (100 Mbps).

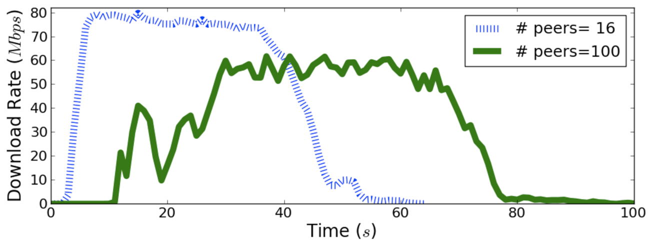


Figure 10: Swarming efficiency.

peer download rate r over time for swarm sizes $n = 16, 100$ in Figure 10.

Examining Figure 10, we see the average peer download rate hit a peak of roughly 80 Mbps after 6 s and remain relatively stable thereafter (until a drop-off when the profile is exhausted and VMTORRENT transitions to `demand`). Contrastingly, the behavior shown for 100 peers differs greatly. Firstly, the peak rate drops to 60 Mbps - a 25% drop from 16 peers. Secondly, the startup period required to attain the peak rate takes almost 6x longer than for the 16 peer swarm. Finally, the rate fluctuates significantly more in the startup phase.

These observations provide additional evidence that the non-optimal scaling behavior stems from our decision to give cache-misses absolute priority over prefetches. With 16 peers, the server still has more than enough capacity to fill all demand requests while still servicing prefetch requests. However, by 100 peers, it appears that the server must often starve prefetch requests in order to keep up with demand requests - leading to a slower, choppier buildup of the piece diversity needed to enable peak efficiency, and ultimately achieves a lower peak rate than that of 16 peers.

5. RELATED WORK

There are only a few studies of efficient on-demand deployment of virtual appliances or machines in the *local area network (LAN)*. The first set of related work presented focuses on efficiently *migrating* - transferring and executing individual VM images, while the second set is predominantly concerned with *reduplication* - scalably distributing copies of VM images.

5.1 Migration

Internet Suspend/Resume (ISR) [23] focuses on the related problem of *VM migration* - suspending a VM on one hardware platform, transferring that VM over the WAN, and resuming its execution at another location. While migration faces none of the scalability concerns posed by our problem of replicated deployment, latency until execution can proceed is of critical importance in both their domain and ours. Their *pure demand-fetch* inspired the on-demand components of both [7] and our own work. Like our own work, ISR also leverages over-the-network pre-fetch, warming the cache by transferring the migrating VM's *working set*. However, this *working set* is a reactive mechanism, simply being composed of those blocks recently accessed in the particular execution being migrated. It provides no help in warming the cache *for tasks a user has yet to do*, since those blocks lie outside of the *working set*. Clark *et al.* [8] extends this working set concept to that of a *write-able working set* in order to reduce the time needed to *stop-and-copy* the migrating VM image. Contrastingly, the Collective [6], a server-based system delivery of managed appliances to personal computer (PC) users, takes a proactive approach to pre-fetching VMs in order to reduce startup and execution delays. The Collective pre-fetches and fills its cache with the most frequently accessed blocks, taken over all appliances. Our technique takes this approach a step further, *predicting* future block accesses based on profiles of block access patterns. These profiles are built by observing previous executions of tasks similar to that being executed by the VM.

Post-copy migration [15] provides another interesting point of comparison. Instead of pre-copying a VM's in-memory working set in multiple rounds, post-copy migration immediately begins the VM running on the target machine and retrieves cache misses across the network (the image itself is stored on the network and thus does not need to be transferred). With a good prediction algorithm, post-copy could proactively prefetch those portions of the working set which were most likely to be needed, reducing execution delay. Our FS and profiling components mirror this strategy, but instead of prefetching pages from a working set, we prefetch pieces from an image file.

More generally, we note that prefetching is not a novel idea. Prefetching has been used to improve storage and file systems for many years. Known techniques rely on leveraging past accesses [5], application hints [30], or training prefetching parameters using sample traces [22], to predict future accesses. However, previous work in this space focus on finding the right trade-off between prediction accuracy and low (CPU and memory) overhead and exploiting domain-specific characteristics, *e.g.,* parallelism in storage systems. Contrastingly, our scenario is far less constrained by memory and focused on specific VM workloads, allowing us to build more comprehensive and targeted access profiles. Further, our work focuses on not just prefetching, but the combined problem of prefetching and data distribution across peers to maximize performance - making predication accuracy of lesser importance.

5.2 Reduplication

The most straightforward approach to optimizing VM deployment without use of additional hardware is to sequentially copy VM images to the target nodes, instead of downloading them in parallel. This approach is common in data centers, which employ provisioning servers to distribute and execute pre-customized images on-demand [26, 34]. However, sequential distribution can lead to long distribution times and network hotspots when VM demand is high.

The work on the Collective evolved into the commercial solution MokaFive [27]. While the internal details of MokaFive are not available, based on publicly available materials, we suspect scalability is achieved with this approach using a combination of hardware over-provisioning and modifying VM image and hypervisor - both of which are proprietary.

The Snowflock implementation [24] of the VM fork abstraction provides for highly efficient and scalable cloning of VMs by building a custom application-level multicast library (*mcdist*) into a modified Xen hypervisor. To obtain such performance, their approach also calls for use of modified VM images, guest OSes, and applications. Finally, the *mcdist* library runs on top of IP-multicast, although they note that to the best of their knowledge cloud providers do not currently enable IP-multicast. Contrastingly, our approach is minimally invasive, requiring no changes to network, hypervisor, VM image, guest, or applications.

IP multicast [19, 20] provides a truly scalable solution for VM image delivery [33] - but also has several drawbacks. Multicast is not geared towards on-demand image distribution since different peers will need only partially overlapping sets of data (and on different schedules), and is not well suited to delivering data to geographically distributed networks, *e.g.,* multi-region data centers or corporations. Even in a single network, multicast has significant setup overhead and is not used by many organizations as a result [12, 17]. More recently, when Etsy attempted to utilize IP multicast functionality to distribute large files within their network via rsync "multicast traffic saturated the CPU on [Etsy] core switches causing all of Etsy to be unreachable", which led them to utilize a P2P solution instead [13].

P2P provides an alternative to IP multicast that sidesteps synchronization and setup issues. We are not the first to apply P2P techniques to VM delivery. However, we have taken a more nuanced approach in the application of P2P, for which performance delays are best measured in seconds, instead of minutes or hours.

Zhang *et al.* [41] proposes a play-on-demand solution for desktop applications. The basic idea is to store user's data at a USB device, and at run time, download desktop applications using P2P (specifically, via unmodified BitTorrent [9]). These applications are then run in a lightweight virtualization environment. The downloaded images here are not standalone VMs, making this approach of more limited applicability. These images are orders of magnitude smaller than those of VMs, consequently a naive application of P2P provides adequate performance and scalability. However, when this same approach is used to distribute VMs to student machines in a training environment [28] both performance and scalability suffer - distribution taking 1-4 hours for a 22 machine deployment on a 100 Mbps network. Chen *et al.* [7] adds on-demand download to naive P2P which improves performance by an order of magnitude on a somewhat larger number of machines. Yet in even fairly small deployments, the wait until a VM can even begin execution takes more than 20 minutes - still not nearly good enough for deployments where time-to-use is critical. The *lxcloud* project at CERN used a customized BitTorrent client to deploy almost 500 10GB VM images in 23 minutes, without on-demand download [38]. However, their base network of 1 Gbps was 10*x* faster than those used by [28, 7]. A general purpose P2P distributed file system such as Shark [4] or CFS [11] could be expected to provide similar performance in provisioning VM images.

Contrastingly, vmTorrent focuses on streaming performance, instead of full-download. By incorporating profile-based prefetch, vmTorrent can fully execute VM tasks in a fraction of the time required by even the best of previous P2P approaches, and do so at scale. However, doing so is not trivial. To obtain efficient piece exchange, vmTorrent seeks to balance the immediate download requirement of the VMM with maintaining a high level of piece diversity in the system. This approach to creating diversity is inspired by P2P video streaming systems which use a sliding window to download pieces needed for immediate playback, while still acquiring non urgent pieces for diversity [37, 42].

In a complementary approach [31] examines the distribution of VMs within several small-to-medium IBM datacenters, finding a high level of similarity in VM content. Their VDN system hashes blocks and then caches them according to hash value, enabling multiple related VM-images to share the same content block. More efficient utilization of the cache fronting their image store, provides for significant performance improvement. Also related is Twitter's Murder system [36] which dramatically cuts down the distribution time for software binaries by optimizing BitTorrent for the data-center. Again, their techniques are complementary to our own and could be used to improve the performance of our BitTorrent backend which is based on the WAN-optimized libtorrent library.

6. CONCLUSIONS

We have presented the design, implementation and evaluation of vmTorrent. vmTorrent decouples P2P-delivery from stream presentation - allowing applications to easily use P2P-streamed data. Doing so enables scalable VM image streaming - based on a combination of novel *profile*-based prefetching and *piece selection* policies that balance cache misses against swarm efficiency. Both our analytic models and experimental evaluation show vmTorrent outperforms current techniques by orders of magnitudes

7. ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-1018355, CNS-0905246, CNS-1162021, CNS-1162447, CNS-1017934, and CCF-0964497.

8. REFERENCES

[1] IOMEGA Solid State Drive to Address Boot Storm Issues. Retrieved from `http://iomega.com/about/prreleases/2011/20110526_vdi_bootstorm.html`, May 2011.

[2] Fusion-io VDI Overview. Retrieved from `http://www.fusionio.com/overviews/fusion-io-virtual-desktop-infrastructure-vdi-overview/`, 2012.

[3] Bindfs: Mount a Directory to Another Location and Alter Permission Bits. Retrieved from `http://code.google.com/p/bindfs`, n.d.

[4] S. Annapureddy et al. Shark: Scaling File Servers via Cooperative Caching. In *USENIX NSDI*, May 2005.

[5] S. H. Baek and K. H. Park. Prefetching with Adaptive Cache Culling for Striped Disk Arrays. In *USENIX ATC*, June 2008.

[6] R. Chandra et al. The Collective: A Cache-Based System Management Architecture. In *USENIX NSDI*, May 2005.

[7] Z. Chen et al. Rapid Provisioning of Cloud Infrastructure Leveraging Peer-to-Peer Networks. In *IEEE ICDCS Workshops*, June 2009.

[8] C. Clark et al. Live Migration of Virtual Machines. In *USENIX NSDI*, May 2005.

[9] B. Cohen. Incentives Build Robustness in BitTorrent. In *P2PECON*, June 2003.

[10] L. P. Cox et al. Pastiche: Making Backup Cheap and Easy. In *USENIX OSDI*, December 2002.

[11] F. Dabek et al. Wide-Area Cooperative Storage with CFS. *ACM SOSP*, October 2001.

[12] A. El-Sayed et al. A Survey of Proposals for an Alternative Group Communication Service. *IEEE Network*, Vol. 17, January/February 2003.

[13] Etsy. Turbocharging Solr Index Replication with BitTorrent. Retrieved from `http://codeascraft.etsy.com/2012/01/23/solr-bittorrent-index-replication`, January 2012.

[14] R. Harris. OpenStack Proposal Blueprint: Download Images using BitTorrent (in XenServer). Retrieved from `https://blueprints.launchpad.net/nova/+spec/xenserver-bittorrent-images`, June 2012.

[15] M. R. Hines and K. Gopalan. Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning. In *ACM VEE*, March 2009.

[16] R. Hirschfeld. OpenStack Deployments Abound at Austin Meetup. Retrieved from `http://www.openstack.org/blog/2011/12/openstack-deployments-abound-at-austin-meetup-129/`, December 2011.

[17] M. Hosseini et al. A Survey of Application-Layer Multicast Protocols. *IEEE Communications Surveys & Tutorials*, Vol. 9, Third Quarter 2007.

[18] Y. Huang et al. Challenges, Design and Analysis of a Large-scale P2P-VoD System. In *ACM SIGCOMM*, August 2008.

[19] IANA. RFC 3171, 2001.

[20] IANA. RFC 4291, 2006.

[21] C. B. M. III and D. Grunwald. Content-Based Block Caching. In *IEEE MSST*, May 2006.

[22] T. Kimbrel et al. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *USENIX OSDI*, October 1996.

[23] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *IEEE Hotmobile*, February 2002.

[24] H. A. Lagar-Cavilla et al. SnowFlock: Virtual Machine Cloning as a First-Class Cloud Primitive. *ACM TOCS*, February 2011.

[25] libtorrent. libtorrent: C++ Bittorrent Library. Retrieved from `http://www.rasterbar.com/products/libtorrent/index.html`, n.d.

[26] R. Mietzner and F. Leymann. Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows and Services to Realize a Unified Provisioning Infrastructure for SaaS Applications. In *IEEE SERVICES*, July 2008.

[27] MokaFive. Retrieved from `http://www.mokafive.com/products/components.php`, n.d.

[28] C. M. O'Donnell. Using BitTorrent to Distribute Virtual Machine Images for Classes. In *ACM SIGUCCS*, October 2008.

[29] N. Parvez et al. Analysis of BitTorrent-Like Protocols for On-Demand Stored Media Streaming. In *ACM SIGMETRICS*, June 2008.

[30] R. H. Patterson et al. Informed Prefetching and Caching. In *ACM SOSP*, December 1995.

[31] C. Peng et al. VDN: Virtual Machine Image Distribution Network for Cloud Data Centers. In *IEEE INFOCOM*, July 2012.

[32] L. Peterson et al. VICCI: A Programmable Cloud-Computing Research Testbed. Technical report, Princeton University, 2011.

[33] M. Schmidt et al. Efficient Distribution of Virtual Machines for Cloud Computing. *Euromicro PDP*, February 2010.

[34] L. Shi et al. Iceberg: An Image Streamer for Space and Time Efficient Provisioning of Virtual Machines. In *IEEE ICPP Workshops*, September 2008.

[35] M. Szeredi. FUSE: File System in Userspace. Retrieved from `http://fuse.sourceforge.net/`, n.d.

[36] Twitter. Twitter's Murder Project at Github. Retrieved from `https://github.com/lg/murder`, n.d.

[37] A. Vlavianos et al. BiToS: Enhancing BitTorrent for Supporting Streaming Applications. In *IEEE Global Internet Symposium*, April 2006.

[38] R. Wartel et al. Image Distribution Mechanisms in Large Scale Cloud Providers. *IEEE CloudCom*, November 2010.

[39] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *ACM SOSP*, October 2001.

[40] B. White et al. An Integrated Experimental Environment for Distributed Systems and Networks. In *USENIX OSDI*, December 2002.

[41] Y. Zhang et al. Portable Desktop Applications Based on P2P Transportation and Virtualization. In *USENIX LISA*, November 2008.

[42] Y. Zhou et al. A Simple Model for Analyzing P2P Streaming Protocols. *IEEE ICNP*, October 2007.