SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing

H. Andrés Lagar-Cavilla, Joseph A. Whitney, Adin Scannell, Philip Patchin, Stephen M. Rumble,
Eyal de Lara, Michael Brudno, M. Satyanarayanan†

University of Toronto and Carnegie Mellon University†
`http://sysweb.cs.toronto.edu/snowflock`

Abstract

Virtual Machine (VM) fork is a new cloud computing abstraction that instantaneously clones a VM into multiple replicas running on different hosts. All replicas share the same initial state, matching the intuitive semantics of stateful worker creation. VM fork thus enables the straightforward creation and efficient deployment of many tasks demanding swift instantiation of stateful workers in a cloud environment, e.g. excess load handling, opportunistic job placement, or parallel computing. Lack of instantaneous stateful cloning forces users of cloud computing into ad hoc practices to manage application state and cycle provisioning. We present SnowFlock, our implementation of the VM fork abstraction. To evaluate SnowFlock, we focus on the demanding scenario of services requiring on-the-fly creation of hundreds of parallel workers in order to solve computationally-intensive queries in seconds. These services are prominent in fields such as bioinformatics, finance, and rendering. SnowFlock provides sub-second VM cloning, scales to hundreds of workers, consumes few cloud I/O resources, and has negligible runtime overhead.

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

Keywords Virtualization, Cloud Computing

1. Introduction

Cloud computing is transforming the computing landscape by shifting the hardware and staffing costs of managing a computational center to third parties such as Yahoo! or Amazon [EC2]. Small organizations and individuals are now able to deploy world-scale services: all they need to pay is the marginal cost of actual resource usage. Virtual machine (VM) technology is widely adopted as an enabler of cloud computing. Virtualization provides many benefits, including security, performance isolation, ease of management, and flexibility of running in a user-customized environment.

A major advantage of cloud computing is the ability to use a variable number of physical machines and VM instances depending on the needs of the problem. For example a task may need only a single CPU during some phases of execution but may be capable of leveraging *hundreds* of CPUs at other times. While current cloud APIs allow for the instantiation of new VMs, their lack of agility fails to provide users with the full potential of the cloud model. Instantiating new VMs is a slow operation (typically taking “minutes” [EC2]), and the new VMs originate either as fresh boots or replicas of a template VM, unaware of the current application state. This forces cloud users into employing ad hoc solutions that require considerable developer effort to explicitly propagate application state and waste resources by pre-provisioning worker VMs that remain mostly idle. Moreover, idle VMs are likely to be consolidated and swapped out [Steinder 2007, Wood 2007], incurring costly migration delays before they can be used.

We introduce VM fork, a clean abstraction that simplifies development and deployment of cloud applications that dynamically change their execution footprint. VM fork allows for the *rapid* (< 1 second) instantiation of *stateful* computing elements in a cloud environment. While VM fork is similar in spirit to the familiar UNIX process fork, in that the child VMs receive a copy of all of the state generated by the parent VM prior to forking, it is different in three fundamental ways. First, our VM fork primitive allows for the forked copies to be instantiated on a set of different physical machines, enabling the task to take advantage of large compute clusters. In contrast, previous work [Vrable 2005] is limited to cloning VMs within the same host. Second, we have made our primitive *parallel*, enabling the creation of multiple child VMs with a single call. Finally, our VM fork replicates all of the processes and threads of the originating VM. This en-

EuroSys'09, April 1–3, 2009, Nuremberg, Germany.
Copyright © 2009 ACM 978-1-60558-482-9/09/04. . . $5.00

(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)
```

Figure 1: 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.

ables effective replication of multiple cooperating processes, e.g. a customized LAMP (Linux/Apache/MySql/Php) stack.

VM fork enables the trivial implementation of several useful and well-known patterns that are based on stateful replication, e.g., inheriting initialized data structures when spawning new workers. Pseudocode for four of these is illustrated in Figure 1 – sandboxing of untrusted code, instantiating new worker nodes to handle increased load (e.g. due to flash crowds), enabling parallel computation, and opportunistically utilizing unused cycles with short tasks. All four patterns exploit fork's ability to create *stateful* workers, and further, they all exploit fork's ability to *instantaneously* create workers.

SnowFlock, our implementation of the VM fork abstraction, provides swift parallel stateful VM cloning with little runtime overhead and frugal consumption of cloud I/O resources, leading to good scalability. SnowFlock takes advantage of several key techniques. First, SnowFlock utilizes *lazy state replication* to minimize the amount of state propagated to the child VMs. Lazy replication allows for extremely fast instantiation of clones by initially copying the minimal necessary VM data, and transmitting only the fraction of the parent's state that clones actually need. Second, a set of *avoidance heuristics* eliminate substantial superfluous memory transfers for the common case of clones allocating new private state. Finally, exploiting the likelihood of child VMs to execute very similar code paths and access common data structures, we use a *multicast distribution* technique for VM state that provides scalability and prefetching.

We evaluated SnowFlock by focusing on a demanding instance of Figure 1 (b): interactive parallel computation, in which a VM forks multiple workers in order to carry out a short-lived, computationally-intensive parallel job. We have conducted experiments with applications from bioinformatics, quantitative finance, rendering, and parallel compilation. These applications are deployed as Internet services [NCBI, EBI] that leverage mass parallelism to provide interactive (tens of seconds) response times to complex queries: find candidates similar to a gene, predict the outcome of stock options, render an animation, etc. On experiments conducted with 128 processors, SnowFlock achieves speedups coming within 7% or better of optimal execution, and offers subsecond VM fork irrespective of the number of clones. SnowFlock is an order of magnitude faster and sends two orders of magnitude less state than VM fork based on suspend/resume or migration.

2. VM Fork

The VM fork 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.

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.

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. We envision 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.

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 same virtual subnet as the VM from which it was created. Child VMs cannot communicate with hosts outside this virtual network. Two aspects of our design deserve further comment. First, the user must be conscious of the IP reconfiguration semantics: for instance, network shares must be (re)mounted after cloning. Second, we provide a NAT layer to allow the clones to connect to certain external IP addresses. Our NAT performs firewalling 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.

3. 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 GBs 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 2 illustrates this by plotting the cost of suspending and resuming a 1GB VM to an increasing number of hosts over NFS (see Section 5 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 2005, VMotion], a popular mechanism for consolidating VMs in clouds [Steinder 2007, Wood 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 4.5) to leverage parallelism in the network hardware. Multicast delivers state simultaneously to all hosts. Scalability in Figure 2 is vastly improved, but overhead is still in the range of minutes. To move beyond this, *we must substantially reduce the total amount of VM state pushed over the network.*

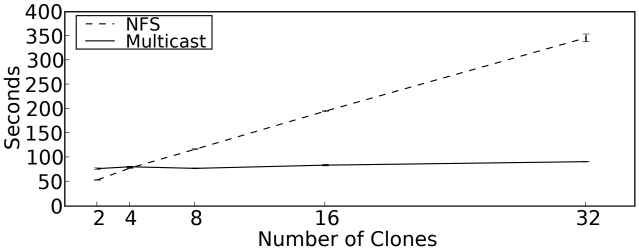


Figure 2: Latency for forking a 1GB VM by suspending and distributing the image over NFS and multicast.

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.

The first two insights led to the design of *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 only 0.1% of the state of the parent, and that children tend to only require a fraction of the original memory image of the parent. Further, 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. We show in Section 5 that this optimization can reduce communication drastically to a mere 40MBs for application footprints of 1GB (4%!). Whereas these observations are based on our work with parallel workloads, they are likely to hold in other domains where a parent node spawns children as workers that execute limited tasks, e.g., load handling in web services.

Compared to ballooning [Waldspurger 2002], memory-on-demand is a non-intrusive approach that reduces state transfer without altering the behaviour of the guest OS. 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 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 provide runtime stateful cloning, since all new VMs are copies of a frozen template.

To take advantage of high correlation across memory accesses of the children (insight iv) and 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.

4. SnowFlock Implementation

SnowFlock is our implementation of the VM fork primitive. SnowFlock is an open-source project [SnowFlock] built on the Xen 3.0.3 VMM [Barham 2003]. Xen consists of a hypervisor running at the highest processor privilege level, controlling the execution of domains (VMs). The domain kernels are paravirtualized, i.e. aware of virtualization, and interact with the hypervisor through a hypercall interface. A privileged VM (domain0) has control over hardware devices and manages the state of all other domains.

SnowFlock is implemented as a combination of modifications to the Xen VMM and daemons that run in domain0. The SnowFlock daemons form a distributed system that controls the life-cycle of VMs, by orchestrating their cloning and deallocation. SnowFlock defers policy decisions, such as resource accounting and the allocation of VMs to physical hosts, to suitable cluster management software via a plug-in architecture. SnowFlock currently supports allocation management with Platform EGO [Platform] and Sun Grid Engine [Gentzsch 2001]. Throughout this paper we use a simple internal resource manager which tracks memory and CPU allocations on each physical host.

SnowFlock's VM fork implementation is based on lazy state replication combined with avoidance heuristics to minimize state transfer. In addition, SnowFlock leverages multicast to propagate state in parallel and exploit the substantial temporal locality in memory accesses across forked VMs to provide prefetching. SnowFlock uses four mechanisms to fork a VM. First, the parent VM is temporarily suspended to produce a *VM descriptor*: a small file that contains VM metadata and guest kernel memory management data. The VM descriptor is then distributed to other physical hosts to spawn new VMs; the entire operation is complete in subsecond time. Second, our memory-on-demand mechanism, *memtap*, lazily fetches additional VM memory state as execution proceeds. Third, the *avoidance heuristics* leverage the cooperation of the guest kernel to substantially reduce the amount of memory that needs to be fetched on demand. Finally, our multicast distribution system *mcdist* is used to deliver VM state simultaneously and efficiently, as well as providing implicit pre-fetching.

The next subsection describes the SnowFlock API. We then describe in detail each of the four SnowFlock mechanisms. For each we present micro-benchmark results that show their effectiveness (see Section 5 for testbed details). We finish this section by discussing the specifics of the virtual I/O devices of a SnowFlock VM, namely the virtual disk and network isolation implementations.

4.1 API

- **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 ≤ n clones.
- **sf_clone(ticket)**: Clones, using the allocation in the ticket. Returns the clone ID, 0 ≤ ID ≤ m.
- **sf_exit()**: For children (1 ≤ ID ≤ 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, immediately terminates all children in ticket and discards the ticket.

Table 1: The SnowFlock VM Fork API

Table 1 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. 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 behaviour 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. 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**.

The API calls from Table 1 are available to applications via a SnowFlock client library, with C and Python bindings. The client library marshals API calls and communicates them to the SnowFlock daemon running on domain0 over a shared memory interface.

While the SnowFlock API is simple and flexible, it nonetheless demands modification of existing code bases. A SnowFlock-friendly implementation of the widely used Message Passing Interface (MPI) library allows a vast corpus of unmodified parallel applications to use SnowFlock's capabilities. Based on mpich2 [MPICH], our implementation replaces the task-launching subsystem of the library by one which invokes **sf_clone** to create the desired number of clones. Appropriately parameterized 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. Future work plans include performing a similar adaptation of the MapReduce toolkit [Dean 2004].

4.2 VM Descriptors

A *VM Descriptor* is a condensed VM image that allows swift VM replication to a separate physical host. Construction of a VM descriptor starts by spawning a thread in the VM kernel that quiesces its I/O devices, deactivates all but one of the virtual processors (VCPUs), and issues a hypercall suspending the VM's execution. When the hypercall succeeds, a privileged process in domain0 maps the suspended VM memory to populate the descriptor. The descriptor contains: (1) metadata describing the VM and its virtual devices, (2) a few memory pages shared between the VM and the Xen hypervisor, (3) the registers of the main VCPU, (4) the Global Descriptor Tables (GDT) used by the x86 segmentation hardware for memory protection, and (5) the page tables of the VM.

The page tables make up the bulk of a VM descriptor. In addition to those used by the guest kernel, each process in the VM generally needs a small number of additional page tables. The cumulative size of a VM descriptor is thus loosely dependent on the number of processes the VM is executing. Entries in a page table are "canonicalized" before saving. They are translated from references to host-specific pages to frame numbers within the VM's private contiguous physical space ("machine" and "physical" addresses in Xen parlance, respectively). A few other values included in the descriptor, e.g. the `cr3` register of the saved VCPU, are also canonicalized.

The resulting descriptor is multicast to multiple physical hosts using the mcdist library we describe in Section 4.5, and used to spawn a clone VM 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 new mapping between VM-specific physical addresses and host machine addresses. The VM replica resumes execution, enables the extra VCPUs, and reconnects its virtual I/O devices to the new frontends.

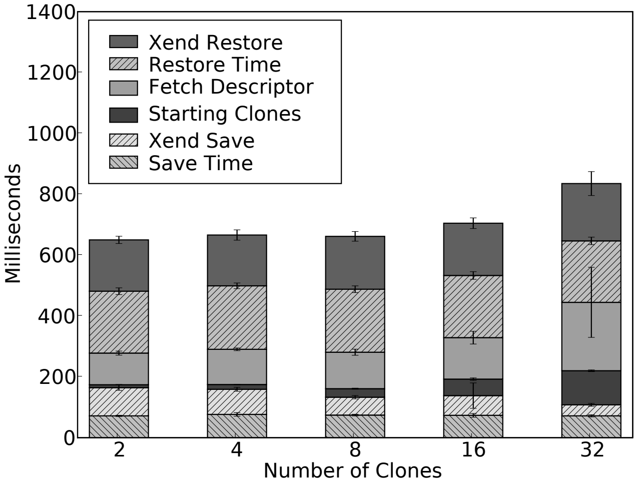


Figure 3: Fast Clone Creation. Legend order matches bar stacking from top to bottom.

Evaluation Figure 3 presents our evaluation of the VM descriptor mechanism, showing the time spent replicating a single-processor VM with 1 GB of RAM to n clones in n physical hosts. The average size of a VM descriptor for these experiments was 1051 ± 7 KB. The time to create a descriptor is "Save Time" (our code) plus "Xend Save" (recycled and unmodified Xen code). "Starting Clones" is the time spent distributing the order to spawn a clone to each host. Clone creation in each host is composed by "Fetch Descriptor" (wait for the descriptor to arrive), "Restore Time" (our code) and "Xend Restore" (recycled Xen code). At this point, all clones have begun execution.

Overall, VM replication is a fast operation, ranging in general from 600 to 800 milliseconds. Further, VM replication time is largely independent of the number of clones created. Larger numbers of clones introduce, however, a wider variance in the total cloning time. The variance is typically seen in the time to multicast the 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. Despite this small variance, the net result is sub-second VM cloning time irrespective of the size of the VM.

4.3 Memory-On-Demand

Immediately after being instantiated from a 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 clone VM's memory with state fetched from the parent, where an immutable copy of the VM's memory from the time of cloning is kept.

Memtap is a combination of hypervisor logic and a user-space domain0 process associated with the clone VM. Memtap implements a *copy-on-access* policy for the clone VM's memory. 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. The memtap process maps the missing page, fetches its contents from the parent, and notifies the hypervisor that the VCPU may be unpaused.

To allow the hypervisor to trap memory accesses to pages that have not yet been fetched, we use Xen shadow page tables. In shadow page table mode, the x86 register indicating the page table currently in use (`cr3`) 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. Shadow page table faults thus indicate that a page of memory is about to be accessed. If this is the first access to a page of memory that has not yet been fetched, the hypervisor notifies memtap. Fetches are also triggered by trapping modifications to page table entries, and accesses by domain0 of the VM's memory for the purpose of virtual device DMA.

On the parent VM, memtap implements a *copy-on-write* policy 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. Upon a write fault, the hypervisor duplicates the page and patches the mapping of the memtap server process to point to the duplicate. The parent VM is then allowed to continue execution.

Our implementation of memory-on-demand is SMP-safe. A shared bitmap is used by Xen and memtap to indicate the presence of VM memory pages. The bitmap is initialized when the VM is built from a descriptor, and is accessed in a lock-free manner with atomic (`test_and_set`, etc) operations. When trapping a shadow page table on-demand fill, Xen checks the present bit of the faulting page, notifies memtap, and buffers the write of the shadow entry. Another VCPU using the same page table entry will fault on the still empty shadow entry. 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 then queued, waiting for the first successful fetch of the missing page. When memtap notifies completion of the fetch, the present bit is set, pending shadow page table writes are applied, and all queued VCPUs are unpaused.

Evaluation To understand the overhead involved in our memory-on-demand subsystem, we devised a microbenchmark in which a VM allocates and fills in a number of memory pages, invokes SnowFlock to have itself replicated, and then touches each page in the allocated set. The results for multiple microbenchmark runs totaling ten thousand page fetches are displayed in Figure 4(a).

The overhead of page fetching is modest, averaging 275 μs with unicast (standard TCP). We split a page fetch operation into six components. “Page Fault” indicates the hardware page fault overheads caused by using shadow page tables. “Xen” is the cost of the Xen hypervisor shadow page table logic. “HV Logic” is the time consumed by our hypervisor logic: bitmap checking and SMP safety. “Dom0 Switch” is the time to context switch to the domain0 memtap process, while “Memtap Logic” is the time spent by the memtap internals, consisting mainly of mapping the faulting VM page. The bulk of the page-fetching time is spent in the sixth component, “Network”, which depicts the software (libc and Linux kernel TCP stack) and hardware overheads of remotely fetching the page contents over gigabit Ethernet. Our implementation is frugal and efficient, and the bulk of the overhead (82%) comes from the network stack.

4.4 Avoidance Heuristics

The previous section shows that memory-on-demand guarantees correct VM execution and is able to fetch pages of memory with speed close to bare TCP/IP. However, fetching pages from the parent still incurs an overhead that may prove excessive for many workloads. We thus augmented the VM kernel with two fetch-avoidance heuristics that allow us to bypass large numbers of unnecessary memory fetches, while retaining correctness.

The first heuristic optimizes the general case in which a clone VM allocates new state. The heuristic intercepts pages selected by the kernel's page allocator. The kernel page allocator is invoked when more memory is needed by a kernel subsystem, or by a user-space process, typically requested indirectly via a `malloc` call. The semantics of these operations imply that the recipient of the selected pages does not care about the pages' previous contents. If the pages have not yet been fetched, there is no reason to do so.

The second heuristic addresses the case where a virtual I/O device writes to the guest memory. Consider the case of 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. Again, there is no need to fetch this page.

The fetch-avoidance heuristics are implemented by mapping the memtap bitmap in the guest kernel's address space. When the kernel decides a page should not be fetched, it “fakes” the page's presence by setting the corresponding bit, and thus prevents Xen or memtap from fetching it.

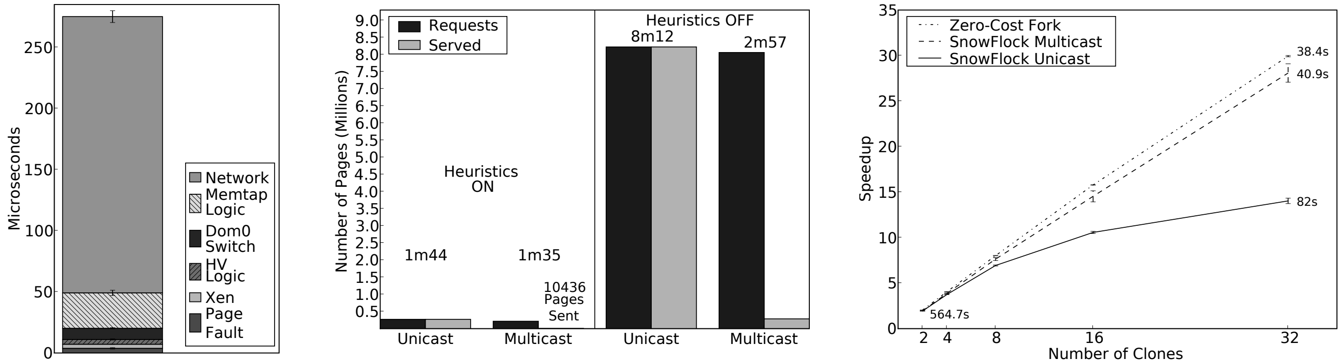
Evaluation We evaluate the effect of the guest avoidance heuristics using SHRiMP, one of the applications described in Section 5.1. Our SHRiMP macrobenchmark spawns *n* uniprocessor VM clones and runs on each a task that demands 1 GB of RAM. Figure 4(b) illustrates the results for 32 clones. While we also vary the choice of networking substrate between unicast and multicast, we study here the effect of the heuristics; we will revisit Figure 4(b) in the next subsection. Experiments with smaller memory footprints and different numbers of clones show similar results and are therefore not shown.

The avoidance heuristics result in substantial benefits, in terms of both runtime and data transfer. Nearly all of SHRiMP's memory footprint is allocated from scratch when the inputs 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 two orders of magnitude. With the heuristics, state transmissions to clones are reduced to 40 MBs, a tiny fraction (3.5%) of the VM's footprint.

4.5 Multicast Distribution

Mcdist is our multicast distribution system that efficiently provides data to all cloned VMs simultaneously. It accomplishes two goals that are not served by point-to-point communication. First, data needed by clones is often prefetched. Once a single clone requests a page, the response also reaches all other clones. Second, the load on the network is 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 clones to co-exist in the cloud.

The mcdist server design is minimalistic, containing only switch programming and flow control logic. No atomicity or ordering guarantees are given by the server and requests



(a) Page Fetching Overhead. Legend order matches bar stacking top to bottom.

(b) Effectiveness of Heuristics and Multicast. SHRiMP, 1GB footprint. Bars show aggregate page requests from 32 clones vs. pages sent. Labels show average benchmark completion times.

(c) Multicast Scales. BLAST, 256MB DB. Speedup vs. one thread, labels show absolute times.

Figure 4: Evaluation of SnowFlock Design Principles

are processed on a first-come, first-served basis. Ensuring reliability thus falls to receivers, through a timeout mechanism. We use IP-multicast in order to send data to multiple hosts simultaneously. IP-multicast is supported by most off-the-shelf commercial Ethernet hardware. Switches and routers maintain group membership information and simultaneously send a frame destined for a multicast group to all subscribed hosts. We believe this approach scales to large clusters; IP-multicast hardware is capable of scaling to thousands of hosts and multicast groups, automatically relaying multicast frames across multiple hops.

The mcdist clients are memtap processes, which will receive pages asynchronously and unpredictably in response to requests by fellow VM clones. For efficiency, memtap clients batch received pages until a threshold is hit, or a page that has been explicitly requested arrives. A single hypercall is invoked to map the pages in a batch. A threshold of 1024 pages has proven to work well in practice.

To maximize total goodput, the server uses flow control logic to limit its sending rate and avoid overwhelming busy clients. Both the server and clients estimate their send and receive rate using a weighted average of the number of bytes transmitted or received every ten milliseconds. Clients provide explicit feedback about their current rate to the server in request messages. The server increases its rate limit linearly and, when a loss is detected through a client request for data that has already been sent, the server scales its rate limit back. We found that scaling the rate back to three quarters of the estimated *mean* client receive rate works effectively.

Another server flow control mechanism is lockstep detection, which aims to leverage the similarity in memory access patterns across clones. For example, shortly after cloning, VM clones share the same code paths due to a deterministic sequence of kernel functions called during resumption of the suspended VM. Large numbers of identical page requests are generated at ostensibly the same time, i.e. in "lockstep". Thus, when multiple requests for the same page are received in succession the server ignores duplicate requests immediately following the first. If the identical requests are due to lost packets as opposed to lockstep, they will eventually be serviced when the request is retransmitted by the client.

Evaluation We evaluate the effects of multicast, revisiting the results obtained with SHRiMP. Recall that in Figure 4(b) we spawn 32 uniprocessor VM clones and run on each a SHRiMP task that demands 1 GB of RAM. Figure 4(b) shows that our multicast distribution's lockstep avoidance works effectively: lockstep-executing VMs issue simultaneous requests that are satisfied by a single response from the server. Hence the difference between the "Requests" and "Served" bars in the multicast experiments. Further, even under the extreme pressure of an uncooperative guest with disabled heuristics, the number of pages served is reduced dramatically, and extra overhead reduced to a minute.

Figure 4(c) shows the benefit of mcdist for a case where an important portion of memory state is needed after cloning and thus the avoidance heuristics cannot help. The figure shows results from an experiment conducted with NCBI BLAST (described in Section 5.1), which executes queries against a 256 MB portion of the NCBI genome database that the parent caches into memory before cloning. The figure shows speedup results for SnowFlock using unicast and multicast, and an idealized *zero-cost fork* configuration in which VMs have been previously allocated, with no cloning or state-fetching overhead. Mcdist achieves almost linear speedup, closely tracking the speedup exhibited with ideal execution, while unicast does not scale beyond 16 clones.

4.6 Virtual I/O Devices in SnowFlock

Outside of the four techniques addressing fast and scalable VM replication, SnowFlock provides a virtual disk for each clone VM and guarantees secure network isolation.

4.6.1 Virtual Disk

The virtual disks of SnowFlock VMs are implemented with a blocktap [Warfield 2005] driver. Multiple views of the virtual disk are supported by a hierarchy of copy-on-write (COW) slices located at the site where the parent VM runs. Each fork 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. Children access a sparse local version of the disk, with the state from the time of cloning fetched on demand from the disk server. The virtual disk exploits the same optimizations as the memory subsystem: unnecessary fetches during writes are avoided using heuristics, and the original disk state is provided to all clients simultaneously via multicast.

In our usage model, the virtual disk is used as the base root partition for the VMs. For data-intensive tasks, we envision serving data volumes to the clones through network file systems such as NFS, or suitable big-data filesystems such as Hadoop [Hadoop] or Lustre [Braam 2002]. The separation of responsibilities results in our virtual disk not being heavily exercised. Most work done by clones is processor intensive, writes do not result in fetches, and the little remaining disk activity mostly hits kernel caches. Our implementation largely exceeds the demands of many realistic tasks and did not cause any noticeable overhead for the experiments in Section 5.

4.6.2 Network Isolation

In order to prevent interference or eavesdropping between unrelated VMs on the shared network, either malicious or accidental, we employ a mechanism to isolate the network. Isolation is performed 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 identifiers. Simple filtering rules are used by all hosts to ensure that no packets delivered to a VM come from VMs that are not its parent or a sibling. Conversely, when a packet is delivered to a SnowFlock VM, the destination MAC address is rewritten to be as expected, rendering the entire process transparent. Additionally, a small number of special rewriting rules are required for protocols with payloads containing MAC addresses, such as ARP. Despite this, filtering and rewriting impose an imperceptible overhead while maintaining full IP compatibility.

5. Application Evaluation

Our evaluation of SnowFlock focuses on a particularly demanding scenario: the ability to deliver interactive parallel computation, in which a VM forks multiple workers to participate in a short-lived computationally-intensive parallel job. This scenario matches existing bioinformatics web services like BLAST [NCBI] or ClustalW [EBI], and other Internet services like render or compile-farms. Users interact with a web frontend and submit queries that are serviced by an embarrassingly parallel algorithm run on a compute cluster. These services are thus capable of providing *interactive* responses in the range of tens of seconds to computationally-demanding queries.

All of our experiments were carried out on a cluster of 32 Dell PowerEdge 1950 blade servers. Each host had 4 GB of RAM, 4 Intel Xeon 3.2 GHz cores, and a Broadcom NetXtreme II BCM5708 gigabit NIC. All machines were running the SnowFlock prototype based on Xen 3.0.3, with paravirtualized Linux 2.6.16.29 running as the OS for both host and guest VMs. The VMs were configured with 1124 MB of RAM. All machines were connected to two daisy-chained Dell PowerConnect 5324 gigabit switches. All results reported are the means of five or more runs, and error bars depict standard deviations.

5.1 Applications

We tested SnowFlock with 3 typical applications from bioinformatics and 3 applications representative of the fields of graphics rendering, parallel compilation, and financial services. We devised workloads for these applications with runtimes ranging above an hour on a uniprocessor machine, but which can be reduced to interactive response times if over a hundred processors are available. Application experiments are driven by a workflow shell script that clones the VM and launches an application process properly parameterized according to the clone ID. The exception to this technique is ClustalW, where we modify the application code directly.

NCBI BLAST [Altschul 1997] 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. We experimented with a BLAST search using 1200 short protein fragments from the sea squirt *Ciona savignyi* to query a 1.5GB portion of NCBI's non-redundant protein database. VM clones access the database files via an NFS share. Database access is parallelized across VMs, each reading a different segment, while query processing is parallelized across process-level clones within each VM.

SHRiMP [SHRiMP] is a tool for aligning large collections of very short DNA sequences ("reads") against a known 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. In our experiments we attempted to align 1.9 million 25 letter-long reads, extracted from a *Ciona savignyi*, to a 5.2 million letter segment of the known C. savignyi genome.

ClustalW [EBI] is a popular program for generating a *multiple alignment* of a collection of protein or DNA sequences.

Like BLAST, ClustalW is offered as a web service by organizations owning large computational resources [EBI]. ClustalW builds a guide tree using progressive alignment, a greedy heuristic requiring precomputation of comparisons between all pairs of sequences. The pairwise comparison is computationally intensive and embarrassingly parallel, since each pair of sequences can be aligned independently. After cloning, 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 allows these scores to be relayed to the parent, before joining the forked VMs. Using this implementation we conducted experiments performing guide-tree generation by pairwise alignment of 200 synthetic protein sequences of 1000 amino acids (characters) each.

QuantLib [Quantlib] is an open source toolkit widely used in quantitative finance. It provides models for stock trading, equity option pricing, risk analysis, etc. A typical quantitative finance program 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 processed 1024 equity options using a set of Monte Carlo, binomial and Black-Scholes models while 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.

Aqsis – Renderman [Aqsis] is an open source implementation of Pixar's RenderMan interface [Pixar], an industry standard widely used in films and television visual effects. Aqsis accepts scene descriptions produced by a modeler and specified in the RenderMan Interface Bitstream (RIB) language. Rendering is 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 2000].

Distcc [distcc] is software which distributes builds of C/C++ programs over the network for parallel compilation. Distcc is not embarrassingly parallel: actions are tightly coordinated by a parent farming out preprocessed files for compilation by children. Resulting object files are retrieved from the children for linking. Preprocessed code includes all relevant headers, thus simplifying requirements on children to just having the same version of the compiler. In our experiments we compile the Linux kernel version 2.6.16.29.

5.2 Results

We executed the above applications in SnowFlock, enabling the combination of VM fork and process fork to take advantage of our SMP hardware. For each application we spawn 128 threads of execution: 32 4-core SMP VMs on 32 physical hosts. We aim to answer the following questions:

- How does SnowFlock compare to other methods for instantiating VMs?
- How close does SnowFlock come to achieving optimal application speedup?
- How scalable is SnowFlock? How does it perform in cloud environments with multiple applications simultaneously and repeatedly forking VMs, even under adverse VM allocation patterns?

Comparison Table 2 illustrates the substantial gains SnowFlock provides in terms of efficient VM cloning and application performance. The table shows results for SHRiMP using 128 processors under three configurations: SnowFlock with all the mechanisms described in Section 4, 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 and an order of magnitude. 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 (1GB), SnowFlock is capable of transmitting in parallel little VM state. Experiments with our other benchmarks show similar results and are therefore not shown.

	Time (s)	State (MB)
SnowFlock	70.63 ± 0.68	41.79 ± 0.7
S/R over multicast	157.29 ± 0.97	1124
S/R over NFS	412.29 ± 11.51	1124

Table 2: SnowFlock vs. VM Suspend/Resume. SHRiMP, 128 threads. Benchmark time and VM state sent.

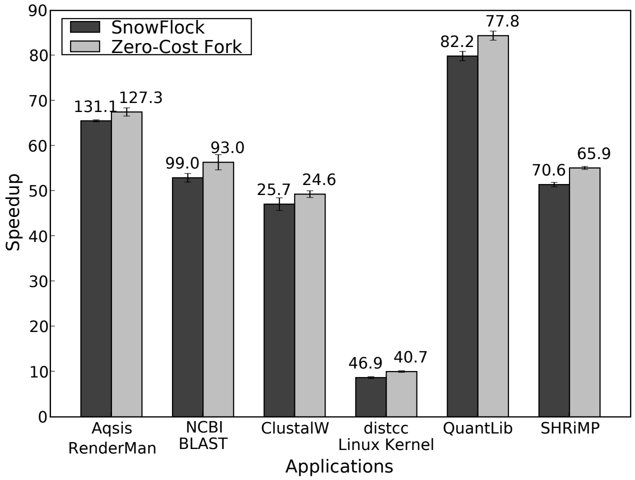


Figure 5: Application Benchmarks. Applications run with 128 threads: 32 VMs × 4 cores. Bars show speedup vs. a single thread zero-cost baseline. Labels show time to completion in seconds.

Application Performance Figure 5 compares SnowFlock to an optimal "zero-cost fork" baseline. We compare against a baseline with 128 threads to measure overhead, and against a baseline with one thread to measure speedup. 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 3.0.3 domains configured identically to SnowFlock VMs in terms of kernel version, disk contents, RAM, and number of processors.

SnowFlock performs extremely well and succeeds in reducing execution time from hours to tens of seconds for all the benchmarks. Moreover, SnowFlock achieves speedups that are very close to the zero-cost optimal, and comes within 7% of the optimal runtime. The overhead of VM replication and on-demand state fetching are small. ClustalW, in particular, yields the best results with less than two seconds of overhead for a 25 second task. This shows that tighter coupling of SnowFlock into application logic is beneficial.

Scale and Agility We address SnowFlock's capability to support multiple concurrent forking VMs. We launch four VMs that each simultaneously forks 32 uniprocessor VMs. To stress the system, after completing a parallel task, each parent VM joins and terminates its children and immediately launches another parallel task, repeating this cycle five times. Each parent VM runs 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 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. 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.

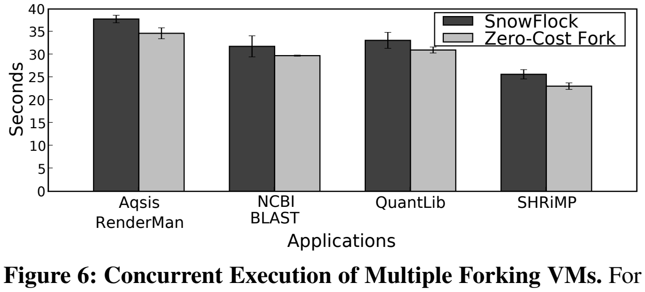


Figure 6: Concurrent Execution of Multiple Forking VMs. For each task we allocate 32 threads (32 VMs × 1 core), and cycle cloning, processing and joining repeatedly.

The results, shown in Figure 6, demonstrate that SnowFlock is capable of withstanding the increased demands of multiple concurrent forking VMs. As shown in section 4.5, this is mainly due to the small overall number of memory pages sent by the combined efforts of the guest heuristics and multicast distribution. The introduction of multiple 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. To summarize, SnowFlock is capable of forking VMs to perform a 32-host 40-seconds or less parallel computation, with five seconds or less of overhead, in spite of the pressure of adversarial allocations and repeated concurrent forking activity.

6. Related Work

To the best of our knowledge, we are the first group to address the problem of low-latency stateful replication of VMs to facilitate application deployment in cluster or cloud computing environments, including the ability to deliver instantaneous parallelism. A number of projects have explored the area of VM replication. The Potemkin project [Vrable 2005] implements a honeypot spanning a large IP address range. Honeypot machines are short-lived lightweight VMs cloned from a static template in the same machine with memory copy-on-write techniques. Potemkin does not address parallel applications and does not fork multiple VMs to different hosts. Remus [Cully 2008] provides instantaneous failover by keeping an up-to-date replica of a VM in a separate host.

Copy on reference, first used for process migration [Theimer 1985] in Accent [Zayas 1987], is a precursor to our memory-on-demand technique. Wide-area VM migration projects [Lagar-Cavilla 2007, Sapuntzakis 2002, Kozuch 2002] have used lazy copy-on reference for VM disk state. The low frequency and coarse granularity of access of secondary storage allows copying large batches of state over low-bandwidth high-latency links.

One objective of SnowFlock is to complement the capabilities of a shared computing platform. The Amazon Elastic Compute Cloud [EC2] is the foremost utility computing platform in operation today. While the details are not publicly known, we believe it follows industry standard techniques for the provisioning of VMs on the fly [Steinder 2007]: consolidation via memory sharing [Waldspurger 2002] or ballooning, resuming from disk, live migration [Clark 2005, VMotion], etc. Amazon's EC2 claims to instantiate multiple VMs in "minutes" – insufficient performance for the agility objectives of SnowFlock.

Similarly, work focusing on multiplexing a set of VMs on a physical cluster has typically resorted to standard techniques, without addressing issues studied here, such as programming primitives and performance capabilities. The term "virtual cluster" is used by many projects [Emeneker 2007, Foster 2006, Chase 2003] focusing on resource provisioning and management. We highlight Usher [McNett 2007], a manager of clusters of VMs that could be plugged in as a SnowFlock resource manager. Emulab [Hibler 2008] uses virtualization to instantiate dozens of nodes for a network emulation experiment. Experiments are long-lived, statically sized, and instantiation of the nodes takes tens to hundreds of seconds.

Emulab uses Frisbee [Hibler 2003] as a multicast distribution tool to apply disk images to nodes during experiment setup. Frisbee and mcdist differ in their domain-specific aspects: e.g. Frisbee uses filesystem-specific compression, not applicable to memory state; conversely, mcdist's lockstep detection does not apply to Frisbee's disk distribution. We view the use of high-speed interconnects such as RDMA or Infiniband [Huang 2007], if available, as a viable alternative to multicasting.

7. Conclusion and Future Directions

In this work we introduced the primitive of VM fork and SnowFlock, our Xen-based implementation. Matching the well-understood semantics of stateful worker creation, VM fork 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 IO resources. VM fork 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. While our evaluation focuses on interactive parallel Internet services, SnowFlock has broad applicability to other applications: flash crowd handling, execution of untrusted code components, instantaneous testing, etc.

SnowFlock makes use of two key observations. First, 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. Moreover, simple modifications to the guest kernel significantly reduce network traffic by eliminating the transfer of pages that will be overwritten. For our application these optimizations can drastically reduce the communication cost for forking a VM to a mere 40 MBs for application footprints of 1GB. Second, the locality of memory accesses across 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.

SnowFlock is an active open-source project [SnowFlock]. 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 2004]. For example, SnowFlock's transient clones cannot be entrusted with replicating and caching data due to their ephemeral natures. Allowing data replication to be handled by hosts enables benefits such as big-data file system agnosticism for the cloned VMs.

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 our 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).

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.

Acknowledgments

We thank the anonymous reviewers and our shepherd Orran Krieger for their helpful suggestions. We thank Bianca Schroeder, Ryan Lilien, Roy Bryant, Olga Irzak, and Charlotte Lin for their feedback and help with the SnowFlock project. We also thank Mike Kozuch, Scott Rixner and Y. Charlie Hu for their input on a draft of this paper.

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; and by Platform Computing Inc.

References

[Altschul 1997] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

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

[Aqsis] Aqsis: Open source 3D rendering solution adhering to the RenderMan standard. `http://aqsis.org/`.

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

[Braam 2002] P. J. Braam. The lustre storage architecture, 2002. http://www.lustre.org/docs/lustre.pdf.

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

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

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

[Dean 2004] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[distcc] samba.org. distcc: a fast, free distributed C/C++ compiler. http://distcc.samba.org/.

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

[EC2] Amazon.com. EC2: Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[Emeneker 2007] W. Emeneker and D. Stanzione. Dynamic Virtual Clustering. In *Proc. Cluster*, Austin, TX, September 2007.

[Foster 2006] I. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayor, and X. Zhang. Virtual Clusters for Grid Communities. In *Proc. Cluster Computing and the Grid*, Singapore, May 2006.

[Gentzsch 2001] W. Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Proc. 1st Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001.

[Hadoop] Apache.org. Hadoop. http://hadoop.apache.org/core/.

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

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

[Huang 2007] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High Performance Virtual Machine Migration with RDMA over Modern Interconnects. In *Proc. Cluster*, Austin, TX, September 2007.

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

[Lagar-Cavilla 2007] H. A. Lagar-Cavilla, N. Tolia, E. de Lara, M. Satyanarayanan, and D. O'Hallaron. Interactive Resource-Intensive Applications Made Easy. In *Proc. 8rm International Middleware Conference*, Newport Beach, CA, November 2007.

[McNett 2007] M. McNett, D. Gupta, A. Vahdat, and G. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In *Proc. 21st LISA*, Dallas, TX, November 2007.

[MPICH] Argonne National Laboratory. MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

[NCBI] National Center for Biotechnology Information. BLAST: Basic Local Alignment and Search Tool. http://blast.ncbi.nlm.nih.gov/Blast.cgi.

[Pixar] RenderMan. https://renderman.pixar.com/.

[Platform] Platform Computing. Platform EGO Home. http://my.platform.com/products/platform-ego-de.

[Quantlib] QuantLib: A Free/Open-source Library for Quantitative Finance. http://quantlib.org/index.shtml.

[Sapuntzakis 2002] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[SHRiMP] University of Toronto. SHRiMP: SHort Read Mapping Package. http://compbio.cs.toronto.edu/shrimp/.

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

[Steinder 2007] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess. Server Virtualization in Autonomic Management of Heterogeneous Workloads. In *Proc. 10th Integrated Network Management (IM) conference*, Munich, Germany, 2007.

[Theimer 1985] M. Theimer, K. Lantz, and D. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proc.* 10th *Symposium on Operating Systems Principles (SOSP)*, Orcas Island, WA, December 1985.

[VMotion] VMware. VMotion: Migrate Virtual Machines with Zero Downtime. http://www.vmware.com/products/vi/vc/vmotion.html.

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

[Waldspurger 2002] C. A. Waldspurger. Memory Resource Management in VMWare ESX Server. In *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, 2002.

[Warfield 2005] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

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

[Zayas 1987] E. Zayas. Attacking the Process Migration Bottleneck. In *Proc. 11th Symposium on Operating System Principles (SOSP)*, Austin, TX, November 1987.