SnowFlock: Virtual Machine Cloning as a First Class Cloud Primitive

H. Andrés Lagar-Cavilla
AT&T Labs Inc. – Research
and
Joseph A. Whitney, Roy Bryant, Philip Patchin, Michael Brudno and Eyal de Lara
University of Toronto
and
Stephen M. Rumble
Stanford University
and
M. Satyanarayanan
Carnegie Mellon University
and
Adin Scannell
GridCentric Inc.

A basic building block of cloud computing is virtualization. Virtual machines (VMs) encapsulate a user's computing environment and efficiently isolate it from that of other users. VMs, however, are large entities, and no clear APIs exist yet to provide users with programatic, fine-grained control on short time scales.

We present SnowFlock, a paradigm and system for cloud computing that introduces VM cloning as a first-class cloud abstraction. VM cloning exploits the well-understood and effective semantics of UNIX fork. We demonstrate multiple usage models of VM cloning: users can incorporate the primitive in their code, can wrap around existing toolchains via scripting, can encapsulate the API within a parallel programming framework, or can use it to load-balance and self-scale clustered servers.

VM cloning needs to be efficient to be usable. It must efficiently transmit VM state in order to avoid cloud I/O bottlenecks. We demonstrate how the semantics of cloning aid us in realizing its efficiency: state is propagated in parallel to multiple VM clones, and is transmitted during runtime, allowing for optimizations that substantially reduce the I/O load. We show detailed microbenchmark results highlighting the efficiency of our optimizations, and macrobenchmark numbers demonstrating the effectiveness of the different usage models of SnowFlock.

This research was supported by the National Science and Engineering Research Council of Canada (NSERC) under grant number 261545-3, a Strategic Grant STPSC 356747 - 07, a Canada Graduate Scholarship, and an Undergraduate Student Research Award; by the Canada Foundation For Innovation (CFI-CRC) under Equipment Grant 202977; by the National Science Foundation (NSF) under grant number CNS-0509004; by Platform Computing Inc; and by a generous equipment donation from Sun Microsystems Inc.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design – Distributed Systems; D.4.1 [**Operating Systems**]: Process Management – Multiprocessing Multiprogramming Multitasking

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Virtualization, Cloud Computing

---

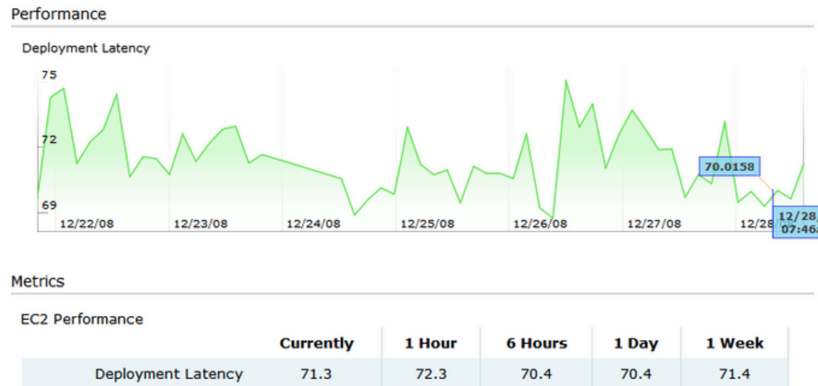
Cloud computing is experiencing a staggeringly fast rate of adoption. Users rent third-party computing infrastructure on a pay-per-use basis. They are not burdened with physical infrastructure maintenance and acquisition. Further, users can in principle scale their footprint on demand to cope with unanticipated peaks.

The tipping point in the cloud computing phenomenon can be attributed to the success of Amazon's EC2 service, which is a prime example of Infrastructure-as-a-Service (IaaS). The fundamental building block in an IaaS deployment is virtualization. By providing a narrow interface, virtualization allows for much greater security and isolation than commodity operating systems. Cloud providers need not maintain or deal with the internals of a user's Virtual Machine (VM), and the user is not burdened, outside of resource contention, by the existence of other users with alternate software stacks.

But virtualization comes at a price. VMs are rather large contraptions, encapsulating the entire user's environment. This expands beyond a specific application to include accessory pieces of state like auxiliary libraries, locales, general-purpose software, configuration structures for a full blown distribution, kernel drivers for a large diversity of hardware, UI environments, etc. The net result is VMs that typically accrue as many as tens of GB of state. Even so-called appliance VMs easily top a GB in disk size.

The work we present here is borne of the observation that the vast state encapsulated by a VM is a fundamental obstacle to fast VM spawning and scalability. Although "elastic" and "on-demand" are terms frequently used when referring to the scalability abilities of cloud computing, the reality is far from being as on-demand or elastic as it could be. Figure 1 shows a sample of VM instantiation latency in seconds for EC2. We harvest a key insight from this figure: VM spawning takes well over a minute and it is not particularly reliable in its latency. To spawn a VM, its state must be "swapped-in", and although there is a wide array of engineering tricks one can use to speed this up, a crucial obstacle remains: the large state footprint of a VM. In spite of the (technical) ease with which cloud providers can scale their disk, processor, and memory capacity, scaling the network capacity remains a fundamental obstacle – large transfers of VM state exacerbate this problem. Users thus rely on over-provisioning and load prediction to avoid hitting the scalability bottleneck, over-burdening the infrastructure and failing to cater for swift spikes in demand.

In this paper we present SnowFlock, a cloud programming paradigm and a system that addresses this problem. With SnowFlock, VMs are cloned into multiple child VMs created in parallel. Child VMs inherit the state of the parent in terms of its memory and private storage contents. They run the same OS, the same pro-



	Currently	1 Hour	6 Hours	1 Day	1 Week
Deployment Latency	71.3	72.3	70.4	70.4	71.4

Latency in seconds for the week of December 21$^{\text{st}}$ to 28$^{\text{th}}$ of 2008. Obtained from `http://cloudstatus.com`.

Fig. 1. EC2 VM Instantiation Latency

cesses, they have the same stacks and heaps and state caches. Child VMs can thus readily participate in ongoing tasks such as SIMD computations or parallel jobs by exploiting shared state or configurations. If used in an straightforward manner, most of the relevant data, executables and configurations needed by a clone are already loaded in its memory by the time of creation.

SnowFlock thus aims to mimic the semantics of UNIX process forking to the extent possible in the world of VMs, and to leverage the familiarity programmers have with such semantics. Crucially, those semantics allow for highly optimized state distribution for clone or child VMs. State can be propagated on demand throughout the clone lifetime, resulting in instantaneous creation of clones since little state remains in the critical path of VM spawning. By deferring propagation into a "late-binding" operation, optimizations that elide the unnecessary transfer of state can be successfully applied. And by propagating in parallel state to all clones of a generation, significant savings are achieved in terms of harvesting temporal locality of memory references.

SnowFlock thus can create dozens of cloned VMs in less than a second. Those VMs can perform complex tasks such as stock pricing, sequence alignment or image rendering, with only a few seconds of extra overhead throughout their task. Cloud I/O is highly optimized, since most VM state is never transferred, or sent in parallel to all children. All of this is achieved without the need of specialized hardware, and by providing users with an intuitive and familiar programming model. In fact, in many cases the user need not be aware that she is using SnowFlock or even running in a cloud. We demonstrate how the use of popular parallel programming APIs such as MPI can successfully leverage SnowFlock benefits without exposing the user to any additional complexity at all – unmodified MPI binaries can leverage instantaneous and efficient cloud scalability. Finally, we show how classical web server stacks can leverage the power of SnowFlock-based VM cloning to seamlessly scale their footprint in a cloud deployment, and cater to spikes in loads and "flash

crowds." Again, this is achieved without demanding any changes to code and retaining binary compatibility.

Besides describing our work on the design and implementation of SnowFlock, in this paper we will show a careful microbenchmark-based evaluation highlighting how the key benefits are realized. We will demonstrate end-to-end efficiency through MPI-based and other macrobenchmarks from the scientific, rendering, finance, and bioinformatics domains. We will show the suitability of SnowFlock for web server scaling with SPECWeb-driven experiments using an Apache server. We will finally comment on the evolution of the SnowFlock project and future directions.

1. THE VM FORK ABSTRACTION

The VM fork [1] abstraction lets an application take advantage of cloud resources by forking multiple copies of its VM, that then execute independently on different physical hosts. VM fork preserves the isolation and ease of software development associated with VMs, while greatly reducing the performance overhead of creating a collection of identical VMs on a number of physical machines.

The semantics of VM fork are similar to those of the familiar process fork: a *parent* VM issues a fork call which creates a number of clones, or *child* VMs. Each of the forked VMs proceeds with an identical view of the system, save for a unique identifier (*vmid*) which allows them to be distinguished from one another and from the parent. However, each forked VM has its own independent copy of the operating system and virtual disk, and state updates are not propagated between VMs.

We deliberately strived to mimic as closely as possible the semantics of UNIX fork [Ritchie and Thompson 1974]. The rationale is to exploit the familiarity of programmers with the fork semantics, and their widespread use in many popular programming patterns. Figure 2 shows four programming patterns that exploit fork's explicit ability to automatically propagate the parent state to all forked children. Implicitly, as well, these four programming patterns exploit fork's ability to *instantaneously* replicate workers. The four patterns represent tasks which are likely candidates for adoption – or have already been adopted – in cloud computing environments.

A key feature of our usage model is the ephemeral nature of children. Forked VMs are transient entities whose memory image and virtual disk are discarded once they exit. Any application-specific state or values they generate (e.g., a result of computation on a portion of a large dataset) must be explicitly communicated to the parent VM, for example by message passing or via a distributed file system.

The semantics of VM fork include integration with a dedicated, isolated virtual network connecting child VMs with their parent. Upon VM fork, each child is configured with a new IP address based on its vmid, and it is placed on the parent's virtual network. This eliminates the chance of a clone disrupting a pre-existing network with IP address collisions. However, it requires the user to be conscious of the IP reconfiguration semantics; for instance, network shares must be (re)mounted after cloning. In principle, child VMs cannot communicate with hosts outside this virtual network. To relax this constraint, we provide a NAT layer to allow the clones to connect to certain external IP addresses. Our NAT performs firewalling

[1] We use "VM Fork" and "VM cloning" interchangeably throughout this paper.

(a) Sandboxing
```
state = trusted_code()
ID = VM_fork(1)
if ID.isChild():
 untrusted_code(state)
 VM_exit()
else:
 VM_wait(ID)
```

(b) Parallel Computation
```
ID = VM_fork(N)
if ID.isChild():
 parallel_work(data[ID])
 VM_exit()
else:
 VM_wait(ID)
```

(c) Load Handling
```
while(1):
 if load.isHigh():
  ID = VM_fork(1)
  if ID.isChild():
   while(1):
    accept_work()
 elif load.isLow():
  VM_kill(randomID)
```

(d) Opportunistic Job
```
while(1):
 N = available_slots()
 ID = VM_fork(N)
 if ID.isChild():
  work_a_little(data[ID])
  VM_exit()
 VM_wait(ID)
```

Fig. 2. Four programming patterns based on fork's stateful cloning. Forked VMs use data structures initialized by the parent, such as `data` in case (b). Note the implicit fork semantics of instantaneous clone creation.

and throttling, and only allows external inbound connections to the parent VM. This is useful to implement, for example, a web-based frontend, or allow access to a dataset provided by another party: in both cases clone VMs are allowed controlled and direct access to the outside world.

VM fork has to be used with care as it replicates all the processes and threads of the parent VM. Conflicts may arise if multiple processes within the same VM simultaneously invoke VM forking. For instance, forking a VM that contains a full desktop distribution with multiple productivity applications running concurrently is certainly not the intended use of VM fork. We envision instead that VM fork will be used in VMs that have been carefully customized to run a single application or perform a specific task, such as serving a web page. The application has to be cognizant of the VM fork semantics, e.g., only the "main" process calls VM fork in a multi-process application.

VM fork is concerned with implementing a fast and scalable cloning interface for VMs. How those clone VMs are mapped to underlying physical resources, scheduled, load balanced, etc, is up to resource management software. There is an abundance of commercial resource management software that is particularly adept at following workplans, observing user quotas, enforcing daily schedules, accommodating for downtimes, etc. All of these activities exceed the scope of VM fork, which aims to treat the resource management software as an independent and modular entity indicating where VM clones should be spawned.

One of the main objectives of VM Fork is to provide efficient "fan-out" of a task on top of a set of cloud physical resources. For this reason, the implementation we present in the following sections is optimized in favor of replicating VMs on top of different physical hosts. However, the resource management software may choose to allocate multiple child VMs on the same multi-processor physical host. While nothing limits this policy, we note that this is not our target scenario, and thus misuses to a large extent many of our implementation techniques (further, it could

be implemented using proven alternative methods [Vrable et al. 2005]). Instead, we recognize the intimate similarities between VM and process fork, and combine the two to optimize use of SMP hosts. Parent VMs are created with the maximum possible number of available virtual CPUs; the actual number of active virtual CPUs is specified by the user. VM replication is performed first, and process-level replication afterward. The set of replicated VMs span multiple physical hosts. On each clone VM, the quantity of virtual CPUs is adjusted to match the allocation of underlying physical CPUs on the host. These CPUs are then spanned by processes forked within each VM.

We also note that VM fork is not a parallel programming paradigm. VM fork is meant to efficiently expand a single VM into a set or cluster of identical VMs. This is desirable because it is easy to reason about identical VMs, and it is straightforward to adapt the notion of an instantaneous cluster of identical VMs to multiple frameworks that require scaling of computation. VM fork thus easily accommodates frameworks that provide parallel programming facilities, or sandboxing, or scaling of web servers, to name a few examples. However, VM fork does not provide in itself facilities for parallel programming, such as shared memory regions, locks, monitors, semaphores, etc. There is great potential for future work in exploring the synergy between VM fork and parallel programming libraries. We present some of that work in the next section.

Finally, VM fork is not a replacement for any storage strategy. There is a great wealth of knowledge and research effort invested into how to best distribute data to compute nodes for data-intensive parallel applications. As a result of the high variability in application data patterns and deployment scenarios, many high performance storage solutions exist. VM fork does not intend to supersede or replace any of this work. VM fork is concerned with automatically and efficiently creating multiple replicas of a VM, understanding the VM as the unit that encapsulates the code and execution. We thus envision a deployment model in which datasets are not copied into the disk of a VM but rather mounted externally. VM fork will thus clone VMs to execute on data which is served by a proper big-data storage mechanism: PVFS [Carns et al. 2000], pNFS [pNFS.com], Hadoop FS [Apache B], etc. We further reflect on the interesting challenges arising from this separation of responsibility in Sections 3.6 and 6.

2. SNOWFLOCK: USAGE MODELS

SnowFlock is our implementation of the VM Fork abstraction. In this section we will describe the SnowFlock API, and three different usage models that highlight the flexibility and usefulness of the VM Fork abstraction for very different applications.

2.1 API

Table I describes the SnowFlock API. VM fork in SnowFlock consists of two stages. First, the application uses **sf_request_ticket** to place a reservation for the desired number of clones. Such a reservation is obtained by interfacing with a resource manager to obtain a suitable set of physical machines which the owner of the VM can use to run clones on top of. To optimize for common use cases in SMP hardware, VM fork can be followed by process replication: the set of cloned VMs span multiple hosts, while the processes within each VM span the physical underlying cores. This

• **sf_request_ticket (n, hierarchical)**: Requests an allocation for n clones. If hierarchical is true, process fork will follow VM fork, to occupy the cores of SMP cloned VMs. Returns a ticket describing an allocation for $m \leq n$ clones.
• **sf_clone(ticket)**: Clones, using the ticket allocation. Returns the clone ID, $0 \leq ID \leq m$.
• **sf_checkpoint_parent()**: Prepares an immutable checkpoint of the parent VM to be used for creating clones at an arbitrarily later time. Returns the checkpoint C.
• **sf_create_clones(C, ticket)**: Same as **sf_clone**, creates clones off the checkpoint C.
• **sf_exit()**: For children ($1 \leq ID \leq m$), terminates the child.
• **sf_join(ticket)**: For the parent ($ID = 0$), blocks until all children in the ticket reach their sf_exit call. At that point all children are terminated and the ticket is discarded.
• **sf_kill(ticket)**: Parent only, discards ticket and immediately kills all associated children.

Table I. **The SnowFlock VM Fork API**

behavior is optionally available if the **hierarchical** flag is set. Due to user quotas, current load, and other policies, the cluster management system may allocate fewer VMs than requested. In this case the application has the option to re-balance the computation to account for the smaller allocation.

The ticket contains information regarding the number of clones which will actually be created, and the parameters for each clone, such as number of virtual CPUs and the private IP address. This information is used by callers to orchestrate the allocation of work to clones.

In the second stage, we fork the VM across the hosts provided by the cluster management system with the **sf_clone** call. When a child VM finishes its part of the computation, it executes an **sf_exit** operation which terminates the clone. A parent VM can wait for its children to terminate with **sf_join**, or force their termination with **sf_kill**. A decoupled version of the clone process is provided by **sf_checkpoint_parent** and **sf_create_clones** – we will comment on these two API calls in Section 2.3.

As we will show in Section 3.9, the API calls from Table I are available to applications via a SnowFlock client library, with C and Python bindings. The Python bindings are further wrapped into scripts. This allows for two direct usage models: one in which the source code of an application is modified to include calls to the SnowFlock API, and one in which VM cloning is directed via scripting.

2.1.1 *ClustalW: An Example of Direct API Usage.* ClustalW [Higgins et al. 1994] is a popular program for generating a *multiple alignment* of a collection of protein or DNA sequences. From our point of view, the ClustalW program can be split into three phases:

(1) Data structure initialization. Sequences are loaded into a matrix.

(2) Pair-wise comparison. Pairs of sequences are compared and each pair obtains a similarity score. Each pair can be processed in a completely independent manner.

(3) Result aggregation. All pair scores are collated and a multiple alignment is generated.

While the third phase could be parallelized with e.g. a message-passing library, the second phase is where the most payoff for parallelization can be found, as the set of pairs can be processed independently by N workers. Our modifications to ClustalW for SnowFlock-based parallelization thus focus on the second phase. Figure 3 shows the integration of API calls into the ClustalW code. After the data structures are initialized, cloning is invoked, and each child computes the alignment of a set of pairs statically assigned according to the clone ID. The result of each alignment is a similarity score. Simple socket code is used by the children to relay scores to the parent, which then joins the set of children, and executes the third phase.

We highlight three aspects of the ClustalW code modifications. First and foremost, they're fairly trivial and non-invasive. Second, replacing VM forking with process forking yields an equivalent parallel program confined to executing within a single machine. Third, the clones automatically inherit all the initialization work performed by the parent, and use it to score pairs of sequences without the need for any further interaction.

```
sequences = InitSequences(InputFile)
ticket = sf_request_ticket(n, hierarchical=true)
m = ticket.allocation
ID = sf_clone(ticket)
for i in sequences:
   for j in sequences[i+1:]:
      if ((i*len(sequences)+j) % m == ID):
         PairwiseAlignment(i, j)
if (ID > 0):
   RelayScores()
   sf_exit()
else:
   PairsMatrix = CollatePairwiseResults()
   sf_join(ticket)
BuildGuideTree(PairsMatrix, OutputFile)
```

Fig. 3. ClustalW Pseudo-Code Using SnowFlock's VM Fork API

2.1.2 *Application Scripting.* A large class of applications exhibit a SIMD behavior. The same code can be applied as a transformation to a large number of independent objects. Sequence alignment in bioinformatics, and frame rendering in the movie industry are two prime examples. These applications are commonly referred to as "embarrassingly parallel", since typically the only practical constraint to the amount of parallelism that can be extracted is the availability of compute resources. These applications thus represent an ideal use-case for cloud computing, as their runtimes can be reduced to mere seconds if allowed to quickly scale in the cloud.

Using SnowFlock for this class of applications is simple. Using the scripting interface, we can produce a straightforward workflow shell script that clones the VM and launches an application process properly parameterized according to the clone ID. Each application process could render a subset of frames in an animation, or price a subset of stocks of interest, etc. Results are dumped to temporary files which the clones send to the parent before reaching an **sf_exit** call. Once the parent

VM successfully completes an **sf_join**, the results are collated. In Section 4.3 we show five examples of this simple yet effective technique.

2.2 MPI-SF: Simplifying Adoption with a Parallel API

While the SnowFlock API is simple and flexible, it nonetheless demands modification of existing code bases, or at the very least scripting. Our SnowFlock-friendly implementation of the widely used Message Passing Interface (MPI) library, which we have dubbed *MPI-SF*, allows a vast corpus of unmodified parallel applications to use SnowFlock's capabilities.

MPI-SF is an instance of a broader principle: encapsulating the details of an emerging cloud API within a well-known parallel programming API. Parallel APIs implicitly (or sometimes very explicitly) expose the notions of node management and worker creation to the user. Cloud APIs duplicate much of this work. Many times, users wishing to scale their parallel codes to cloud settings need to learn the specific details of a cloud API which is not germane to parallel processing. Users need to manually instantiate, configure and maintain the virtual hosts in their cloud slice. Only then can they bring up the parallel API subsystem, and deal with a secondary layer of management. With MPI-SF we take the first step in demonstrating that this unnecessary management overhead and duplication of responsibilities can be eliminated, while retaining the effort invested in existing code bases.

Our implementation of MPI-SF replaces the task-launching subsystem of MPI libraries by one which leverages the SnowFlock API to create the desired number of clones. Appropriately parameterized MPI worker processes are started on each clone. Each worker uses unmodified MPI routines from then on, until the point of application termination, when the VMs are joined. MPI-SF ports of both mpich2 [ANL] and Open MPI [Gabriel et al. 2004] are available, and are binary-compatible with applications written in C, C++ and Fortran, without even the need for re-linking.

2.3 Load Balancer-Driven Server Scaling

In Section 2.1.2 we alluded to the SIMD behavior of several cloud-friendly applications. Servers can also be characterized as SIMD processes. Web servers execute common code for each connection they manage: executing a PHP script, serving a static file, etc. Application servers maintain persistent state on a per-connection or per-session basis and typically allocate one thread of execution to each session.

SnowFlock can provide several advantages to this application domain. Most notably, instantaneous scaling of a server running in the cloud can allow seamless adaptation to sharp spikes in activity, or so-called flash crowds. With the cloud computing model of pay for use, a SnowFlock-enabled adaptive server saves money for users. Additionally, shutting down idle servers would save electrical power and money for cloud providers, and statistical multiplexing could reduce the required size of datacenters. And if it could bring new resources online fast enough, it would also guarantee a consistent QoS, even in the face of flash crowds.

To demonstrate the benefits that a server can reap from leveraging the capabilities of SnowFlock, we exploit the existing building blocks for clustered web servers. Clustered web servers are typically structured in multiple tiers, managed by a

load balancer or frontend. The load balancer offers the external Internet-visible interface and redirects connections to one of many web server instances. These web servers can be seen as SIMD workers, and are to a large extent stateless. In some scenarios, a portion of the work is delegated to an application server. The server state is managed by a backend, a file system in simpler cases but otherwise a database centralizing data management.

The SnowFlock API is primarily *reflective* in that clone requests apply to the caller itself, following classical UNIX fork semantics. By extending the API into a *transitive* mode we allow the caller to trigger the cloning of a third party. In this fashion the load balancer, or server frontend, can dynamically grow or shrink its roster of web servers to cater for spikes in activity. A callback mechanism needs to be installed in the clonable VM in order to react to completely asynchronous VM forking events – by, for example, reestablishing connections to data backends.

A final extension to the API allows decoupling the clone activity into two stages. **sf_checkpoint_parent** creates a clone epoch or generation. It records the state of the parent VM into an immutable checkpoint and returns a handle. No allocation for clones is necessary, as the clone creation is a separate event. With a proper ticket allocation and a checkpoint handle, **sf_create_clones** can be invoked to finalize the cloning process. In this manner, multiple clones can be created at different points in time out of the same clone epoch, which could be an optimally configured web server instance.

We have built an Apache-based adaptive web server. The SnowFlock transitive API is used to trigger the cloning of a web server VM and respond to increases in load. In this paper we show a simple case that demands no changes whatsoever to an existing web server stack. Only a post-cloning callback, which is an OS configuration task, needs to be installed. This exercise demonstrates the usefulness of the technique and motivates future research in this direction.

3. IMPLEMENTATION OF THE SNOWFLOCK SYSTEM

In this section we describe the design rationale behind the SnowFlock system and its implementation of a fast and scalable VM fork primitive. The SnowFlock software comprises a set of modifications to the Xen [Barham et al. 2003] Virtual Machine Monitor (VMM), and is available as open source under the GPLv2 license [UofT].

The Xen VMM is a hypervisor-based system. The Xen hypervisor is a thin software layer running at the highest level of processor privilege and providing isolation and scheduling services for a number of “domains” or VMs. The hypervisor contains very little code outside memory management, virtual CPU (VCPU) scheduling and event delivery, and support for virtualization hardware like IOMMUs. The hypervisor offers a hypercall interface, similar to a classical OS syscall interface, although much narrower. Guest OSes are paravirtualized in that they are aware of the underlying hypervisor and the hypercall interface. Guest OSes execute at the lowest level of processor privilege (both kernel and user-space) and use hypercalls to have privileged services (such as mapping a page of memory) be executed by the hypervisor on their behalf. A privileged administrative domain dubbed domain 0 executes a complete Linux software stack, with the drivers for all hardware devices and a user-space control toolstack that performs tasks such

as VM creation or migration. Virtual I/O drivers are provided with a split-driver model. A frontend offering a generic interface (e.g. a block device or an Ethernet interface) executes inside the guest VM and shares pages of memory with a backend executing in domain 0, which reroutes the I/O to the actual hardware.

As we will detail in the following sections, the SnowFlock modifications entail all levels of the system: the hypervisor, the guest VM kernel, the domain 0 kernel, and the domain 0 user-space privileged utilities. The first SnowFlock prototype was based off the 3.0.3 version of the Xen VMM, it used a paravirtualized version of the 2.6.16.29 Linux kernel for both guest VMs and domain 0, and only supported 32bit VMs. The more recent version of SnowFlock supports 64bit VMs, is based off the 3.4.0 version of Xen, and uses paravirtualized versions of Linux 2.6.18 for guest VMs and 2.6.27.45 for domain 0.

We first summarize the rationale for our design decisions. We then discuss the four main components of the SnowFlock system. We study the storage and networking models of SnowFlock VMs. We also describe the orchestration tasks that a SnowFlock control stack needs to perform. We close this section by describing the implementation of the SnowFlock API and the different usage models.

3.1 Challenges and Design Rationale

Performance is the greatest challenge to realizing the full potential of the VM fork paradigm. VM fork must *swiftly* replicate the state of a VM to many hosts simultaneously. This is a heavyweight operation as VM instances can easily occupy several GB of RAM. While one could implement VM fork using existing VM suspend/resume functionality, the wholesale copying of a VM to multiple hosts is far too taxing, and decreases overall system scalability by clogging the network with gigabytes of data.

Figure 4 illustrates this by plotting the cost of suspending and resuming a one GB VM to an increasing number of hosts over NFS (see Section 4 for details on the testbed). As expected, there is a direct relationship between I/O involved and fork latency, with latency growing to the order of hundreds of seconds. Moreover, contention caused by the simultaneous requests by all children turns the source host into a hot spot. Despite shorter downtime, live migration [Clark et al. 2005; Nelson et al. 2005], a popular mechanism for consolidating VMs in clouds [Steinder et al. 2007; Wood et al. 2007], is fundamentally the same algorithm plus extra rounds of copying, thus taking longer to replicate VMs.

A second approximation to solving the problem of VM fork latency uses our multicast library (see Section 3.5) to leverage parallelism in the network hardware. Multicast delivers state simultaneously to all hosts. Scalability in Figure 4 is vastly improved. However, overhead is still in the range of minutes and several gigabytes of VM state are still transmitted through the network fabric. To move beyond this, *we must substantially reduce the total amount of VM state pushed over the network.*

SnowFlock, our fast VM fork implementation, is based on the following four insights: (i) it is possible to start executing a child VM on a remote site by initially replicating only minimal state; (ii) children will typically access only a fraction of the original memory image of the parent; (iii) it is common for children to allocate memory after forking; and (iv) children often execute similar code and access common data structures.

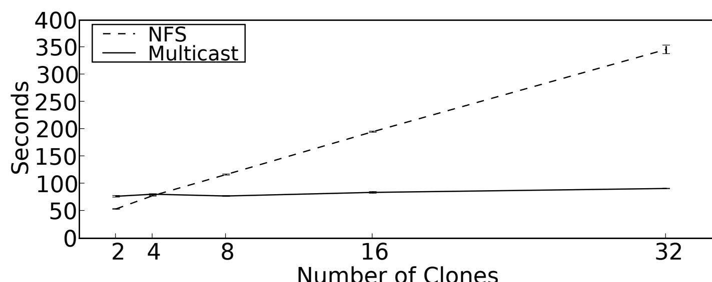


Fig. 4. Cloning a 1 GB VM by suspending and distributing the image over NFS and multicast.

The first two insights led to the design of *Architectural VM Descriptors*, a lightweight mechanism which instantiates a new forked VM with only the critical metadata needed to start execution on a remote site, and *Memory-On-Demand*, a mechanism whereby clones lazily fetch portions of VM state over the network as it is accessed. Our experience is that it is possible to start a child VM by shipping a minimal fraction of the state of the parent, and that children tend to only require a fraction of the original memory image of the parent. By propagating state lazily as execution progresses, it is possible to perform runtime optimizations that are not available to approaches that pre-copy all of the VM state. For example, it is common for children to allocate memory after forking, e.g., to read portions of a remote dataset or allocate local storage. This leads to fetching of pages from the parent that will be immediately overwritten. We observe that by augmenting the guest OS with *avoidance heuristics*, memory allocation can be handled locally by avoiding fetching pages that will be immediately recycled, thus exploiting our third insight. We show in Section 4 that this optimization can reduce communication drastically to a mere 40 MB for application footprints of one GB ($\simeq$4%). We further show this is not an isolated scenario but rather a common feature that can be exploited across several different workloads.

Compared to ballooning [Waldspurger 2002; Sapuntzakis et al. 2002], the SnowFlock approach is non-intrusive and reduces state transfer without altering the behavior of the guest OS. Even though the guest has to provide information to SnowFlock, its allocation decisions are not changed in any way, unlike with ballooning. This is of crucial importance because ballooning a VM down to the easily manageable footprint that our design achieves would trigger swapping and lead to abrupt termination of processes. Another non-intrusive approach for minimizing memory usage is copy-on-write, used by Potemkin [Vrable et al. 2005]. However, copy-on-write limits Potemkin to cloning VMs within the same host whereas we fork VMs across physical hosts. Further, Potemkin does not use multicast or avoidance heuristics, and does not provide runtime stateful cloning, since all new VMs are copies of a frozen template.

To take advantage of the fourth insight of high correlation across memory accesses of the children, and to prevent the parent from becoming a hot-spot, we *multicast* replies to memory page requests. Multicast provides scalability and prefetching: it may service a page request from any number of children with a single response, simultaneously prefetching the page for all children that did not yet request it. Our

design is based on the observation that the multicast protocol does not need to provide atomicity, ordering guarantees, or reliable delivery to prefetching receivers in order to be effective. Children operate independently and individually ensure delivery of needed pages; a single child waiting for a page does not prevent others from making progress.

Lazy state replication and multicast are implemented within the Virtual Machine Monitor (VMM) in a manner transparent to the guest OS. Our avoidance heuristics improve performance by adding VM fork-awareness to the guest. Uncooperative guests can still use VM fork, with reduced efficiency depending on application memory access patterns.

3.2 Architectural VM Descriptors

We observe that a VM suspend and resume cycle can be distilled to the minimal operations required to begin execution of a VM replica on a separate physical host. Further, in the case of Xen those minimal operations involve the setup of basic architectural data structures needed by any x86-based VM to function. We thus modify the standard Xen suspend and resume process to yield *Architectural VM Descriptors*, condensed VM images that allow for swift clone spawning.

Construction of an architectural VM descriptor starts by spawning a thread in the VM kernel that quiesces its I/O devices and issues a hypercall suspending the VM from execution. When the hypercall succeeds, a privileged process in domain 0 populates the descriptor. Unlike a regular Xen suspended VM image, the descriptor only contains architectural metadata describing the VM and its virtual devices: virtual CPU registers, age tables, segmentation tables, device specifications, and a few special memory pages shared between the VM and the Xen hypervisor.

The bulk of the VM descriptor is made out of the page tables of the VM. In the x86 architecture each process has a separate page table, although there is a high-degree of inter-process page table sharing, particularly for kernel code and data. The cumulative size of a VM descriptor is thus loosely dependent on the number of processes the VM is executing. Additionally, a VM descriptor preserves the Global Descriptor Tables (GDT). These per-processor tables are required by the x86 segmentation hardware, and Xen relies on them to establish privilege separation between the hypervisor and the kernel of a VM.

A page table entry in a Xen paravirtualized VM contains a virtual to "machine" address translation [Barham et al. 2003; Bugnion et al. 1997]. In Xen parlance, the machine address space is the true physical address space of the host machine, while physical frame numbers refer to the VM's notion of its own contiguous physical memory. A VM keeps an internal physical-to-machine table describing this additional level of indirection, and Xen maintains the reverse machine-to-physical table. When saving page tables to a descriptor, all the valid entries must be made host-independent, i.e. converted to VM-specific physical frame numbers. Certain values in the VCPU registers and in the pages the VM shares with Xen need to be translated as well.

The resulting architectural descriptor is multicast to multiple physical hosts using the multicast library we describe in Section 3.5, and used to spawn a VM replica on each host. The metadata is used to allocate a VM with the appropriate virtual devices and memory footprint. All state saved in the descriptor is loaded: pages

shared with Xen, segment descriptors, page tables, and VCPU registers. Physical addresses in page table entries are translated to use the physical-to-machine mapping at the new host. The VM replica resumes execution by returning from its suspend hypercall, and undoing the actions of its suspend thread: reconnecting its virtual I/O devices to the new backends.

We evaluate the VM descriptor mechanism in Section 4.1.1. In summary, the VM descriptors can be used to suspend a VM and resume a set of 32 replicas in less than a second, with an average descriptor size of roughly one MB for a VM with one GB of RAM.

3.3 Memory On Demand

Immediately after being instantiated from an architectural descriptor, the VM will find it is missing state needed to proceed. In fact, the code page containing the very first instruction the VM tries to execute upon resume will be missing. SnowFlock's memory on demand subsystem, *memtap*, handles this situation by lazily populating the VM replica's memory image with state fetched from the originating host, where an immutable copy of the VM's memory from the time of cloning is kept.

Memtap is a user-space process attached to the VM replica that communicates with the hypervisor via a shared memory page and a set of event channels (akin to software interrupt lines), one per VCPU. The hypervisor detects when a missing page will be accessed for the first time by a VCPU, pauses that VCPU and notifies the memtap process with the corresponding event channel. Memtap fetches the contents of the missing VM page, and notifies the hypervisor to unpause the VCPU.

The interactions between the hypervisor and domain 0 portions of memtap involve several aspects of interest we discuss in the following paragraphs: use of shadow page tables, use of a novel dynamic memory map abstraction, use of Copy-on-Write (CoW) on the parent side to preserve the integrity of the memory image, lockless shared-data structure access for efficiency, SMP-safety, and interactions with virtual I/O devices.

Use of Shadow Page Tables. To allow the hypervisor to trap memory accesses to pages that have not yet been fetched, we leverage Xen shadow page tables. In shadow page table mode, the x86 register that points to the base of the current page table is replaced by a pointer to an initially empty page table. The shadow page table is filled on demand as faults on its empty entries occur, by copying entries from the real page table. No additional translations or modifications are performed to the copied page table entries; this is commonly referred to as direct paravirtual shadow mode. Shadow page table faults thus indicate that a page of memory is about to be accessed. At this point the hypervisor checks if this is the first access to a page of memory that has not yet been fetched, and, if so, notifies memtap. We also trap hypercalls by the VM kernel requesting explicit page table modifications. Paravirtual kernels have no means outside of these hypercalls of modifying their own page tables, since otherwise they would be able to address arbitrary memory outside of their sandbox.

Dynamic Memory Map. Memtap uses the notion of a "dynamic memory map" to access the memory of a clone VM. The creation of a dynamic memory map is a new privileged operation we have added to the Xen hypervisor. First, a dynamic

memory map establishes mappings in a privileged domain 0 process (i.e. memtap) by batching all the page table modifications into a single round that is executed by the hypervisor. This is a substantial improvement over standard Xen, in which the mapping of each single guest VM page is a different hypercall. Second, and foremost, the hypervisor will keep track of the domain 0 page table pages that contain the relevant entries mapping the clone VM into the memtap user-space process. Whenever a "memory event" occurs, the hypervisor will automatically and synchronously patch the affected entries and flush TLBs. In this fashion the memtap process need only execute a single hypercall and rest assured it will have a consistent and persistent map of the clone's memory to which it can write fetched contents. In the case of memtap, the memory events of interest are modifications to the physical address space of the VM that add or steal pages, such as ballooning. The dynamic memory map will make sure the ballooned pages (which could go to any other domain) will not be accessible by memtap anymore.

CoW parent memory. On the parent VM, memtap implements a Copy-on-Write policy [Bugnion et al. 1997; Waldspurger 2002; Vrable et al. 2005] to serve the memory image to clone VMs. To preserve a copy of the memory image at the time of cloning, while still allowing the parent VM to execute, we use shadow page tables in log-dirty mode. All parent VM memory write attempts are trapped by disabling the writable bit on shadow page table entries. The memory server also uses a dynamic memory map to access the memory of the parent VM. However, the memory server dynamic memory map, upon a write fault, duplicates the page and patches the mapping of the server process to point to the duplicate. The parent VM is then allowed to continue execution, and modify its own copy of the page. By virtue of this CoW mechanism, a version of memory as expected by the clones is retained, and the total memory accounted to the parent VM grows by the number of pages duplicated via CoW. At worst, the VM's memory footprint will be X times larger, X being the number of clone generations simultaneously existing. In general, this number will be much lower (a similar behavior was studied in more depth by the Potemkin group [Vrable et al. 2005]). Finally, as different generations of clones are destroyed, the different dynamic memory maps are cleaned up and the corresponding CoW pages are returned to the system.

Efficiency. Memory-on-demand is supported by a single data structure, a bitmap indicating the presence of the VM's memory pages. The bitmap is indexed by physical frame number in the contiguous address space private to a VM. It is initialized by the architectural descriptor resume process by setting all bits corresponding to pages constructed from the VM descriptor. The Xen hypervisor reads the bitmap to decide if it needs to alert memtap. When receiving a new page, the memtap process sets the corresponding bit. The guest VM also uses the bitmap when avoidance heuristics are enabled. We will describe these in the next Section.

SMP Safety. Our implementation of memory-on-demand is SMP-safe. The shared bitmap is accessed in a lock-free manner with atomic (`test_and_set`, etc) operations. When a shadow page table write triggers a memory fetch via memtap, we pause the offending VCPU and buffer the write of the shadow page table. Another VCPU using the same page table entry will fault on the still empty shadow en-

try. Another VCPU using a different page table entry but pointing to the same VM-physical address will also fault on the not-yet-set bitmap entry. In both cases the additional VCPUs are paused and queued as depending on the very first fetch. When memtap notifies completion of the fetch, all pending shadow page table updates are applied, and all queued VCPUs are allowed to proceed with execution. We have successfully deployed 16-VCPU clones on 16-core machines with no deadlock or loss of correctness.

Virtual I/O Devices. Certain paravirtual operations need the guest kernel to be aware of the memory-on-demand subsystem. When interacting with virtual I/O devices, the guest kernel hands *page grants* to domain 0. A grant authorizes domain 0 to behave as a DMA unit, by performing direct I/O on VM pages. This behavior effectively bypasses the VM's page tables, and would not alert memtap of memory accesses to pages not yet fetched. To prevent domain 0 from reading undefined memory contents, we simply touch the target pages before handing the grant, thus triggering the necessary fetches.

3.4 Avoidance Heuristics

While memory-on-demand guarantees correct VM execution, it may still have a prohibitive performance overhead, as page faults, network transmission, and multiple context switches between the hypervisor, the VM, and memtap are involved. We thus augmented the VM kernel with two fetch-avoidance heuristics that allow us to bypass unnecessary memory fetches, while retaining correctness.

The first heuristic intercepts pages selected by the kernel's page allocator. The kernel page allocator is invoked when a user-space process requests more memory, typically via a `malloc` call (indirectly), or when a kernel subsystem needs more memory. The semantics of these operations imply that the recipient of the selected pages does not care about the pages' previous contents. Thus, if the pages have not yet been fetched, there is no reason to do so. Accordingly, we modified the guest kernel to set the appropriate present bits in the bitmap, entirely avoiding the unnecessary memory fetches.

The second heuristic addresses the case where a virtual I/O device writes to the guest memory. Consider block I/O: the target page is typically a kernel buffer that is being recycled and whose previous contents do not need to be preserved. If the full page will be written, the guest kernel can set the corresponding present bits and prevent the fetching of memory that will be immediately overwritten.

A key insight here is that these optimizations reduce the amount of state transmitted by taking advantage of the late binding made possible by on-demand propagation. Standard VM mechanisms for state transmittal pre-copy all the state of a VM. By propagating on demand we can optimize during runtime and based on future events, such as page allocations, in order to harvest opportunities for propagation avoidance.

In Section 4.1.3 we show the substantial improvement that these heuristics have on the performance of SnowFlock for representative applications. In summary, the descriptors and the heuristics can reduce the amount of VM state sent by three orders of magnitude, down to under a hundred MB when replicating a VM with one GB of RAM to 32 hosts.

3.5 Multicast Distribution

A multicast distribution system, *mcdist*, was built to efficiently provide data to all cloned virtual machines simultaneously. This multicast distribution system accomplishes two goals that are not served by point-to-point communication. First, data needed by clones will often be prefetched. Once a single clone requests a page, the response will also reach all other clones. Second, the load on the network will be greatly reduced by sending a piece of data to all VM clones with a single operation. This improves scalability of the system, as well as better allowing multiple sets of cloned VMs to co-exist.

To send data to multiple hosts simultaneously we use IP-multicast, which is commonly supported by high-end and off-the-shelf networking hardware. Multiple IP-multicast groups may exist simultaneously within a single local network and mcdist dynamically chooses a unique group in order to eliminate conflicts. Multicast clients[2] subscribe by sending an IGMP protocol message with the multicast group to local routers and switches. The switches then relay each message to a multicast IP address to all switch ports that subscribed via IGMP.

In mcdist, the server is designed to be as minimal as possible, containing only membership management and flow control logic. No ordering guarantees are given by the server and requests are processed on a first-come first-served basis. Ensuring reliability thus falls to receivers, through a timeout-based mechanism.

To the best of our knowledge, current cloud providers do not expose multicast capabilities to users. We believe this is due to an understandable concern for abuse and denial of service. We highlight that our use of multicast is confined to and controlled by the VMM. While outside the scope of this work, implementing quotas on top of our multicast facility is trivial. Thus, even for a user performing the equivalent of a "fork bomb" attack, the VMM can simply turn multicast for that user off, only affecting the performance of the clone VMs for that user.

We begin the description of mcdist by explaining the changes to memtap necessary to support multicast distribution. We then proceed to describe two domain-specific enhancements, lockstep detection and the push mechanism, we describe our flow control algorithm, and finish with the introduction of a final optimization called "end-game."

Memtap Modifications. Our original implementation of memtap used standard TCP networking. A single memtap process would receive only the pages that it had asked for, in exactly the order requested, with only one outstanding request, the current one, at a time.

Multicast distribution forces memtap to account for asynchronous receipt of data, since a memory page may arrive at any time by virtue of having been requested by another VM clone. This behavior is exacerbated in push mode, as described below. Furthermore, in an SMP environment, race conditions may arise when writing the contents of pages not explicitly requested: any VCPU may decide to employ the avoidance heuristics and use any of these pages without fetching them.

Consequently, in multicast mode, memtap batches all asynchronous responses until a threshold is hit, or a page that has been explicitly requested arrives. To

[2]In the context of this Section, the memtap processes serving child VMs are mcdist clients.

avoid races, all VCPUs of the VM are paused. Batched pages not currently present in the VM's physical space (due to e.g. ballooning), or for which the corresponding bitmap entry is already set, are discarded. The page contents are copied, bits are set, and the VCPUs unpaused. This mechanism is evaluated in Section 4.1.2.

A future work optimization to prevent full VM pausing is the use of "inter-domain spinlocks". A number of bits in a shared page can be designated as spinlocks to be accessed concurrently by memtap and the guest VM. The bits are accessed with atomic and cache-coherent bit setting operations that obviate the need for context switching. Before memtap writes a page, or before the guest VM marks a page as present, the corresponding spinlock bit is acquired, and only released afterward. Load is spread across the bits by uniformly hashing the different VM pages to different bits, increasing the opportunities for concurrency. However, benefits of this approach should be weighed against potential for inter-VCPU skew [Uhlig et al. 2004]: so far we have not seen any loss of correctness arise from individually blocking VCPUs in an SMP VM.

Lockstep Detection. Lockstep execution is a term used to describe computing systems executing the same instructions in parallel. Many clones started simultaneously exhibit a very large amount of lockstep execution. For example, shortly after cloning, VM clones generally share the same code path because there is a deterministic sequence of kernel hooks called during resumption of the suspended VM. Large numbers of identical page requests are generated at the same time.

When multiple requests for the same page are received sequentially, requests following the first are ignored, under the assumption that they are not the result of lost packets, but rather of lockstep execution. These requests will be serviced again after a sufficient amount of time, or number of requests, has passed.

Push. Push mode is a simple enhancement to the server which sends data proactively. Push assumes that the memory access patterns of a VM exhibit spatial locality in the physical address space – in Section 6.1 we describe our plans for overcoming the limitations of this assumption. With push mode, all VM memory state is eventually propagated to clones, except for those pages for which the heuristics have managed to prevent fetches. This is only useful for workloads that will eventually access the majority of the parent VM's state. While closer to standard live migration in its propagation of (almost) the entire footprint, clones are still instantiated immediately as opposed to waiting after all state has been copied.

Our algorithm works as follows: the server maintains a pool of counters; when a request for a page comes to the server, in addition to providing the data for that request, the server adds a counter to the pool. The new counter starts at the requested page plus one. When there is no urgent work to do, the counters in the pool are cycled through. For each counter the current page is sent, and the counter is then incremented. As pages are sent out by the server, through the counters mechanism or due to explicit requests, a corresponding bit is set in a global bitmap. No page is sent twice due to the automated counters, although anything may be explicitly requested any number of times by a client, as in pull mode. Experimentally, we found that using any sufficiently large number of counters (e.g., greater than the number of clients) provides very similar performance.

Flow Control. Sending data at the highest possible rate quickly overwhelms clients, who face a significant cost of synchronization and data writing. A flow control mechanism was designed for the server which limits and adjusts its sending rate over time. Both server and clients estimate their send and receive rate, respectively, using a weighted average of the number of bytes transmitted each millisecond. Clients provide explicit feedback about their current rate to the server in request messages, and the server maintains an estimate of the *mean* client receive rate. The server increases its rate limit linearly, and when a loss is detected implicitly by a client request for data that has already been sent, the rate is scaled back to a fraction of the client rate estimate. Throughout our evaluation of SnowFlock, we experimentally adjusted the parameters of this algorithm. We used a rate of increase of 10 KB/s every 10 milliseconds, and we scaled back to three quarters of the estimate client rate upon detection of packet loss.

End Game. The final optimization is the addition of an "end-game" state to the server. In end-game mode, the server only unicasts data responses. As the name implies, end-game mode is only entered after a substantial amount of page requests have been served, assuming most clients have all the state they need. The further assumption is that outstanding requests are either spurious, or due to receipt failures of individual clients. In any case, other clients will not benefit from implicit multicast-based prefetching. End-game mode can optionally be entered after a long period of client inactivity, assuming once again all clients have hoarded the majority of the state they need.

3.6 Virtual Disk

The virtual disk of a SnowFlock VM is implemented with a blocktap[Warfield et al. 2005] user-space driver running in domain 0. Multiple views of the virtual disk are supported by a hierarchy of CoW slices located at the site where the parent VM runs. Each clone operation adds a new CoW slice, rendering the previous state of the disk immutable, and launches a disk server process that exports the view of the disk up to the point of cloning.

Our CoW mechanism is implemented using sparse files for the actual CoW slice and per-slice bitmap files that indicate which disk pages (same 4KB granularity as memory) are present in the slice. We note that this simple mechanism is highly time-efficient at creating additional CoW layers. It is also highly space-efficient in terms of actual disk storage, as the only overhead is the very compact bitmap representation of metadata. Arbitrary deletion of internal CoW layers is performed safely but offline with a simple garbage collection mechanism. The only scalability concern is the inode count when reaching thousands of CoW slices, and the potential exhaustion of the file descriptor table by the parent disk driver. We believe the overall simplicity of our design is actually a relative strength compared to other CoW disk systems such as VHD [Microsoft 2009] or QCOW2 [McLoughlin 2008].

Child VMs allocate a sparse local version of the disk, which is filled with the state from the time of cloning fetched on-demand from the disk server. Based on our experience with the memory on demand subsystem, we apply similar techniques for the virtual disk. First, we use multicast to distribute disk state to all clones and exploit similarity in access patterns. Second, we devise heuristics complementary

to those used for memory. When the VM writes a page of data to disk, the previous contents of that page are not fetched. For simplicity and convenience, we chose the 4KB memory page size as the granularity of disk operations. No individual request to the disk subsystem will span more than a page-sized page-aligned chunk. CoW operations on the parent VM, data transfer from the disk server to children, and the discard-on-write heuristic are easily and efficiently implemented in this way. Although we have not tested it, this should also allow for memory swapping to disk to "just work."

Much like memory, the disk is replicated to clones "one way". In other words, the children and parent see the same state up to the point of cloning, but there is no sharing or write-through channels back to the parent. If sharing is desired at the file system level, it can be trivially implemented by using a distributed file system like NFS over the virtual private network. More specialized – and convenient – mechanisms, such as a key-value store for clones to indicate completion and results of assigned work, are left for future work.

The SnowFlock virtual disk is meant for internal private storage: libraries, executables, configurations and scripts. It is not the objective to provide a scalable cluster storage system for big data applications. In fact, we defer to such systems (e.g. Lustre, HadoopFS, or even basic NFS) the handling of the large input datasets needed by data-intensive applications.

For these reasons, the virtual disk is not heavily exercised in most cases by a SnowFlock child VM. Data intensive reads go to the appropriate shared storage backplane. Most other I/O operations result in little disk activity as they typically hit the in-memory kernel page cache. Further, thanks to the fetch avoidance heuristic previously mentioned, writes generally do not result in fetches. For all these reasons, our implementation largely exceeds the demands of many realistic tasks and did not cause any noticeable overhead for the experiments in Section 4.

CoW for VM disks was used by Disco [Bugnion et al. 1997]. We will leave the exploration of interactions with virtual disk infrastructures such as Parallax [Meyer et al. 2008] for future work. Parallax aims to support multiple VM disks while minimizing disk creation and snapshot time. Parallax's design resorts to aggressive Copy-on-Write sharing of multiple VM disk images in a cluster, and achieves this by relying on cluster-wide availability of a single (large) storage volume.

3.7 Network Isolation

Several aspects need to be taken care of for the network substrate of a clone VM. First, in order to prevent interference or eavesdropping between unrelated VMs on the shared network, either malicious or accidental, we implemented an isolation mechanism operating at the level of Ethernet packets, the primitive exposed by Xen virtual network devices. Before being sent on the shared network, the source MAC addresses of packets sent by a SnowFlock VM are rewritten as a special address which is a function of both the parent and child VM identifiers. Simple filtering rules are used by all hosts to ensure that no packets delivered to a VM come from VMs other than its parent or fellow clones. Conversely, when a packet is delivered to a SnowFlock VM, the destination MAC address is rewritten to be as expected by the VM, rendering the entire process transparent.

We implemented our translation and filtering layer in the netback driver, domain

0's backend driver for virtual network interfaces, for two reasons. First, netback already peers into the Ethernet header of all packets going through; the filtering and translation overhead is a minimal addition. Second, this allows us to not rely on, and not impose on the end-user, any specific networking subsystem for the VMs, such as iptables-based routing, ebtables-based translation, or the more recent Open vSwitch [Open vSwitch].

Two protocols need to be handled exceptionally as they include MAC addresses in their payload. These are the ARP and BOOTP (i.e. DHCP) protocols. It is worth noting that these are also the only broadcast messages a clone VM is authorized to emit. We use DHCP also as a means for automatically configuring the guest VM networking parameters. A small state machine executes within the netback driver intercepting DHCP messages and crafting appropriate DHCP replies. Due to the vast adoption of DHCP, this allows parent and clone VMs to be instantaneously and easily configured network-wise, regardless of the internals of the guest Linux distribution. For clone VMs, additionally, the network driver performs an automatic reconfiguration of its IP address upon resuming from cloning.

Because of the MAC-based isolation, SnowFlock VMs are essentially unable to talk to the outside world. A NAT layer can be used to allow controlled access to the Internet for these VMs. Our initial design used a small router VM. This represented a convenient centralization point for NAT, firewall and even traffic throttling rules. However, it also entailed a serious limit to performance, as all external traffic would be routed through a single point. More recently we have begun experimenting with applying the routing rules on each individual domain 0 host.

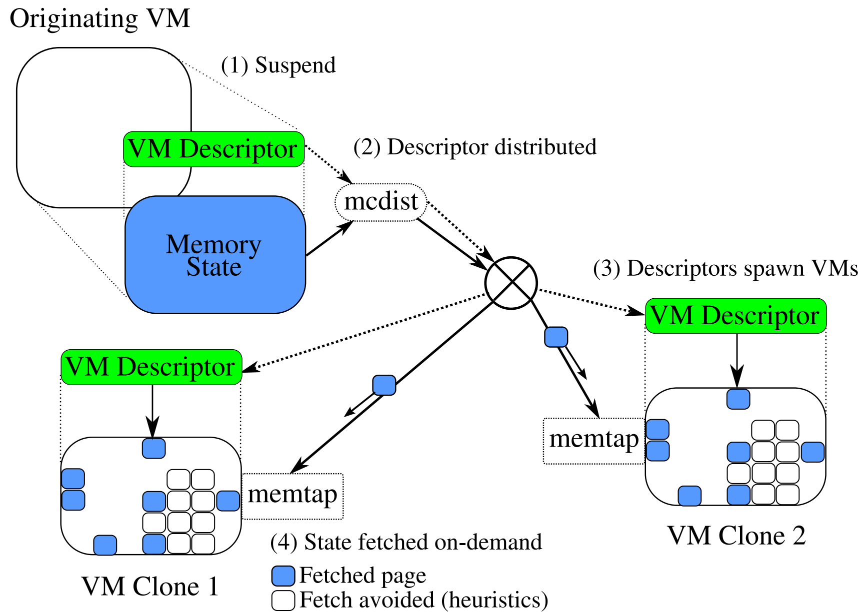
3.8 Putting it All Together: A Control Stack

So far we have described the individual building blocks that allow SnowFlock to efficiently clone VMs and distribute the memory state of the cloned children. A control stack is needed to orchestrate these mechanisms, and to expose them to client programs.

Ironically enough, over the two years of existence of the SnowFlock project, the control stack has been the least resilient software component. We clearly underestimated the scope of a control stack in this environment: it not only needs to clone VMs, but rather take on the holistic problem of managing VMs in a cluster or cloud setting.

Our control stack is thus composed of a persistent per-host daemon ("SnowFlock Local Daemon" or SFLD) that manages the state of the resident VMs and also officiates as the listening endpoint for SnowFlock client requests. This daemon runs in the domain 0 user-space and interfaces with the Xen management daemon (Xend) in order to create, destroy, and clone VMs. The juxtaposition of responsibilities between SFLD and Xend has led to a reformulation of SFLD in which the need for Xend is obviated, and all VM management activities are carried out by SFLD using libxenlight. This experimental version of SFLD has not been used in the evaluation section, though.

The different SFLDs in different hosts form in effect a distributed system that coordinates the task of cloning a VM. Figure 5 summarizes these tasks: creating the architectural descriptor and the memory server for a parent VM, distributing the architectural descriptor to all hosts using mcdist, launching each clone, and sup-



Condensed architectural VM descriptors are distributed to cluster hosts to spawn VM replicas. Memtap populates the VM replica state on demand, using multicast distribution. Avoidance heuristics reduce the number of necessary fetches.

Fig. 5. **SnowFlock VM Replication Architecture**

plying status messages to all VMs once finished. SFLDs also manage the creation of CoW slices for the parent VM disk, the creation of disk state servers, the setting up of sparse files for clone disks, the configuration of the appropriate network parameters for each clone VM, and the launching of the optional router VM.

We note that the decoupling extension to the SnowFlock API discussed in Section 2.3 is easily implemented by re-organizing a small amount of logic inside the SFLDs. Checkpointing a master involves generating the architectural descriptor, adding a CoW slice to the disk, and launching the memory and disk state servers. Clones are created at any time by finalizing the cloning process in a deferred fashion.

The SFLDs defer policy decisions to a centralized daemon that interfaces with an allocator system. The allocator performs resource accounting and the allocation of VMs to physical hosts. For hierarchical allocations, it decides the number of child VMs to be allocated and the number of physical processors assigned to each. This information is returned to callers via the API ticket. Suitable third-party cluster management software is used for the allocator. SnowFlock currently interfaces with Platform EGO [Platform Computing 2006] and Sun Grid Engine [Gentzsch 2001] – other possible options include Usher [McNett et al. 2007] or Moab [Cluster Resources]. Throughout Section 4 we use a simple internal resource manager which tracks memory and CPU allocations on each physical host.

SFLDs and its slave processes, such as memtap or the memory and disk state

servers, are lightweight processes. They spend most of their time asleep, use a a small amount of logic (in part due to mcdist's minimalistic design) to handle API requests, and mostly defer to the OS to fulfill their I/O requests. We have not observed any of these processes becoming a CPU or memory bottleneck, and thus have not had to bother about their impact on physical resource allocation.

3.9 API Implementation

The SnowFlock API presented in Section 2.1 is implemented by pickling the client requests, transforming them into string literals, and posting them to a well-known messaging endpoint. The SFLD monitors the messaging endpoint for each VM it manages, in order to capture and enact SnowFlock API requests. The messaging endpoint is a path in the XenStore database. XenStore is a shared-memory inter-VM messaging interface which is ideally suited for low-throughput control messaging. The SFLD unpacks the messages and later posts status replies back to the VM via the XenStore. The client performing the API request blocks waiting for this reply.

Client-side API bindings have been implemented in both C and Python. The C bindings are used by the MPI-SF ports to manage VM cloning appropriately. The Python bindings are also encapsulated in a set of Python scripts that allow for simple command line control of the cloning functionality and shell scripting.

3.9.1 *MPI-SF Implementation.* MPICH [ANL] and Open MPI [Gabriel et al. 2004] are the two main open source implementations of the Message Passing Interface (MPI) paradigm. When an MPI application is to be run, the mpirun program is used to initiate the execution as follows:

```
mpirun -np num_procs program arguments
```

where num_procs is the desired number of MPI processes or workers. MPICH and Open MPI differ in their process management planes, but both end up spawning num_procs processes in a number of different machines. The application executable and necessary data or configuration files are expected to be present in the same paths in all machines. When spawning the application binary, the process environment is set up with a number of variables which will be used by the MPI library upon start up. These variables include the MPI rank, an integral number uniquely identifying each worker in the MPI computation, and TCP/IP coordinates to reach the management plane or other application workers. Application processes link against the MPI library and are expected to call immediately upon startup MPI_Init to parse the environment variables. The application process can then perform calls such as MPI_Comm_rank to determine its rank, or MPI_Comm_size to find out about the total number of workers spawned. Throughout execution, workers transmit information to one another via the MPI_Send and MPI_Recv calls. Workers talk to one another using point-to-point connections, and identify each other by rank; application code has no knowledge of network coordinates. Connections between two workers are brokered by the MPI library and the management plane.

Open MPI is structured in a plugin architecture. Different plugins can be used for resource allocation, process spawning, etc. The simplest alternative uses a file with IP addresses of all available machines, and the SSH utility to spawn MPI

processes in other hosts. The mpirun program parses the file and controls the SSH connections, setting up the environment variables of the processes remotely spawned and watching their output for redirection and exception handling. Adapting Open MPI to SnowFlock demanded a set of modifications to the mpirun utility. VMs are forked and their private IP addresses, obtained from the ticket, are compiled into a list of available hosts. Remote processes are then instantiated via SSH as per-regular behavior – SSH could be altogether avoided as an optimization. Upon exit **sf_join** is called. The MPI library itself remains unchanged.

The MPICH port was more convoluted. MPICH expects a ring of mpd processes. These are persistent processes, one per host, that manage the launching of all MPI programs. The MPICH version of mpirun is a script which contacts the local mpd, delegates the creation of MPI processes, and receives the redirection of input and output. For each MPI job, the mpd daemons spawn a secondary one-way ring of mpdman processes that are specific to this job. Each mpdman manages a single MPI process. The MPICH library causes the individual application processes to interact with their private mpdman in order to redirect output and find peer processes. These requests are routed through the mpdman ring to the appropriate rank.

The MPI-SF port of MPICH modifies both mpirun and the library. The mpirun program is entirely rewritten to obviate the need for mpd and mpdman rings. It first invokes SnowFlock to create as many clones as necessary to satisfy the requested number of MPI workers. It then uses the information in the ticket to deterministically derive the TCP/IP coordinates of each worker. The mpirun program forks itself in each cloned VM, sets up the environment, and execs the application process. The modified MPI library communicates with the original mpirun process to redirect output and find other peers. At the end of execution **sf_join** is used.

3.9.2 *Web Server Scaling.* For our Apache implementation, we use the Linux IP Virtual Server (ipvs) as the load balancing frontend. The parent VM is a Linux/Apache/PHP stack. A daemon using the Python API runs next to the ipvs load-balancer and directs cloning of the parent VM to increase the request processing capacity. The number of web servers is increased in an exponential fashion. The trigger for cloning new servers is the load-balancer detecting a saturation of the current connection-handling capacity. If the load later drops, extraneous workers are starved of new connections, and once their current work is complete they are shut down. We did not use the decoupling extension for the experiments in this paper – each set of clones is derived from the latest state of the parent.

Once the Apache VM is cloned, the clone's IP address is reconfigured and as a result all connections cloned from the parent time out and fall off. This is harmless from a functional point of view, although it induces a small performance impact. In our proof-of-concept implementation, the data backend for the web server is an NFS share containing uniformly-sized randomly-filled files. A clone-reconfiguration callback was necessary in order to remount the NFS share with the new clone IP address. This reconfiguration callback is triggered by invoking a user process with a well-known name at the end of guest kernel resume code path (akin to the way in which kernel hotplug events propagate to user-space).

4. EVALUATION

In this Section we first turn our attention to a detailed micro evaluation of the different aspects that contribute to SnowFlock's performance and overhead. We then examine SnowFlock's performance through macro-benchmark experiments using representative application workloads, including embarrassingly parallel cluster applications, MPI codes, and a SPECWeb-driven Apache clustered server.

Our microbenchmark, parallel application and Apache experiments were carried out on a cluster of 32 Dell PowerEdge 1950 servers. Each machine had 4 GB of RAM and 4 Intel Xeon 3.2 GHz cores. We used the 32bit version of SnowFlock. The MPI-SF experiments were carried out on a cluster of 8 Dell R601 PowerEdge servers with 8 2.27 GHz Nehalem cores and 64 GB of RAM, using the 64 bit version of SnowFlock. In all cases we used Gigabit Broadcom Ethernet NICs, Gigabit Dell switches and locally-attached LSI-logic SATA storage.

Unless otherwise noted, all results reported in the following sections are the means of five or more runs, and error bars depict standard deviations. All VMs were configured with 1124 MB of RAM, which is the memory footprint needed by our most memory-intensive application (SHRiMP). All VMs had a nominal maximum of 32 VCPUs, but an actual allocation of a single VCPU throughout an experiment, unless otherwise noted. All VMs had a root disk of 8 GB in size, with no swap.

In several experiments we compare SnowFlock's performance against a "zero-cost fork" baseline. Zero-cost results are obtained with VMs previously allocated, with no cloning or state-fetching overhead, and in an idle state, ready to process the jobs allotted to them. As the name implies, zero-cost results are overly optimistic and not representative of cloud computing environments, in which aggressive consolidation of VMs is the norm and instantiation times are far from instantaneous. The zero-cost VMs are vanilla Xen domains configured identically to SnowFlock VMs in terms of kernel version, disk contents, RAM, and number of processors.

We focus first on the microbenchmarks. We then describe the parallel and MPI applications and workloads. We discuss the results of those macrobenchmarks before closing with our Apache results.

4.1 Micro Evaluation

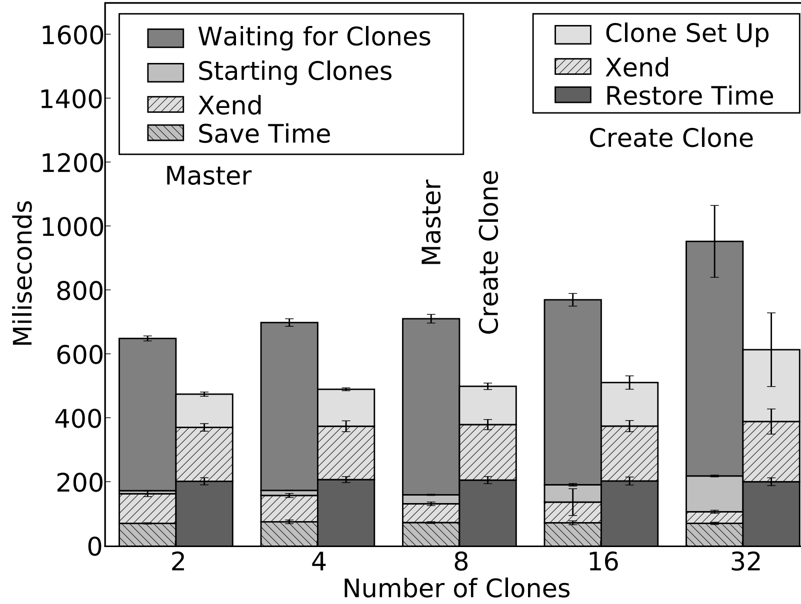
To better understand the behavior of SnowFlock, and to characterize the contributions of each of our design decisions to the overall performance of the system, we performed experiments to answer several performance questions:

—How fast does SnowFlock clone a VM, and how scalable is the cloning operation?

—What are the sources of overhead when SnowFlock fetches memory on demand to a VM?

—How sensitive is the performance of SnowFlock to the use of avoidance heuristics and the choice of networking strategy?

4.1.1 *Fast VM Replica Spawning.* Figure 6 shows the time spent replicating a single-processor VM to n VM clones, with each new VM spawned on a different physical host. For each size n we present two bars: the "Master" bar shows the global view of the operation, while the "Create Clone" bar shows the average time spent by all VM resume operations on the n physical hosts. The average size of a VM descriptor for these experiments was 1051 ± 7 KB.



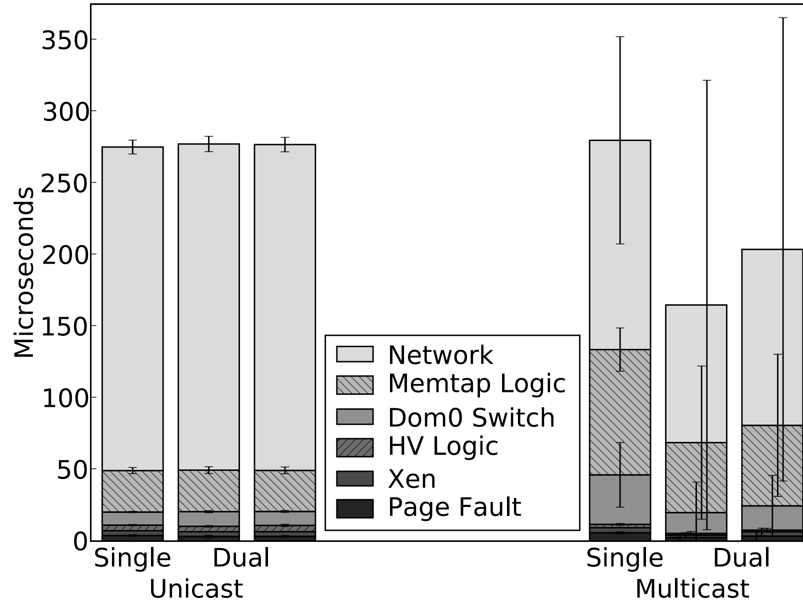
"Master" is the global view, while "Create Clone" is the average VM resume time for the n clones. Legend order matches bar stacking from top to bottom.

Fig. 6. Time To Create Clones

Recall that in Section 3, the process used to replicate a VM was introduced. For the purposes of evaluating the time required for VM cloning, we decompose this process into the following steps corresponding to bars in Figure 6: 1) Suspending the running VM and generating a VM descriptor. These are the "Save Time" and "Xend" components in the "Master" bar, with "Xend" standing for unmodified Xen code we leverage for VM metadata and device management; 2) contacting all target physical hosts to trigger VM instantiation ("Starting Clones"); 3) each target host pulls the VM descriptor via multicast ("Clone Set up" in the "Create Clone" bars); 4) spawning each clone VM from the descriptor ("Restore Time" and "Xend" in the "Create Clone" bars); and 5) waiting to receive notification from all target hosts ("Waiting for Clones", which roughly corresponds to the total size of the corresponding "Create Clone" bar).

From these results we observe the following: first, VM replication is an inexpensive operation ranging in general from 600 to 800 milliseconds; second, VM replication time is largely independent of the number of replicas being created. Larger numbers of replicas introduce, however, a wider variance in the total time to fulfill the operation. The variance is typically seen in the time to multicast each VM descriptor, and is due in part to a higher likelihood that on some host a scheduling or I/O hiccup might delay the VM resume for longer than the average.

4.1.2 *Memory On Demand.* To understand the overhead involved in our memory-on-demand subsystem, we devised a microbenchmark in which a VM allocates and



Components involved in a SnowFlock page fetch. Legend order matches bar stacking from top to bottom. "Single" bars for a single clone fetching state, "Dual" bars for two clones concurrently fetching state.

Fig. 7. Page Fault Time

fills in a number of memory pages, invokes SnowFlock to have itself replicated and then touches (by reading and writing the first byte) each page in the allocated set. We instrumented the microbenchmark, the Xen hypervisor and memtap to timestamp events during the fetching of each page. The results for multiple microbenchmark runs totaling ten thousand page fetches are displayed in Figure 7. The "Single" columns depict the result when a single VM fetches state. The "Dual" columns show the results when two VM clones concurrently fetch state.

We split a page fetch operation into six components. "Page Fault" indicates the hardware overhead of using the shadow page tables to detect first access to a page after VM resume. "Xen" is the cost of executing the Xen hypervisor shadow page table logic. "HV Logic" is the time consumed by our logic within the hypervisor (bitmap checking and SMP safety.) "Dom0 Switch" is the time spent while context switching to the domain 0 memtap process, while "Memtap Logic" is the time spent by the memtap internals. Finally, "Network" depicts the software and hardware overheads of remotely fetching the page contents over gigabit Ethernet.

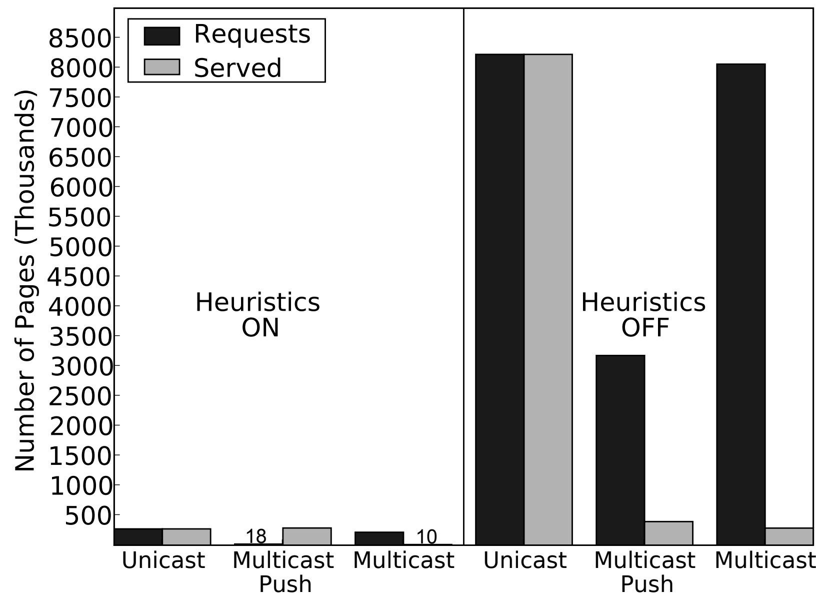
The overhead of page fetching is modest, averaging 275 μs with unicast (standard TCP). Our implementation is frugal, and the bulk of the time is spent in the networking stack. With multicast, substantially higher variances are observed in three components: "Network", "Memtap Logic', and "Domain 0 Switch". As explained in Section 3.5, memtap fully pauses the VM, and writes to the VM memory all batched received pages, making the average operation more costly. Also, mcdist's

logic and flow control are not as optimized as TCP's, and run in user space. The need to perform several system calls results in a high scheduling variability. While the "Network" bar may seem smaller than for TCP, an important amount of work that TCP performs in kernel context (e.g. resend, buffer stitching) is performed by mcdist in user-space, within the "Memtap Logic" bar. A final contributor to multicast's variability is the effectively bimodal behavior caused by implicit prefetching. Sometimes, the page the VM needs may already be present, in which case the logic defaults to a number of simple checks. This is far more evident in the dual case, in which requests by one VM result in prefetching for the other, and explains the high variance and lower overall "Network" averages for the multicast case.

4.1.3 *Sensitivity Analysis.* In this Section we perform a sensitivity analysis on the performance of SnowFlock, and measure the benefits of the heuristics and multicast distribution. Throughout these experiments we employed SHRiMP (see Section 4.3.2) as the driving application. The experiment spawns n uniprocessor VM clones, and on each clone it executes a set of reads of the same size against the same genome. N clones perform n times the amount of work as a single VM, and should complete in the same amount of time, yielding an n-fold throughput improvement. We tested twelve experimental combinations by: enabling or disabling the avoidance heuristics; increasing SHRiMP's memory footprint by doubling the number of reads from roughly 512 MB (167116 reads) to roughly 1 GB (334232 reads); and varying the choice of networking substrate between unicast, multicast, and multicast with push.

Figure 8 illustrates the memory-on-demand activity for a memory footprint of 1 GB and 32 VM clones. Consistent results were observed for all experiments with smaller numbers of clones, and with the 512 MB footprint. The immediate observation is the substantially beneficial effect of the avoidance heuristics. Nearly all of SHRiMP's memory footprint is allocated from scratch when the reads are loaded. The absence of heuristics forces the VMs to request pages they do not really need, inflating the number of requests from all VMs by an order of magnitude. Avoidance heuristics have a major impact in terms of network utilization, as well as in enhancing the scalability of the experiments and the system as a whole. The lower portion of Figure 9 shows that the decreased network utilization allows the experiments to scale gracefully, such that even unicast is able to provide good performance, up to 16 clones.

Three aspects of multicast execution are reflected in Figure 8. First, even under the extreme pressure of disabled heuristics, the number of pages served is reduced dramatically. This enables a far more graceful scaling than that achieved with unicast, as seen in the upper portion of Figure 9. Second, lockstep avoidance works effectively: lockstep executing VMs issue simultaneous requests that are satisfied by a single response from the server. Hence, a difference of an order of magnitude is evident between the "Requests" and "Served" bars in three of the four multicast experiments. Third, push mode increases the chances of successful prefetching and decreases the overall number of requests from all VMs, at the cost of sending more pages. The taller error bars in Figure 9 (up to 8 seconds) of the push experiments, however, reveal the instability caused by the aggressive distribution. Constantly receiving pages keeps memtap busy and not always capable of responding quickly



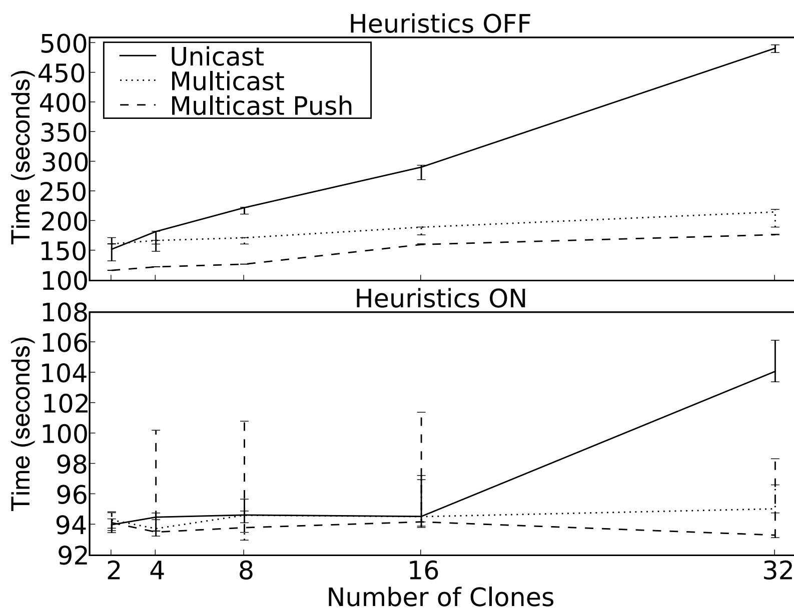
Comparison of aggregate page requests from 32 clones vs. number of pages sent by the memory server. Labels **18** and **10** stand for **18384** and **10436** pages served, respectively.

Fig. 8. Pages Requested vs. Served – SHRiMP

to a page request from the VM, affecting the runtime slightly and causing the VMs to lag behind. Further, multicast with push aggressively transmits the full set of pages comprising the VM memory state, mostly negating the opportunities for network utilization savings that on-demand propagation, coupled with our avoidance heuristics, creates.

The key takeaway from Figure 8 is the exceedingly small set of pages transmitted by the multicast memory server in the most efficient mode: multicast with push disabled and heuristics enabled. Less than eleven thousand pages of memory (i.e. less than 44 MB) are sent, including resends due to unreliable delivery. This is three orders of magnitude less than what state-of-the-art methods would entail (32 GB as seen for unicast without heuristics). Consider as well that this discussion is limited to memory. The frugality in the use of I/O resources is one of the key advantages of SnowFlock for cloud scalability, and is not limited to SHRiMP. For example, for QuantLib (see Section 4.3.4) we have seen less than 16 MB of total state (disk and memory) being served for the average experiment. None of the MPI-SF experiments in Section 4.5.4 demanded more than 30 GB of memory state propagation. The macrobenchmark that demanded the most state propagation was the web server in Section 4.5.5, which in average transmitted roughly 150 MB of memory state – still two orders of magnitude less than techniques in use in clouds today.

Figure 10 shows the requests of three randomly selected VMs as time progresses during the execution of SHRiMP. Avoidance heuristics are enabled, the footprint



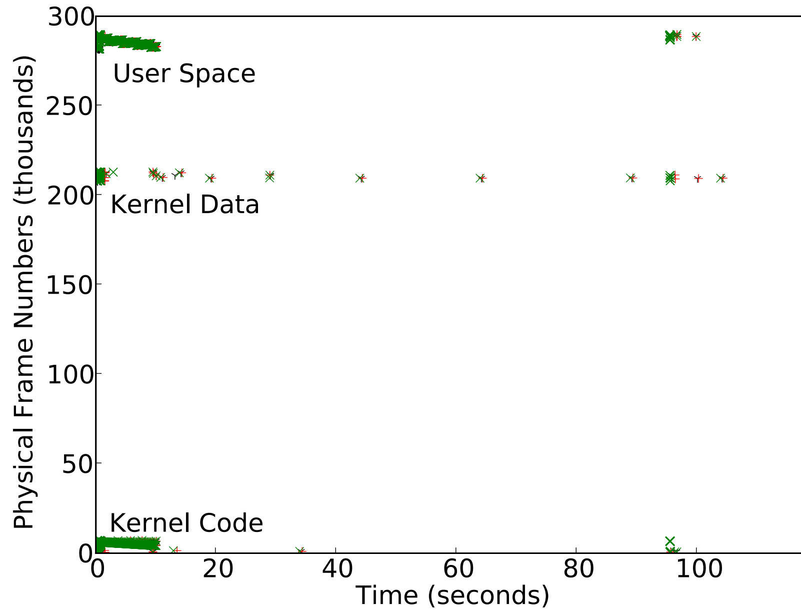
Uniprocessor VMs, n clones perform n times the work of 1 VM. Points are medians, error bars show min and max.

Fig. 9. Time to Completion – SHRiMP

is one GB, and there are 32 VM clones using multicast. Page requests cluster around three areas: kernel code, kernel data, and previously allocated user space code and data. Immediately upon resume, kernel code starts executing, causing the majority of the faults. Faults for pages mapped in user-space are triggered by other processes in the system waking up, and by SHRiMP forcing the eviction of pages to fulfill its memory needs. Some pages selected for this purpose cannot be simply tossed away, triggering fetches of their previous contents. Due to the avoidance heuristics, SHRiMP itself does not directly cause any fetches, and takes over the middle band of physical addresses for the duration of the experiment. Fetches are only performed rarely when kernel threads need data.

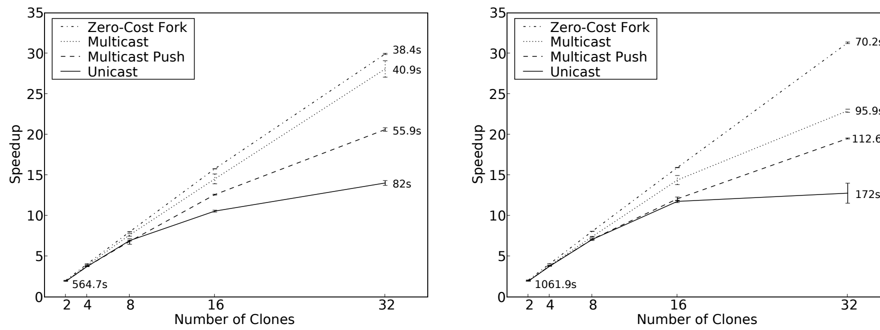
4.1.4 *Heuristic-adverse Experiment.* While the results with SHRiMP are highly encouraging, this application is not representative of workloads in which an important portion of memory state is needed after cloning. In this scenario, the heuristics are unable to ameliorate the load on the memory-on-demand subsystem. To synthesize such behavior we ran a similar sensitivity analysis using NCBI BLAST (see Section 4.3.1). We aligned 2360 queries split among an increasing number of n VM clones. The queries were run against a portion of the NCBI genome DB cached in main memory before cloning the VMs. With avoidance heuristics enabled, we varied the networking substrate and the size of the cached DB, from roughly 256 MB to roughly 512 MB.

Figure 11 (a) plots the speedups achieved for a DB of 256 MB by cloning a unipro-



Page requests over time for three randomly selected VMs out of 32 clones, using multicast with avoidance heuristics enabled.

Fig. 10. Pages Requested vs. Time – SHRiMP



Speedup against a single thread, with labels showing absolute times.

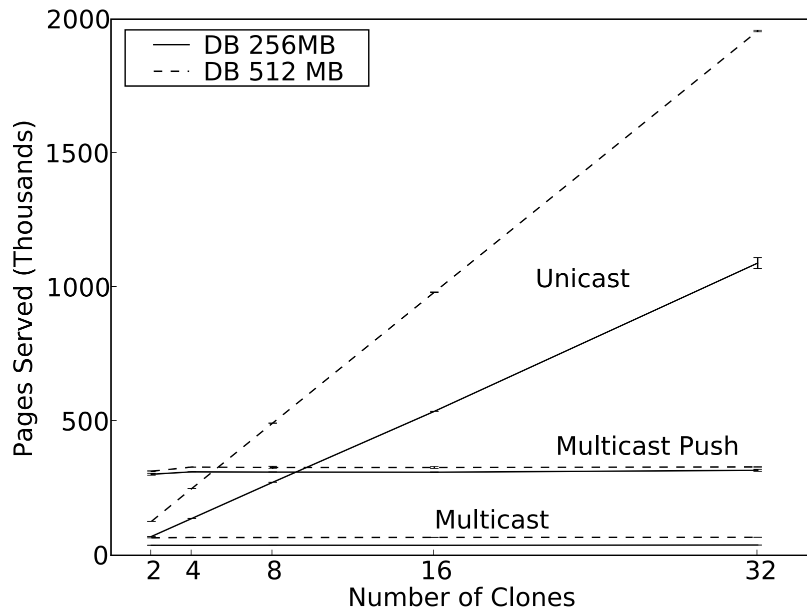
(a) Speedup – 256 MB DB

Speedup against a single thread, with labels showing absolute times.

(b) Speedup – 512 MB DB

Fig. 11. Sensitivity Analysis with NCBI BLAST

cessor VM to n replicas. Speedups are against a zero-cost ideal with a single VM, and the labels indicate the runtime for 32 VMs. We see an almost linear speedup for multicast, closely tracking the speedup exhibited with ideal execution, while unicast ceases to scale after 16 clones. Multicast push shows better performance



Aggregate number of pages sent by the memory server to all clones.

Fig. 12. Memory Pages Served – NCBI BLAST

than unicast but is unable to perform as well as multicast, due to memtap and network congestion. With a larger database (Figure 11 (b)) even multicast starts to suffer a noticeable overhead, hinting at the limits of the current implementation. We also note that because of the way parallelism is structured in this workload, multicast is actually counterproductive: each clone accesses a disjoint subset of the database, thus multicasting database pages to all clones is a waste of network resources. We address possible solutions to this in Section 6.1.

Figure 12 shows the number of pages fetched vs. the number of clones for the three networking substrates. The information is consistent with that presented in Figure 8. The linear scaling of network utilization by unicast leads directly to poor performance. Multicast is not only the better performing, but also the one imposing the least load on the network, allowing for better co-existence with other sets of SnowFlock cloned VMs. These observations, coupled with the instabilities of push mode shown in the previous Section, led us to choose multicast with no push as SnowFlock's default behavior for the macro-evaluation in Section 4.2.

4.2 Macro Evaluation

We now turn our attention to a macrobenchmark based evaluation. We study "embarrassingly parallelizable" applications, MPI applications, and a web server.

4.3 Embarrassingly Parallel Applications

To evaluate the generality and performance of SnowFlock, we tested several usage scenarios involving six typical applications from bioinformatics, graphics rendering, financial services, and parallel compilation. We devised workloads for these applications with runtimes of above an hour on a single-processor machine, but which can be substantially reduced to the order of a hundred seconds if provided enough resources. One of the applications is ClustalW, whose code we modified directly to adapt it to SnowFlock, as shown in Section 2.1.1. For the other five, we use scripting to drive VM cloning, as explained in Section 2.1.2.

4.3.1 *NCBI BLAST.* The NCBI implementation of BLAST[Altschul et al. 1997], the Basic Local Alignment and Search Tool, is perhaps the most popular computational tool used by biologists. BLAST searches a database of *biological sequences*, strings of characters representing DNA or proteins, to find sequences similar to a query. Similarity between two sequences is measured by an *alignment* metric, that is typically similar to edit distance. BLAST is demanding of both computational and I/O resources; gigabytes of sequence data must be read and compared with each query sequence, and parallelization can be achieved by dividing either the queries, the sequence database, or both. We experimented with a BLAST search using 1200 short protein fragments from the sea squirt *Ciona savignyi* to query a 1.5 GB portion of NCBI's non-redundant protein database, a consolidated reference of protein sequences from many organisms. VM clones access the database, which is a set of plain text files, via an NFS share, which is a typical setup in bioinformatics labs. Database access is parallelized across VMs, each reading a different segment, while query processing is parallelized across process-level clones within each VM.

4.3.2 *SHRiMP.* SHRiMP [Rumble et al. 2009] (SHort Read Mapping Package) is a tool for aligning large collections of very short DNA sequences ("reads") against a known genome: e.g. the human genome. This time-consuming task can be easily parallelized by dividing the collection of reads among many processors. While similar overall to BLAST, SHRiMP is designed for dealing with very short queries and very long sequences, and is more memory intensive, requiring from a hundred bytes to a kilobyte of memory for each query. In our experiments we attempted to align 1.9 million 25 letter-long reads, extracted from a *Ciona savignyi* individual using the AB SOLiD sequencing technology, to a 5.2 million letter segment of the known *Ciona savignyi* genome.

4.3.3 *ClustalW.* ClustalW [Higgins et al. 1994] was first introduced in Section 2.1.1, where we described the direct code modifications employed to parallelize the program using SnowFlock. Like BLAST, ClustalW is offered as a web service by organizations owning large computational resources [EBI]. ClustalW builds a guide tree of pairwise similarities between biological sequences, such as proteins or DNA. To build this guide tree, ClustalW uses progressive alignment, a greedy heuristic that significantly speeds up the multiple alignment computation, but requires pre-computation of pairwise comparisons between all pairs of sequences. The pairwise comparison is computationally intensive and embarrassingly parallel, since each pair of sequences can be aligned independently. We conducted experiments with the SnowFlock-aware version of ClustalW performing guide-tree generation

for 200 synthetic protein sequences of 1000 amino acids (characters) each.

4.3.4 *QuantLib.* QuantLib [QuantLib.org] is an open source development toolkit widely used in quantitative finance. It provides a vast set of models for stock trading, equity option pricing, risk analysis, etc. Quantitative finance programs are typically single-instruction-multiple-data (SIMD): a typical task using QuantLib runs a model over a large array of parameters, e.g. stock prices, and is thus easily parallelizable by splitting the input. In our experiments we ran a set of Monte Carlo, binomial and Black-Scholes variant models to assess the risk of a set of equity options. Given a fixed maturity date, we processed 1024 equity options varying the initial and striking prices, and the volatility. The result is the set of probabilities yielded by each model to obtain the desired striking price for each option.

4.3.5 *Aqsis – Renderman.* Aqsis [Aqsis.org] is an open source implementation of Pixar's RenderMan interface [Pixar], an industry standard widely used in films and television visual effects since 1989. This renderer accepts scene descriptions produced by a modeler and specified in the RenderMan Interface Bitstream (RIB) language. Rendering also belongs to the SIMD class of applications, and is thus easy to parallelize: multiple instances can each perform the same task on different frames of an animation. For our experiments we fed Aqsis a sample RIB script from the book "Advanced RenderMan" [Apodaka and Gritz 2000].

4.3.6 *Distcc.* Distcc [Samba.org] is software which distributes builds of C/C++ programs over the network for parallel compilation. It operates by sending preprocessed code directly to a compiling process on each host and retrieves object file results back to the invoking host. Because the preprocessed code includes all relevant headers, it is not necessary for each compiling host to access header files or libraries; all they need is the same version of the compiler. Distcc is different from our previous benchmark in that it is not strictly embarrassingly parallel: actions are coordinated by a master host farming out preprocessed files for compilation by workers. In our experiments we compile the Linux kernel (version 2.6.16.29) from `kernel.org`, using gcc version 4.1.2 and the default kernel optimization level (`O2`.)

4.4 MPI Applications

This section describes the applications that we used for evaluating our SnowFlock implementation of MPI. We tested the Open MPI-based version of MPI-SF with four applications representative of the domains of bioinformatics, rendering, physics, and chemistry. We did not have to modify a single line of code in any of these applications. The same binaries we used for zero-cost experiments were used in SnowFlock experiments with no inconveniences.

4.4.1 *ClustalW-MPI.* This is the MPI version of the ClustalW program. The ClustalW-MPI [Li 2003] code is substantially different from the basic ClustalW code described previously, and it parallelizes both phases of the computation using MPI. ClustalW-MPI aggregates sequence pairs into small batches that are sent to nodes as they become available, thus preventing workers from idling. In our experiment we compute the multiple alignment of 600 synthetically generated peptide sequences with a length of one thousand amino acids each.

4.4.2 *MrBayes.* MrBayes [Huelsenbeck and Ronquist 2001] builds phylogenetic trees showing evolutionary relationships between species across generations, using Bayesian inference and Markov chain Monte-Carlo (MCMC) processes. MrBayes uses MPI to distribute computation across multiple nodes, although with a high degree of synchronization, and to swap state between nodes. The number of chains, probabilities, number of swaps, and frequency of swaps are all parameters that govern the degree of parallelism and synchronization. Our MrBayes experiment builds the evolutionary tree of 30 different bacterium species based on gapped DNA data, tracking back nine hundred generations and using up to 256 chains for the MCMC processes.

4.4.3 *VASP.* VASP [VASP - GROUP] is a package for performing ab-initio quantum-mechanical simulations, using different pseudopotentials and plane wave basis sets. VASP is particularly suited to inorganic chemistry analyses. MPI parallelization of VASP involves partitioning a three-dimensional spatial grid into bands. This maximizes communication between nodes performing simulation of intra-band interactions, although synchronization is still required for inter-band dynamics. Further parallelization can be achieved by splitting the computation across plane wave coefficients. Our VASP test performs an electronic optimization process on a hydrogen molecule.

4.4.4 *Tachyon.* Tachyon [Stone, J.] is a standard ray-tracing based renderer. Ray tracing is a technique that follows all rays of light in a scene as they reflect upon the different surfaces, and perform highly detailed renderings. Ray tracers like Tachyon are highly amenable to parallelization, by partitioning the rendering space into a grid and computing each grid partition in a mostly independent fashion. We used Tachyon to render a twenty-six thousand atom macro-molecule using ambient occlusion lighting and eight anti-aliasing samples.

4.5 Macrobenchmarks

In our macro evaluation we aim to answer the following five questions:

- How does SnowFlock compare to other methods for instantiating VMs?
- What is the performance of SnowFlock for the embarrassingly parallel applications described in Section 4.3?
- How does SnowFlock perform in cloud environments with multiple applications simultaneously and repeatedly forking VMs, even under adverse VM allocation patterns?
- What is the performance of MPI-SF for the applications described in Section 4.4?
- Finally, what are the SPECWeb performance metrics for a simple SnowFlock-based clustered Apache web server?

4.5.1 *Comparison.* Table II illustrates the substantial gains SnowFlock provides in terms of efficient VM cloning and application performance. The table shows results for SHRiMP using 128 processors (32 4-VCPU VMs) under three configurations: SnowFlock with all the mechanisms described in Section 3, and two versions of Xen's standard suspend/resume that use NFS and multicast to distribute the

suspended VM image. The results show that SnowFlock significantly improves execution time with respect to traditional VM management techniques, with gains ranging between a factor of two to a factor of seven. Further, SnowFlock is two orders of magnitude better than traditional VM management techniques in terms of the amount of VM state transmitted. Despite the large memory footprint of the application (one GB), SnowFlock is capable of transmitting, in parallel, little VM state. Experiments with our other application benchmarks present similar results and are therefore not shown.

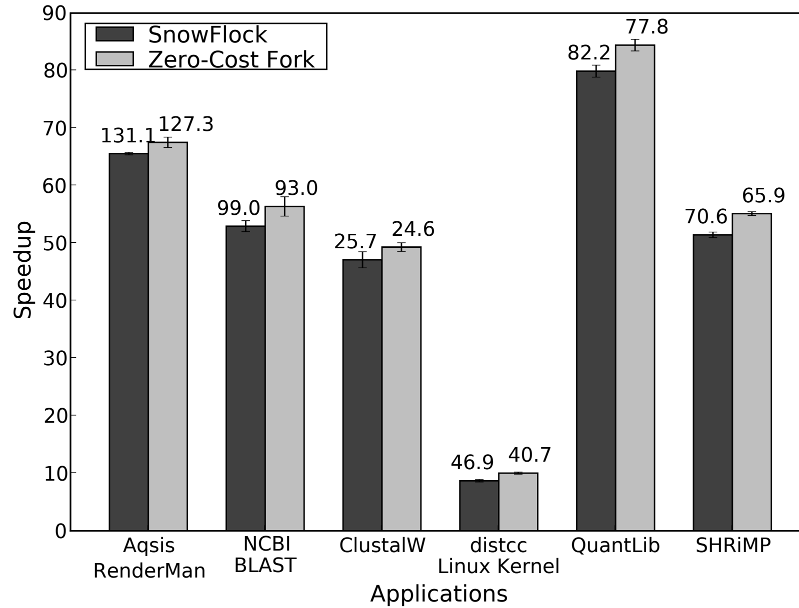
	Benchmark Time (s)	State Sent (MB)
SnowFlock	70.63 ± 0.68	41.79 ± 0.7 (multicast)
Suspend/Resume multicast	157.29 ± 0.97	1124 (multicast)
Suspend/Resume NFS	412.29 ± 11.51	$1124 * 32$ (unicast)

Table II. SnowFlock vs. VM Suspend/Resume. Benchmark time and VM state sent, for SHRiMP with 128 threads.

4.5.2 *Application Performance.* We tested the performance of SnowFlock with the applications described in Section 4.3 and the following features enabled: memory on demand, multicast without push, avoidance heuristics, and hierarchical cloning. For each application we spawned 128 threads of execution (32 4-VCPU SMP VMs on 32 physical hosts) in order to fully utilize our testbed. SnowFlock was tested against a zero-cost ideal with 128 threads to measure overhead, and against a single thread in order to measure speedup.

Figure 13 summarizes the results of our application benchmarks. We obtain speedups very close to the zero-cost ideal, and total time to completion no more than five seconds (or 7%) greater than the zero-cost ideal. The overheads of VM replication and on-demand state fetching are small. These results demonstrate that the execution time of a parallelizable task can be reduced, given enough resources, to the order of a minute (which is typically considered an "interactive" range in high performance computing.) We note three aspects of application execution. First, ClustalW yields the best results, presenting an overhead with respect to zero-cost as low as one second. Second, even though most of the applications are embarrassingly parallel, the achievable speedups are below the maximum. Some children may finish ahead of others, such that "time to completion" effectively measures the time required for the slowest VM to complete. Third, distcc's tight job synchronization results in underutilized children and low speedups, although distcc still delivers a time to completion of under a minute.

4.5.3 *Scale and Agility.* We address SnowFlock's capability to support multiple concurrent forking VMs. We launched four VMs such that each simultaneously forked 32 uniprocessor VMs. To stress the system, after completing a parallel task, each parent VM joined and terminated its children and immediately launched another parallel task, repeating this cycle five times. Each parent VM executed a different application. We selected the four applications that exhibited the highest degree of parallelism (and child occupancy): SHRiMP, BLAST, QuantLib, and Aqsis. To further stress the system, we abridged the length of the cyclic parallel



Applications ran with 128 threads: 32 VMs × 4 VCPUs. **Bars** show **speedup**, measured against a 1 VM × 1 VCPU ideal. **Labels** indicate **time to completion** in seconds.

Fig. 13. Application Benchmarks

task so that each cycle would finish in between 20 and 35 seconds. We employed an "adversarial allocation" in which each task uses 32 processors, one per physical host, so that 128 SnowFlock VMs are active at most times, and each physical host needs to fetch state from four parent VMs. Note that the allocation is not adversarial enough to overflow the available resources and overcommit processors: resource management and overcommit is not within the scope of this paper. The zero-cost results were obtained with an identical distribution of VMs. Since there is no state-fetching performed in the zero-cost case, the actual allocation of VMs does not affect those results.

The results, shown in Figure 14, demonstrate that SnowFlock is capable of withstanding the increased demands of multiple concurrent forking VMs. As we have shown in Section 4.1.3, this is mainly due to the small overall number of memory pages sent by the multicast server, when pushing is disabled and heuristics are enabled. The introduction of multiple concurrently forking VMs causes no significant increase in overhead, although outliers with higher time to completion are seen, resulting in wider error bars. These outliers are caused by occasional congestion when receiving simultaneous bursts of VM state for more than one VM; we believe optimizing mcdist will yield more consistent running times.

4.5.4 *MPI Applications.* We compared our four MPI applications against zero-cost fork alternatives in which a cluster of identical VMs was already set up. We used the Open MPI variant of MPI-SF. We executed these experiments with 32

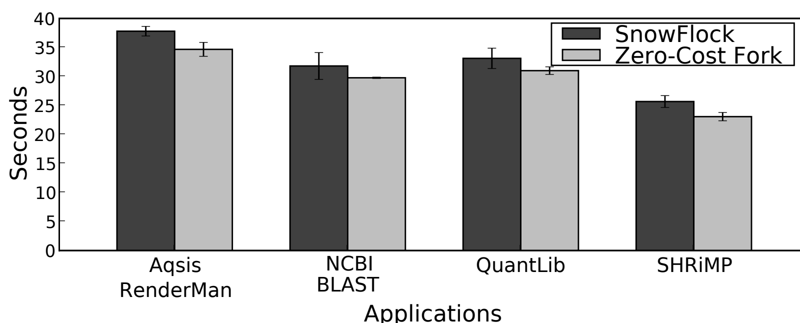


Fig. 14. Concurrent execution of multiple forking VMs. For each task we repeatedly cycled cloning, processing and joining.

MPI processes, taking advantage of four processors in each of eight machines. It is worth pointing out again that the version of ClustalW used here is completely different from the one used in the previous section.

The results observed in Figure 15 are fairly similar to those seen for other parallel applications in the previous section. There is a rather small overhead with SnowFlock, in spite of starting with a single VM, cloning, establishing the MPI SSH connections, and propagating state during runtime.

The overhead is, however, not as small as we would like, typically averaging over ten seconds. The network-intensive nature of these MPI applications aggressively exercises the socket buffer slab cache allocator in the guest kernel. This is a buddy allocator of sub-page data structures, in this case the socket buffers containing all the headers for a network packet. Obtaining a new socket buffer results in a reference to a slab cache page, whose fetch may not be prevented by our heuristics due to containing other "in-use" socket buffers. Thus, aggressive network traffic results in aggressive fetching of socket buffer pages. We believe we can substantially trim down the fetching of these pages by managing the slab allocator in a way that is more SnowFlock-aware.

A secondary culprit for the increased overhead is the need to populate ARP caches with the MAC addresses of these suddenly appearing clones – and the immediate need for these ARP resolutions. Because IP reconfiguration in the clones is neither instantaneous nor synchronous, a period of inconsistency in the ARP caches of all VMs has to be weathered before all connections are ready to proceed. Optimizing the IP reconfiguration mechanism should prevent these hiccups.

4.5.5 *SnowFlock-based Adaptive Web Server.* As described in Section 3.9.2, we implemented an adaptive SnowFlock-based web server by augmenting an ipvs load balancer with the SnowFlock transitive API. The load-balancer can thus trigger the creation of new VMs that are clones of a Linux/Apache/PHP stack. The roster of cloned VMs serve static web pages from an NFS share which they remount with their new IP after cloning. To evaluate this setup and the ability of SnowFlock to implement a simple self-scaling server, we used the SPECweb2005 benchmark [SPEC 2005].

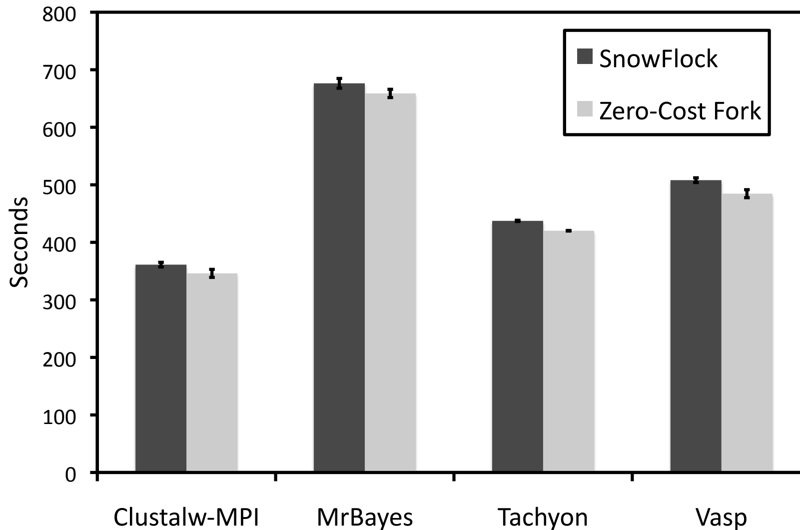


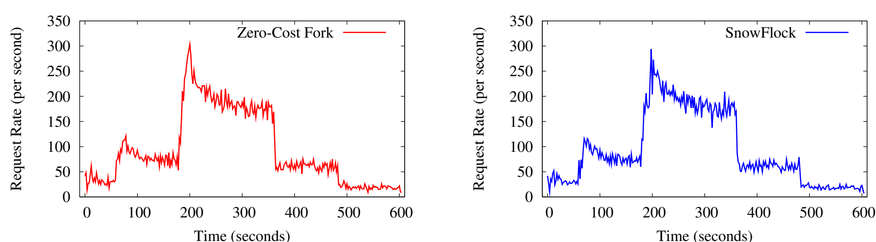
Fig. 15. Performance of MPI Applications

SPECweb2005 is an industry-standard benchmark used to measure Web server performance. We drive our experiment with SPECweb's Support workload, which models a support website where users browse product listings, search for products and download files. The workload uses Markov chains in a closed-loop to model user visits, with a typical distribution of file sizes and overlaps in file use. To model a bursty workload, we increase and decrease the number of simulated users as shown in Figure 16 (a).

We use the SPECweb test harness to measure the prototype's instantaneous QoS as perceived by the client. SPECweb considers a fetch to be "Good" if the client receives a small requested file in no more than three seconds, and large files with an average throughput of no less than 99 Kbps. For each elapsed second of our experiments, we measure the QoS degradation as the percentage of fetches issued in that second that fail to meet SPECweb's standard of Good.

The SnowFlock-based prototype was tested under the same bursty SPECweb Support workload as zero-cost fork ideal with 8 VMs pre-allocated and assigned to an unmodified ipvs load-balancing frontend. As expected, the prototype was responsive and produced new workers promptly (Figures 16 (b) and (c)). Each time the current load exceeded its capacity, the adaptive server doubled it's footprint in a single generation of new clone workers, up to a maximum of eight. The footprint was then pared back as possible to match the eventual need. At 8 uniprocessor VMs the gigabit pipe was filled with outbound traffic in our testbed; we used uniprocessor VMs to exact the maximum demand on the VM replication system.

Because the working set had not yet been transferred, the performance of new workers was briefly degraded while they faulted on the missing pages (Figure 16 (d)). When the workload spiked from 300 to 800 users, although the adaptive server failed to meet the SPECweb standard for 14 of the next 19 seconds, within 20 seconds the newly cloned workers' QoS matched that of their ideal zero-cost counterparts.

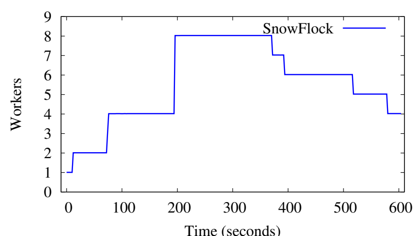


The SPECweb workload steps from 100 to 300 and then 800 simulated users, with each increase occurring over 20 seconds.

(a) Bursty Workload

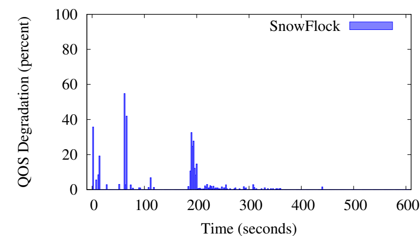
The SnowFlock adaptive server provides 98.8% of the throughput of the static server. Over 28,000 requests are served.

(b) Keeping Up



The adaptive server detects the spiking load and reactively provisions new workers.

(c) Adaptive Workers



SnowFlock cloned VM workers are sufficiently agile to service the load, but with a brief partial degradation of QoS.

(d) QoS is Briefly Degraded

Fig. 16. SnowFlock-based Adaptive Apache Web Server

Overall, the SnowFlock-based adaptive server throughput of Good QoS fetches (goodput) fell only 2.8% short of the static cluster's, while the overall throughput fell only 1.2%.

Much like the heuristics-adverse experiment of Section 4.1.4, the web server experiments highlight some of the current limitations of SnowFlock. This rather simple macrobenchmark propagates roughly 150 MB of memory state at the expense of a small degradation in quality. The state propagation overhead needs to be improved for more demanding scaling scenarios or more complex server stacks, lest the degradations in throughput and QoS become prohibitive. Our future work plans for this challenge rely on bridging the semantic gap and gleaning higher quality information in terms of what each region of memory is used for by the VM. By tailoring prefetching policies to the use of each region of memory, we expect to be able to hide most of the state propagation overhead of SnowFlock, yet retain a use of I/O and network cloud resources as tightly adjusted as possible.

5. RELATED WORK

Several projects in the areas of virtualization and high performance computing are related to the ideas presented in this paper. The fundamental primitive of SnowFlock, VM forking or cloning, shares the same conceptual backbone as process forking, an OS technique dating back to the initial implementations of UNIX [Ritchie and Thompson 1974]. Except during bootstrap, the only way to start processes in UNIX is by cloning the running structures of an already existing process, and adjusting relevant values. A separate `exec` call is necessary to repopulate the memory image of the process with a different executable.

When implementing a cloning facility, lazy-copying of contents, as introduced by Accent [Zayas 1987], is the norm. In other words, pieces of the cloned entity are fetched on demand. One incarnation of this technique is CoW cloning, in which the cloned entities share all possible resources initially, and only instantiate a private copy of an element before modifying it. Modern UNIX process forking implementations leverage the CoW technique to minimize the amount of memory copying for a clone operation.

CoW techniques are particularly useful in file system environments for performing snapshots, as exemplified by ZFS [Sun Microsystems] and LVM [Red Hat]. Wide-area VM migration projects such as the Collective [Sapuntzakis et al. 2002], Internet Suspend/Resume [Kozuch and Satyanarayanan 2002] and SnowBird [Lagar-Cavilla et al. 2007] have also used lazy copy-on reference for VM disk state. The low frequency and coarse granularity of access to secondary storage allows copying large batches of state over low-bandwidth high-latency links.

Cloning and CoW are frequently used techniques in research projects. In the area of virtualization we highlight the Cloneable JVM and the Potemkin Honeyfarm. The Cloneable JVM [Kawachiya et al. 2007] allows a running Java servlet to effect a full clone of the underlying JVM, with a dual objective: first, to overlay additional MMU-based protection for the running server, on top of the isolation already provided by the JVM; and second, to minimize the startup time for a new JVM by sharing memory pages and "JiT-ed" (Just-in-Time compiled) code from the original JVM. This borrows on the implicit notion shared by many parallel programming paradigms based on cloning: that the new cloned copy is stateful and keeps a memory of all processing performed up to that stage, thus removing the overhead of a cold start.

In Potemkin [Vrable et al. 2005], lightweight Xen VMs are created when traffic on an unattended IP address is detected. This allows for the flexible instantiation and destruction of honeypot nodes on demand. Honeypot creation is achieved by cloning a base VM image and sharing pages through CoW mechanisms. There are several important differences with the fully-featured VM clones that SnowFlock can instantiate across multiple hosts. First, by virtue of using CoW of memory pages, the clones are restricted to exist on the same physical host as the originating VM. Second, the honeypot VMs are short-lived, have a very small memory footprint (128 MB) and do not use secondary storage (the file system is mounted in RAM), thereby greatly simplifying the cloning task.

Live migration [Clark et al. 2005; Nelson et al. 2005], is a classic way of manipulating VM state through iterative rounds of pre-copying. Remus [Cully et al. 2008]

provides instantaneous failover for mission-critical tasks by keeping an up-to-date replica of a VM in a separate host. Using a modified version of live migration, dirty VM state is continually pushed to the failover site, while a synchronization protocol (similar to Rethink the Sync's [Nightingale et al. 2006]) ensures that no output becomes world-visible until synchronization of an epoch (typically the last 25 to 50 milliseconds) succeeds. Finally, post-copy VM migration [Hines and Gopalan 2009] is a scheme in which the site of execution of a VM is immediately changed while pages of the VM's memory are propagated post-facto.

Another relevant project is Denali [Whitaker et al. 2002], which dynamically multiplexes VMs that execute user-provided code in a web-server, with a focus on security and isolation. Operation replay can be used to recreate VM state by starting from the same initial image and replaying all non-deterministic inputs. This technique is used for forensics studies of intrusions [Dunlap et al. 2002], and can be used to efficiently migrate VM state optimistically [Surie et al. 2008].

The seminal work by Waldspurger [Waldspurger 2002] with VMware's ESX server laid the groundwork for VM consolidation. Through sophisticated memory management techniques like *ballooning* and *content-based page sharing*, ESX can effectively oversubscribe the actual RAM of a physical host to multiple VMs. Content-based page sharing is a generalization of the notion of content addressable storage, by which similar objects are identified and indexed via a cryptographic hash of their contents. Content-based indexing of VM state has been employed for security [Litty et al. 2008] and migration purposes [Surie et al. 2008]. Further work in the area was presented by the Difference Engine group [Gupta et al. 2008], which aims to provide additional memory savings by scanning RAM and synthesizing little used pages as binary diffs of other similar pages.

Content addressable storage is a promising technique for minimizing the amount of state to be transferred. Content-addressing has been heavily used for content sharing [Tolia et al. 2006], and distributed file systems [Tolia et al. 2003]. Content addressability allows cross-indexing of similar content from multiple caches, and thus enables the possibility of minimizing the amount of transferred data to that which is truly "unique" (i.e. no other cache contains it). We believe that the large amount of software sharing between VM images will enable the use of content addressability as a viable means to minimize the amount of disk state replication. For example, in a Xen paravirtual VM, it is extremely likely that the paravirtualized kernel in use will be the unique version distributed by the Xen developers.

VM-based server consolidation has become a *de facto* standard for commodity computing, as demonstrated by the Amazon Elastic Cloud [Amazon.com], and other infrastructures like GoGrid [GoGrid] or Mosso [Mosso]. Manipulation of VM state lies at the the heart of most industry standard techniques for managing commodity computing clusters [Steinder et al. 2007], sometimes referred to as clouds. These techniques include suspending to and resuming from disk, live migration [Clark et al. 2005; Nelson et al. 2005; Wood et al. 2007], and possibly ballooning and memory consolidation. The current deployment models based on these techniques suffer from a relative lack of speed. With VM state ranging in tens of GB (disk plus memory), it is no surprise that Amazon's EC2 instantiates VMs in minutes, as shown in Figure 1.

The ability to dynamically move, suspend, and resume VMs allows for aggressive multiplexing of VMs on server farms, and as a consequence several groups have put forward variations on the notion of a "virtual cluster" [Emeneker and Stanzione 2007; Ruth et al. 2005]. The focus of these lines of work has been almost exclusively on resource provisioning and management aspects; one salient commercial instance of this is VMware's DRS [VMware]. Furthermore, implementation is typically performed in a very un-optimized manner, by creating VM images from scratch and booting them [Murphy et al. 2009], and typically lacks the ability to allow dynamic runtime cluster resizing [Foster et al. 2006; Zhang et al. 2005]. [Chase et al. 2003] present dynamic virtual clusters in which nodes are wiped clean and software is installed from scratch to add them to a long-lived, on-going computation. Nishimura's virtual clusters [Nishimura et al. 2007] are statically sized, and created from a single image also built from scratch via software installation. [Fox et al. 1997] keep a pool of pre-configured machines on standby, and bring these generic hot spares to bear as required. [Urgaonkar et al. 2008] employ predictive and reactive methods to respond to increased load by provisioning additional VMs, taking up to five minutes to double the application capacity.

Further research in aspects of VM management has been focused on scheduling VMs based on their different execution profiles [Lin and Dinda 2005] (e.g. batch vs. interactive,) grid-like queuing of VMs for deployment as cluster jobs [Zhang et al. 2005], and assigning resources based on current use [Alvarez et al. 2001]. One particular resource of interest is energy management in server farms [Nathuji and Schwan 2007]. Our work on an adaptively self-scaling web server finds points of similarity with several projects that attempt to optimize the QoS of a server within a static allocation of resources, be they physical machines or VMs. Some of the methods used include VM migration or scheduling [Ranjan et al. 2003], workload management and admission control to preserve the QoS through optimal resource use [Zhang et al. 2009; Urgaonkar and Shenoy 2008; Elnikety et al. 2004; Bartolini et al. 2009], and allowing applications to barter resources [Norris et al. 2004].

The Emulab [White et al. 2002] testbed predates many of the ideas behind virtual clusters. After almost a decade of evolution, Emulab uses virtualization to efficiently support large-scale network emulation experiments [Hibler et al. 2008]. An "experiment" is a set of VMs connected by a virtual network, much like a virtual cluster. Experiments are long-lived and statically sized: the number of nodes does not change during the experiment. Instantiation of all nodes takes tens to hundreds of seconds.

Emulab uses Frisbee [Hibler et al. 2003] as a multicast distribution tool to apply disk images to nodes during experiment setup. To our knowledge, Frisbee is the first storage substrate to aggressively employ content multicasting through receiver initiated transfers that control the data flow. Frisbee and mcdist differ in their domain-specific aspects: e.g. Frisbee uses filesystem-specific compression, not applicable to memory state; conversely, lockstep detection in mcdist does not apply to disk distribution. We view the use of high-speed interconnects such as RDMA on Infiniband [Huang et al. 2007], if available, as a viable alternative to multicasting. While efficient point-to-point VM migration times with RDMA on Infiniband have been demonstrated, we note that the latency of this method increases linearly

when a single server pushes a 256 MB VM to multiple receivers, which is a crucial operation to optimize for use in SnowFlock.

Regarding MPI, the suitability of virtualization for MPI-based applications has been explored by Youseff et al. [Youseff et al. 2006], who demonstrated that MPI applications can run with minimal overhead in Xen environments. There are several other parallel programming platforms for which we expect to apply the same SnowFlock-adaptation principles demonstrated for MPI. We mention here the Parallel Virtual Machine (PVM) [Geist et al. 1994], which provides a message passing model very similar to that of MPI; OpenMP [Chandra et al. 2000], which targets parallel computing in a shared memory multiprocessor; and MapReduce [Dean and Ghemawat 2004; Apache A]. MapReduce is particularly relevant today as it targets the emergent class of large parallel data-intensive computations. These embarrassingly parallel problems arise in a number of areas spanning bioinformatics, quantitative finance, search, machine learning, etc. We believe that the API encapsulation approach presented here can be equally applied to interfaces such as MapReduce or PVM.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the primitive of VM fork and SnowFlock, our Xen-based implementation. Matching the well-understood semantics of stateful worker creation, SnowFlock provides cloud users and programmers the capacity to instantiate dozens of VMs in different hosts in sub-second time, with little runtime overhead, and frugal use of cloud I/O resources. SnowFlock thus enables the simple implementation and deployment of services based on familiar programming patterns that rely on fork's ability to quickly instantiate stateful workers.

SnowFlock makes use of three key observations. First, that it is possible to drastically reduce the time it takes to clone a VM by copying only the critical state, and fetching the VM's memory image efficiently on-demand. Second, that simple modifications to the guest kernel significantly reduce network traffic by eliminating the transfer of pages that will be overwritten. For our applications, these optimizations can drastically reduce the communication cost for forking a VM to a mere 40 MB for application footprints up to one GB. Third, the similarity of the memory access patterns among cloned VMs makes it beneficial to distribute VM state using multicast. This allows for the instantiation of a large number of VMs at a (low) cost similar to that of forking a single copy.

In closing, SnowFlock lowers the barrier of entry to cloud computing and opens the door for new cloud applications. VM fork provides a well-understood programming interface with substantial performance improvements; it removes the obstacle of long VM instantiation latencies, and greatly simplifies management of application state. SnowFlock thus places in the hands of users the full potential of the cloud model by simplifying the programming of applications that dynamically change their execution footprint. In particular, SnowFlock is of immediate relevance to users wishing to test and deploy parallel applications in the cloud.

6.1 Future Directions for SnowFlock

SnowFlock is an active open-source project [UofT]. Our future work plans involve adapting SnowFlock to big-data applications. We believe there is fertile

research ground studying the interactions of VM fork with data parallel APIs such as MapReduce [Dean and Ghemawat 2004]. For example, SnowFlock's transient clones cannot be entrusted with replicating and caching data due to their ephemeral nature. Allowing data replication to be handled by hosts, as opposed to the VMs, endows the VMs with big-data file system agnosticism. In other words, it allows the VMs to remain oblivious to how the big-data replication and distribution policies are actually implemented by the underlying storage stack.

The evaluation results presented here show that many of the performance shortcomings of SnowFlock can be overcome with a tighter integration and collaboration with the guest VM. This is typically referred to as bridging the semantic gap. For example, an awareness of the sub-page slab allocator used by Linux for network packet headers would allow SnowFlock to optimize state propagation for clones performing intensive networking, as in our MPI workloads. An awareness of the different uses of the memory regions of a VM, such as code, data, file system page cache, etc, may allow us to efficiently tailor prefetching policies to each region and eliminate most of the state propagation overhead seen for web server workloads. In particular, it is well understood that physical memory is highly fragmented for different uses: a push policy operating on subsets of semantically-related memory (such as the user-space memory image of process X), as opposed to blindly following physical memory locality, is bound to perform much better. Finally, observing the access patterns of the parent VM post-cloning may yield valuable information on the immediate access patterns of all clones (e.g. the common code and data they will all access to recover from cloning).

Another way to bridge the semantic gap would entail explicit application support. As shown in Section 4.1.4, multicast may not always be a good idea. In some situations clone VMs access disjoint regions of the parent VM's memory, negating the prefetching benefits of multicast. Providing applications with the ability to tag certain memory regions as "disjoint access" would allow SnowFlock to prevent useless multicast and tailor memory propagation in a finer-grained manner.

SnowFlock is promising as a potential transforming force in the way clusters or clouds are managed today. A building block of cloud computing is task consolidation, or resource over-subscription, as a means to exploit statistical multiplexing. The underlying rationale for consolidation is that more resources than actually available can be offered. Since simultaneous peak consumption of allotted resources by all tasks happens very rarely, if ever, the overall capacity of the cloud will not be exceeded. Scale flexibility at the degree offered by SnowFlock is very hard to achieve in clouds using consolidation principles. Currently, consolidation of idle VMs is performed primarily by suspending them to secondary storage, or by live-migrating them to highly over-subscribed hosts where time sharing of the processor by idle tasks is not an issue [Steinder et al. 2007; Wood et al. 2007]. However, when an idle VM transitions to a stage of activity, sharing due to consolidation becomes harmful to performance. Resuming and/or live-migrating a consolidated VM to a host where it can execute with no sharing is costly. These methods are plagued by the same problems as booting from images kept in backend storage: high latency, and lack of scalability, due to an abuse of the network interconnect by pushing multiple GB of VM state.

Beyond its appealing computing model, the performance characteristics of SnowFlock make it an attractive choice to rethink the current consolidation-based model, and effectively do away with the need for consolidation in many cases. In this paper we show that for a large set of workloads including MPI jobs, rendering, stock pricing, web servers, and sequence alignment, SnowFlock is able to instantiate VMs at a very frugal cost, in terms of time and I/O involved for state transmission. Thus, for many workloads, there is no need to keep idle VMs lurking around the cloud, unless they accomplish a specific task; new generic computing elements can be instantiated quickly and inexpensively. In an ideal SnowFlock cloud, there are no idle machines, since a VM is destroyed after it finishes a compute task, its resources are reclaimed and can be immediately assigned to a new VM. However, a resource of interest that the VM may have accrued throughout its execution is the contents of its buffer cache. These contents can survive VM destruction if administered in collaboration with the host OS. This mechanism is in consonance with the file system agnosticism proposed elsewhere [el Malek et al. 2009].

SnowFlock's objective is performance rather than reliability. While memory-on-demand provides significant performance gains, it imposes a long-lived dependency on a single source of VM state. Another aspect of our future work involves studying how to push VM state in the background to achieve a stronger failure model, without sacrificing the speed of cloning or low runtime overhead.

Finally, we wish to explore applying the SnowFlock techniques to wide-area VM migration. This would allow, for example, "just-in-time" cloning of VMs over geographical distances to opportunistically exploit cloud resources. We foresee modifications to memory-on-demand to batch multiple pages on each update, replacement of IP-multicast, and use of content-addressable storage at the destination sites to obtain local copies of frequently used state (e.g. libc).

ACKNOWLEDGMENTS

We thank Orran Krieger, Bianca Schroeder, Steven Hand, Angela Demke-Brown, Ryan Lilien, Olga Irzak, Alexey Tumanov, Michael Mior, Tim Smith, Vivek Lakshmanan, Lionel Litty, Niraj Tolia, Matti Hiltunen, Kaustubh Joshi, Mary Fernandez, and Tim Marsland for their input throughout this project. We thank the anonymous reviewers and the editor for their helpful and detailed comments.

REFERENCES

Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. 1997. Gapped BLAST and PSI–BLAST: a New Generation of Protein Database Search Programs. *Nucleic Acids Research 25,* 17 (Sept.), 3389–3402.

Alvarez, G. A., Borowsky, E., Go, S., Romer, T. R., Szendy, R. B. B., Golding, R., Merchant, A., Spasojevic, M., Veitch, A., and Wilkes, J. 2001. Minerva: an Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Transactions on Computer Systems 19,* 4 (Nov.), 483–518.

Amazon.com. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

ANL. Argonne National Laboratory MPICH2. `http://www.mcs.anl.gov/research/projects/mpich2/`.

Apache A. The Apache Software Foundation – Hadoop. `http://hadoop.apache.org/core/`.

Apache B. The Apache Software Foundation – Hadoop Distributed File System: Architecture and Design. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

Apodaka, A. A. and Gritz, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Academic Press.

Aqsis.org. Aqsis Renderer – Freedom to Dream. `http://aqsis.org/`.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the Art of Virtualization. In *Proc. 17th Symposium on Operating Systems Principles (SOSP)*. Bolton Landing, NY, 164–177.

Bartolini, N., Bongiovanni, G., and Silvestri, S. 2009. Self-* Through Self-Learning: Overload Control for Distributed Web Systems. *Computer Networks 53,* 5, 727–743.

Bugnion, E., Devine, S., and Rosenblum, M. 1997. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proc.* 16th *Symposium on Operating Systems Principles (SOSP)*. Saint Malo, France, 143–156.

Carns, P. H., Ligon III, W. B., Ross, R. B., and Thakur, R. 2000. PVFS: A Parallel File System for Linux Clusters. In *Proc. 4th Annual Linux Showcase and Conference*. Atlanta, GA, 317–327.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. 2000. *Parallel Programming in OpenMP*. Elsevier.

Chase, J. S., Irwin, D. E., Grit, L. E., Moore, J. D., and Sprenkle, S. E. 2003. Dynamic Virtual Clusters in a Grid Site Manager. In *Proc. 12th IEEE International Symposium on High Performance Distributed Computing (HPDC)*. Washington, DC, 90–103.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. 2005. Live Migration of Virtual Machines. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA.

Cluster Resources. Moab Cluster Suite. `http://www.clusterresources.com/pages/products/moab-cluster-suite.php`.

Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. 5th Symposium on Networked Systems Design and Implementation (NSDI)*. San Francisco, CA.

Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*. San Francisco, CA, 137–150.

Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA, 211–224.

EBI. European Bioinformatics Institute – ClustalW2. `http://www.ebi.ac.uk/Tools/clustalw2/index.html`.

el Malek, M. A., Wachs, M., Cipar, J., Sanghi, K., Ganger, G. R., Gibson, G. A., and Reiter, M. K. 2009. File System Virtual Appliances: Portable File System Implementations. Tech. Rep. CMU-PDL-09-102, Carnegie Mellon University.

Elnikety, S., Nahum, E., Tracey, J., and Zwaenepoel, W. 2004. A Method for Transparent Admission Control and Request Scheduling in e-Commerce Web Sites. In *Proc. 13th International Conference on World Wide Web (WWW)*. New York City, NY, 276–286.

Emeneker, W. and Stanzione, D. 2007. Dynamic Virtual Clustering. In *Proc. IEEE International Conference on Cluster Computing (Cluster)*. Austin, TX, 84–90.

Foster, I., Freeman, T., Keahey, K., Scheftner, D., Sotomayor, B., and Zhang, X. 2006. Virtual Clusters for Grid Communities. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. Singapore, 513–520.

Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based Scalable Network Services. In *Proc. 16th Symposium on Operating Systems Principles (SOSP)*. Saint Malo, France, 78–91.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proc., 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 97–104.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. 1994. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press.

Gentzsch, W. 2001. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. Brisbane, Australia, 35–36.

GoGrid. GoGrid Cloud Hosting. `http://www.gogrid.com/`.

Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. 2008. Difference Engine: Harnessing Memory Redundancy in Virtual Machine. In *Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, 309–322.

Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., and Lepreau, J. 2008. Large-scale Virtualization in the Emulab Network Testbed. In *Proc. USENIX Annual Technical Conference*. Boston, MA, 113–128.

Hibler, M., Stoller, L., Lepreau, J., Ricci, R., and Barb, C. 2003. Fast, Scalable Disk Imaging with Frisbee. In *Proc. USENIX Annual Technical Conference*. San Antonio, TX, 283–296.

Higgins, D., Thompson, J., and Gibson, T. 1994. CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment Through Sequence Weighting, Position-specific Gap Penalties and Weight Matrix Choice. *Nucleic Acids Research 22,* 22 (Nov.), 4673–4680.

Hines, M. and Gopalan, K. 2009. Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging And Dynamic Self-Ballooning. In *Proc. Conference on Virtual Execution Environments (VEE)*. Washington, DC, 51–60.

Huang, W., Gao, Q., Liu, J., and Panda, D. K. 2007. High Performance Virtual Machine Migration with RDMA over Modern Interconnects. In *Proc. IEEE International Conference on Cluster Computing (Cluster)*. Austin, TX, 11–20.

Huelsenbeck, J. P. and Ronquist, F. 2001. MRBAYES: Bayesian Inference of Phylogenetic Trees. *Bioinformatics 17,* 8, 754–755. `http://mrbayes.csit.fsu.edu/`.

Kawachiya, K., Ogata, K., Silva, D., Onodera, T., Komatsu, H., and Nakatani, T. 2007. Cloneable JVM: A New Approach to Start Isolated Java Applications Faster. In *Proc. Conference on Virtual Execution Environments (VEE)*. San Diego, CA, 1–11.

Kozuch, M. and Satyanarayanan, M. 2002. Internet Suspend/Resume. In *Proc. 4th Workshop on Mobile Computing Systems and Applications (WMCSA)*. Callicoon, NY, 40–46.

Lagar-Cavilla, H. A., Tolia, N., de Lara, E., Satyanarayanan, M., and O'Hallaron, D. 2007. Interactive Resource-Intensive Applications Made Easy. In *Proc. ACM/IFIP/USENIX 8th International Middleware Conference*. Newport Beach, CA, 143–163.

Li, K.-B. 2003. ClustalW-MPI: ClustalW Analysis Using Distributed and Parallel Computing. *Bioinformatics 19,* 12, 1585–1586. `http://www.bii.a-star.edu.sg/achievements/applications/clustalw/index.php`.

Lin, B. and Dinda, P. 2005. VSched: Mixing Batch and Interactive Virtual Machines Using Periodic Real-time Scheduling. In *Proc. ACM/IEEE Conference on Supercomputing*. Seattle, WA, 8–20.

Litty, L., Lagar-Cavilla, H. A., and Lie, D. 2008. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proc. 17th USENIX Security Symposium*. San Jose, CA, 243–258.

McLoughlin, M. 2008. The QCOW2 Image Format. `http://people.gnome.org/~markmc/qcow-image-format.html`.

McNett, M., Gupta, D., Vahdat, A., and Voelker, G. 2007. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In *Proc. 21st Large Installation System Adminstration Conference (LISA)*. Dallas, TX, 167–181.

Meyer, D., Aggarwal, G., Cully, B., Lefebvre, G., Hutchinson, N., Feeley, M., and Warfield, A. 2008. Parallax: Virtual Disks for Virtual Machines. In *Proc. Eurosys 2008*. Glasgow, Scotland, 41–54.

Microsoft. 2009. Virtual Hard Disk Image Format Specification. `http://technet.microsoft.com/en-us/virtualserver/bb676673.aspx`.

MOSSO. Mosso Cloud Service. http://www.mosso.com/.

MURPHY, M. A., KAGEY, B., FENN, M., AND GOASGUEN, S. 2009. Dynamic Provisioning of Virtual Organization Clusters. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. Shaghai, China, 364–371.

NATHUJI, R. AND SCHWAN, K. 2007. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In *Proc.* 21st *Symposium on Operating Systems Principles (SOSP)*. Skamania Lodge, WA, 265–278.

NELSON, M., LIM, B., AND HUTCHINS, G. 2005. Fast Transparent Migration for Virtual Machines. In *Proc. USENIX Annual Technical Conference*. Anaheim, CA, Short paper.

NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P., AND FLINN, J. 2006. Rethink the Sync. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, 1–14.

NISHIMURA, H., MARUYAMA, N., AND MATSUOKA, S. 2007. Virtual Clusters on the Fly – Fast, Scalable and Flexible Installation. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. Rio de Janeiro, Brazil, 549–556.

NORRIS, J., COLEMAN, K., FOX, A., AND CANDEA, G. 2004. OnCall: Defeating Spikes with a Free-Market Application Cluster. In *Proc. 1st International Conference on Autonomic Computing (ICAC)*. New York City, NY, 198–205.

OPEN VSWITCH. An Open Virtual Switch. http://openvswitch.org.

PIXAR. RenderMan. https://renderman.pixar.com/.

PLATFORM COMPUTING. 2006. Platform Enterprise Grid Orchestrator (Platform EGO). http://www.platform.com/Products/platform-enterprise-grid-orchestrator.

PNFS.COM. Parallel NFS. http://www.pnfs.com/.

QUANTLIB.ORG. QuantLib: a Free/Open-source Library for Quantitative Finance. http://quantlib.org/index.shtml.

RANJAN, S., ROLIA, J., FU, H., AND KNIGHTLY, E. 2003. Qos-Driven Server Migration for Internet Data Centers. In *Proc. 10th International Workshop on Quality of Service (IWQoS)*. Miami Beach, FL, 3–12.

RED HAT. LVM2 Resource Page. http://sources.redhat.com/lvm2/.

RITCHIE, D. AND THOMPSON, K. 1974. The UNIX Time-sharing System. *Communications of the ACM 17,* 7 (July), 365–375.

RUMBLE, S. M., LACROUTE, P., DALCA, A. V., FIUME, M., SIDOW, A., AND BRUDNO, M. 2009. SHRiMP: Accurate Mapping of Short Color-space Reads. *PLoS Computational Biology 5,* 5 (May).

RUTH, P., MCGACHEY, P., JIANG, J., AND XU, D. 2005. VioCluster: Virtualization for Dynamic Computational Domains. In *Proc. IEEE International Conference on Cluster Computing (Cluster)*. Burlington, MA, 1–10.

SAMBA.ORG. distcc: a Fast, Free Distributed C/C++ Compiler. http://distcc.samba.org/.

SAPUNTZAKIS, C. P., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M. S., AND ROSENBLUM, M. 2002. Optimizing the Migration of Virtual Computers. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA, 377–390.

SPEC. 2005. Standard Performance Evaluation Corporation SPECweb2005. http://www.spec.org/web2005/.

STEINDER, M., WHALLEY, I., CARRERA, D., GAWEDA, I., AND CHESS, D. 2007. Server Virtualization in Autonomic Management of Heterogeneous Workloads. In *Proc. 10th Integrated Network Management (IM) Conference*. Munich, Germany, 139–148.

STONE, J. Tachyon Parallel / Multiprocessor Ray Tracing System. http://jedi.ks.uiuc.edu/~johns/raytracer/.

SUN MICROSYSTEMS. Solaris ZFS - Redefining File Systems as Virtualised Storage. http://nz.sun.com/learnabout/solaris/10/ds/zfs.html.

SURIE, A., LAGAR-CAVILLA, H. A., DE LARA, E., AND SATYANARAYANAN, M. 2008. Low-Bandwidth VM Migration via Opportunistic Replay. In *Proc. 9th Workshop on Mobile Computing Systems and Applications (HotMobile)*. Napa Valley, CA, 74–79.

Tolia, N., Kaminsky, M., Andersen, D. G., and Patil, S. 2006. An Architecture for Internet Data Transfer. In *Proc. 3*rd *Symposium on Networked Systems Design and Implementation (NSDI)*. San Jose, CA, 253–266.

Tolia, N., Kozuch, M., Satyanarayanan, M., Karp, B., Perrig, A., and Bressoud, T. 2003. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proc. USENIX Annual Technical Conference*. San Antonio, TX, 127–140.

Uhlig, V., LeVasseur, J., Skoglund, E., and Dannowski, U. 2004. Towards Scalable Multiprocessor Virtual Machines. In *Proc. 3*rd *Virtual Machine Research and Technology Symposium*. San Jose, CA, 43–56.

UofT. University of Toronto – SnowFlock: Swift VM Cloning for Cloud Computing. http://sysweb.cs.toronto.edu/snowflock.

Urgaonkar, B. and Shenoy, P. 2008. Cataclysm: Scalable Overload Policing for Internet Applications. *Journal of Network and Computing Applications 31,* 4 (Nov.), 891–920.

Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. 2008. Agile Dynamic Provisioning of Multi-tier Internet Applications. *ACM Transactions in Autonomous Adaptive Systems 3,* 1, 1–39.

VASP - GROUP. VASP – Vienna Ab initio Simulation Package, Theoretical Physics Department, Vienna. http://cms.mpi.univie.ac.at/vasp/.

VMware. vSphere Resource Management Guide. www.vmware.com/pdf/vsphere4/r41/vsp_41_resource_mgmt.pdf.

Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A., Voelker, G., and Savage, S. 2005. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. In *Proc.* 20th *Symposium on Operating Systems Principles (SOSP)*. Brighton, UK, 148–162.

Waldspurger, C. A. 2002. Memory Resource Management in VMware ESX Server. In *Proc. 5*th *Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA, 181–194.

Warfield, A., Hand, S., Fraser, K., and Deegan, T. 2005. Facilitating the Development of Soft Devices. In *Proc. USENIX Annual Technical Conference*. Anaheim, CA, Short paper.

Whitaker, A., Shaw, M., and Gribble, S. D. 2002. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5*th *Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA, 195–209.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. 5*th *Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA, 255–270.

Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M. 2007. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proc. 4*th *Symposium on Networked Systems Design and Implementation (NSDI)*. Cambridge, MA, 229–242.

Youseff, L., Wolski, R., Gorda, B., and Krintz, C. 2006. Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. In *Proc. 2*nd *International Workshop on Virtualization Technology in Distributed Computing (VTDC)*. Washington, DC, 1–8.

Zayas, E. 1987. Attacking the Process Migration Bottleneck. In *Proc. 11*th *ACM Symposium on Operating System Principles (SOSP)*. Austin, TX, 13–24.

Zhang, H., Jiang, G., Yoshihira, K., Chen, H., and Saxena, A. 2009. Resilient Workload Manager: Taming Bursty Workload of Scaling Internet Applications. In *Proc. 6th International Conference on Autonomic Computing and Communications, Industry Session (ICAC-INDST)*. Barcelona, Spain, 19–28.

Zhang, X., Keahey, K., Foster, I., and Freeman, T. 2005. Virtual Cluster Workspaces for Grid Applications. Tech. Rep. TR-ANL/MCS-P1246-0405, University of Chicago. Apr.