THE POWER OF CLOUD COMPUTING COMES TO SMARTPHONES

Neeraj B. Bharwani
B.E. Student (Information Science and Engineering)
SJB Institute of Technology, Bangalore 60

Table of Contents

Introduction

Mobile Internet Devices (MIDs) such as the Nokia N810- and Moblin-based devices provide a rich, untethered Internet experience. With popularity, such devices have spawned new applications by a broader set of developers that go beyond the mobile staples of personal information management and music playback. Now, mobile users play games; capture, edit, annotate, and upload video; handle finances; and manage personal health and "wellness" (e.g. iPhone Heart Monitor and Diamedic). However, with greater application power comes greater responsibility for the mobile execution platform: it is now important to track memory leaks and runaway processes that drain power, avoid or detect malicious intrusions and private data disclosure, and manage applications that have expensive tastes for high volume data or advanced computational capabilities such as floating point or vector operations.

The problem with this ongoing demand is that everyone wants to stay in touch with the whole world at their fingertips which includes social networking, emails, pictures, video streaming, video chat, conferencing, playing music, making use of various applications, browsing the Internet, taking high resolution pictures, transferring data, and using Bluetooth, 3G, and LTE connectivity. Everyone wants all these features to be done by a single device which can easily fit in their pocket so that it can be accessed everywhere and can be carried anywhere. The main problem is hardware and battery limitations, which leads to slow processing speed that can freeze your phone or adversely affect multitasking capabilities and low battery backup due to power consumption in processing, multitasking, and providing energy to illuminate the large, high-resolution display.

For example, antivirus software operates by performing frequent complete scans of all files in a file system, and by imposing on access scans on the virtual memory contents of a process, including memory mapped files. On a smartphone, even users patient enough to wait until such a CPU- and I/O-intensive scan completed might still hit memory limits or run out of battery power. It only gets worse if one considers tools such as taint checking for data leak prevention, floating point and vector operations for mathematical or signal processing applications such as face detection in media, and so on.

On one hand, laptop, desktop and server resources are abundant, ubiquitous, and continuously reachable, as ensured by cloud computing, multi-core desktop processors, and plentiful wireless connectivity such as 3G, Ultra Wideband, Wi-Fi, and WiMax technologies. The disparity in capability between such computers and the untethered smartphone is high and persistent. On the other hand, technologies for replicating/migrating execution among connected computing substrates, including live

virtual machine migration and incremental check pointing, have matured and are used in production systems.

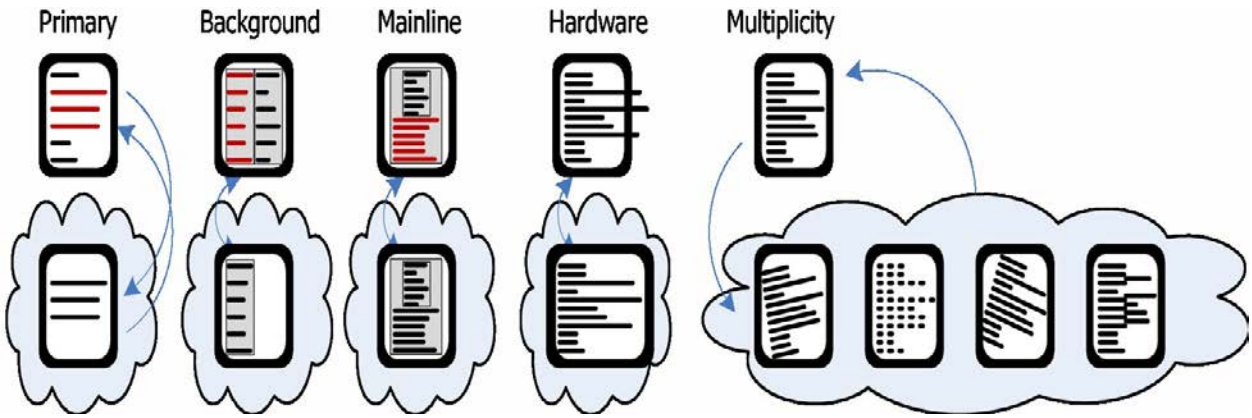
Need for Clone Cloud

Let's capitalize on this opportunity by proposing a simple idea: let the smartphone host its expensive, exotic applications. However, this requires an execution engine that augments the smartphone's capabilities by seamlessly off loading some tasks to a nearby computer, where they are executed in a cloned whole system image of the device, reintegrating the results in the smartphone's execution upon completion. This augmented execution overcomes smartphone hardware limitations and is provided (semi) automatically to applications whose developers need few or no modifications to their applications. Some augmentation can operate in the background, for asynchronous operations such as periodic file scans. For synchronous operations intrinsic to the application (e.g. a train of floating point instructions in the application code), augmentation can be performed by blocking progress on the smartphone until the result arrives from the clone in the cloud[1]. For concurrent operations to the application that operate "around" it (e.g. taint checking), augmentation can also be concurrent in the clone cloud or even speculative with the ability to undo operations on the smartphone according to the result from the clone.

While the ability to off load expensive computations from weak, mobile devices to powered, powerful devices has been recognized before, the novelty of this approach lies in using loosely synchronized virtualized or emulated replicas of the mobile device on the infrastructure to maintain two illusions: first, that the mobile user has a much more powerful, feature-rich device than she does in reality, and second, that the programmer is programming such a powerful, feature-rich device, without having to manually partition his application, explicitly provision proxies, or just dumb down the application.

Augmented Execution

The scope of augmented execution from the infrastructure is fairly broad.



- **Primary functionality outsourcing:** Computation-hungry applications such as speech processing, video indexing, and super resolution are automatically split, so that the user interface and other low octane processing is retained at the smartphone, while the high power, expensive computation is off-loaded to the infrastructure, synchronously. This is similar to designing the application as a client server service, where the infrastructure provides the service (e.g. the translation of speech to text) or as a thin client environment.

- **Background augmentation:** Unlike primary functionality outsourcing, this category deals with functionality that does not need to interact with users in a short time frame. Such is functionality that typically happens in the background, for instance, scanning the file system for viruses, indexing files for faster search, analyzing photos for common faces, crawling news web pages, and so on. In this case, entire processes can be marked (by the user or by the programmer) or automatically inferred as "background" processes and migrated to the infrastructure wholesale. Furthermore, off-loaded functionality can take on the role of a "virtual client." Even when the smartphone is turned off, the virtual client can continue to run background tasks. Later when the smartphone returns online, it can synchronize its state with the infrastructure.

- **Mainline augmentation:** This category sits between primary functionality outsourcing and background augmentation. Here the user may opt to run a particular application in a wrapped fashion, altering the method of its execution but not its semantics. Examples are private data leak detection (e.g. to taint check an application or application set), fault tolerance (e.g. to employ multi-variant execution analysis to protect the application from transparent bugs), or debugging (e.g.

keep track dynamically of allocated memory in the heap to catch memory leaks). Unlike background augmentation, mainline augmentation is interspersed in the execution of the application. Many possibilities exist: for example, when a decision point is reached in the taint check example, the application on the smartphone may block, perhaps causing the clone to rewind back to a known checkpoint and to execute forward with taint tracking before deciding.

Hardware augmentation: This category is interesting because it compensates for fundamental weaknesses of the smartphone platform, such as memory caps or other constraints and hardware peculiarities.

For demonstration, a file system scanning application is the Dalvik VM, the execution environment of the original Google Android phone (HTC G1). It was made to scan 100,000 directories and files. On the HTC G1 the process took 3953 seconds. This was much higher than expected. Through a debugger, it was discovered that the program invokes garbage collection very frequently due to memory pressure. Just using faster hardware—run on a QEMU-emulated single core virtual machine on a Dell Desktop with a 2.83GHz CPU and 4GB RAM—significant savings can be observed even while thrashing: the scenario only took 336 seconds (11.8x). Significant improvement can be achieved if modification of the heap and stack allocation can be done of the virtual machine to remove most garbage collection activity. A similarly powerful augmentation might execute a clone on an x86 port of the Android platform, removing the costs of emulating the ARM processor in the G1 Android Smartphone

- **Augmentation through multiplicity:** The last category to be considered is unique in that it uses multiple copies of the system image executed in different ways. This can help running data parallel applications (e.g. doing indexing for disjointed sets of images). This can also help the application to “see the future” by exhaustively exploring all possible next steps within some small horizon—as would be done for model checking—or to evaluate in maximum detail all possible choices for a decision before making that decision. Consider an energy conserving process scheduler that, in the absence of future knowledge, can only guarantee decisions that are close but not at the optimum. Instead, the whole system image could be replicated multiple times in the infrastructure, choosing all possible interleaving of processes during execution, and evaluating energy expenditure via some consumption model for the device, ultimately making the scheduling decision that results in minimum expenditure. In this category of augmentation, infrastructure cycles are lavished on essentially a Monte Carlo simulation of all possible outcomes of the scheduler's choices to make the optimal decision. All of us end up wasting a great amount of

energy (at the infrastructure) to save a little bit of energy on the mobile device. However, given the opportunity cost of being left with a dead battery during a critical time, this rather extravagant use of the infrastructure may have significant benefits.

Architecture

Conceptually, the system provides a way to boost a smartphone application by utilizing heterogeneous computing platforms through cloning and computation transformation. To do so, the system (semi) automatically transforms a single machine execution (e.g. smartphone computation) into a distributed execution (e.g. smartphone plus cloud computation) in which the resource intensive part of the execution is run in powerful clones. An additional benefit of cloning is that if the smartphone is lost or destroyed, the clone can be used as a backup. Figure 1 illustrates the high level system model of the approach.

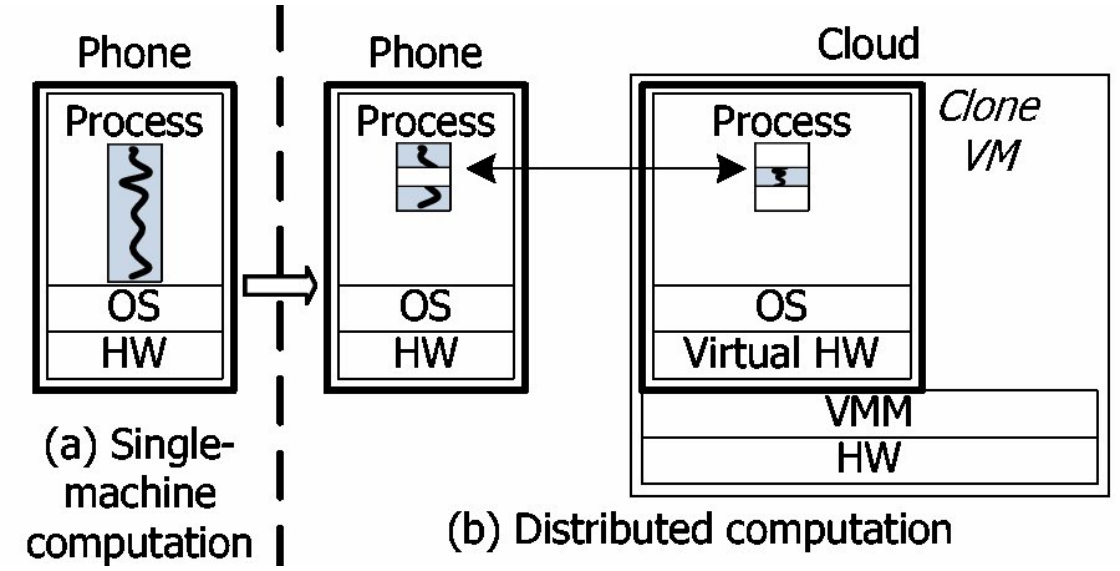


Figure 1: The system transforms a single machine execution (Smartphone computation) into a distributed execution (Smartphone and cloud computation) (semi) automatically.

Augmented Execution is performed in four steps:

1) Initially, a clone of the smartphone is created within the cloud (laptop, desktop, or server nodes)
2) The state of the primary (phone) and the clone is synchronized periodically or on demand
3) Application augmentations (whole applications or augmented pieces of applications) are executed in the clone, automatically or upon request
4) Results from clone execution are reintegrated into the smartphone state.

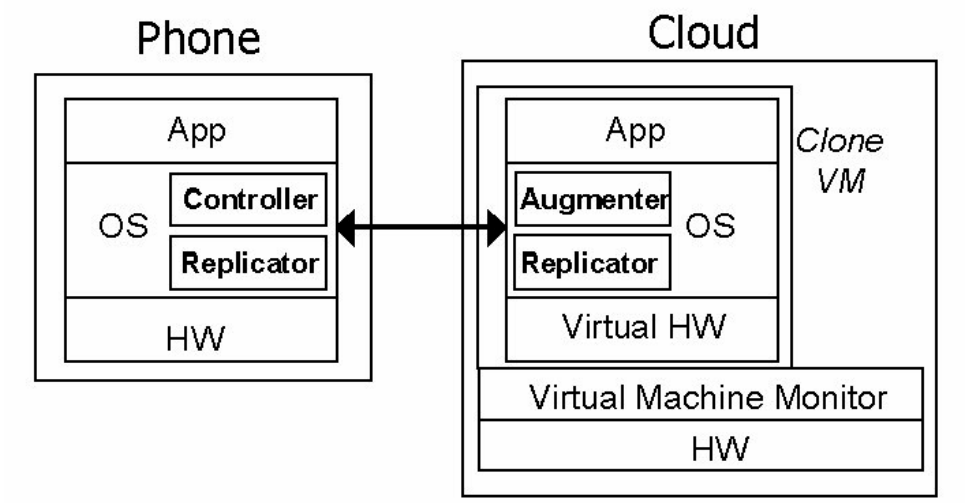


Figure 2: Clone execution architecture for smart phones.

Figure 2 shows a high level view of this system architecture. The system design is achieved by combining whole system replication through incremental check pointing, (semi) automatic partitioning and invocation of augmented execution, and coordination of computation between the primary (phone) and the clone. The system components are running inside the operating system. The Replicator is in charge of synchronizing the changes in phone software and state to the clone. The Controller running in the Smartphone invokes an augmented execution and merges its results back to the Smartphone. It interacts with the Replicator to synchronize states while coordinating the augmentation. The Augmenter running in the clone manages the local execution, and returns a result to the primary.

Once a computation block for remote execution is specified, the following steps are performed for the primary functionality outsourcing augmentation category. The steps for other augmentations are omitted due to space constraints.

1. Smartphone application process enters a sleep state.
2. The process transfers its state to the clone VM.
3. The VM allocates a new process state and overlays what it received from the phone with hardware description translation.
4. The clone executes from the beginning of the computation block until it reaches the end of the computation block.
5. The clone transfers its process state back to the phone.
6. The phone receives the process state and reintegrates it, and wakes up the sleeping process to continue its execution.

This description omits much detail. Other augmentation categories can be even less straightforward.

Snow Flock: Rapid Virtual Machine Cloning for Cloud Computing[3]

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.

Snow Flock is an implementation of the VM fork abstraction. To evaluate Snow Flock, the main focus is 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. Snow Flock provides sub-second VM cloning, scales to hundreds of workers, consumes few cloud I/O resources, and has negligible runtime overhead.

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 the 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. It is envisioned 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 webpage. 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 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 this 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, it provides a NAT2 layer to allow the clones to connect to certain external IP addresses. This 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.

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, 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 on 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.

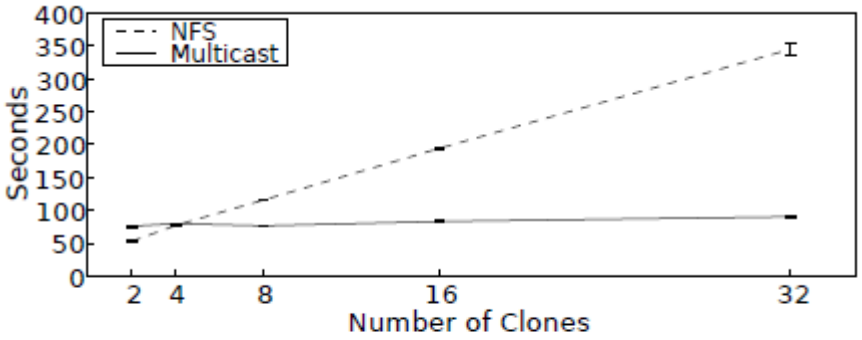


Figure 2: Latency for forking a 1GB VM by suspending and distributing the image over NFS and multicast.

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 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 behavior 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 tasteful 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 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.

Snow Flock Implementation

Snow Flock is our implementation of the VM fork primitive. Snow Flock is an open source project [Snow Flock] 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 hyper call interface. A privileged VM (domain0) has control over hardware devices and manages the state of all other domains.

Snow Flock is implemented as a combination of modifications to the Xen VMM and daemons that run in domain0. The Snow Flock daemons form a distributed system that controls the life cycle of VMs, by orchestrating their cloning and deal location. Snow Flock 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. Snow Flock currently supports allocation management with Platform EGO [Platform] and Sun Grid Engine [Gentzsch 2001]. Throughout this article, we use a simple internal resource manager which tracks memory and CPU allocations on each physical host.

Snow Flock's VM fork implementation is based on lazy state replication combined with avoidance heuristics to minimize state transfer. In addition, Snow Flock leverages multicast to propagate state in parallel and exploit the substantial temporal locality in memory accesses across forked VMs to provide prefetching. Snow Flock 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 sub-second 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 prefetching.

The next subsection describes the Snow Flock API. We then describe in detail each of the four Snow Flock 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 Snow Flock VM, namely the virtual disk and network isolation implementations.

API

Table 1 describes the Snow Flock API. VM fork in Snow Flock 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 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.

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 a **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**.

- **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 allocation in the ticket. Returns the clone ID, $0 \leq ID \leq m$.
- **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, immediately terminates all children in ticket and discards the ticket.

Table 1: The SnowFlock VM Fork API

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

While the Snow Flock API is simple and flexible, it nonetheless demands modification of existing code bases. A Snow Flock-friendly implementation of the widely used Message Passing Interface (MPI) library allows a vast corpus of unmodified parallel applications to use

Snow Flock'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 Map Reduce toolkit [Dean 2004].

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 which contains:

- metadata describing the VM and its virtual devices
- a few memory pages shared between the VM and the Xen hypervisor
- registers of the main VCPU
- Global Descriptor Tables (GDT) used by the x86 segmentation hardware for memory protection
- page tables of the VM

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 "canonical zed" 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 canonical zed.

The resulting descriptor is multicast to multiple physical hosts using the mcdist library described 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.

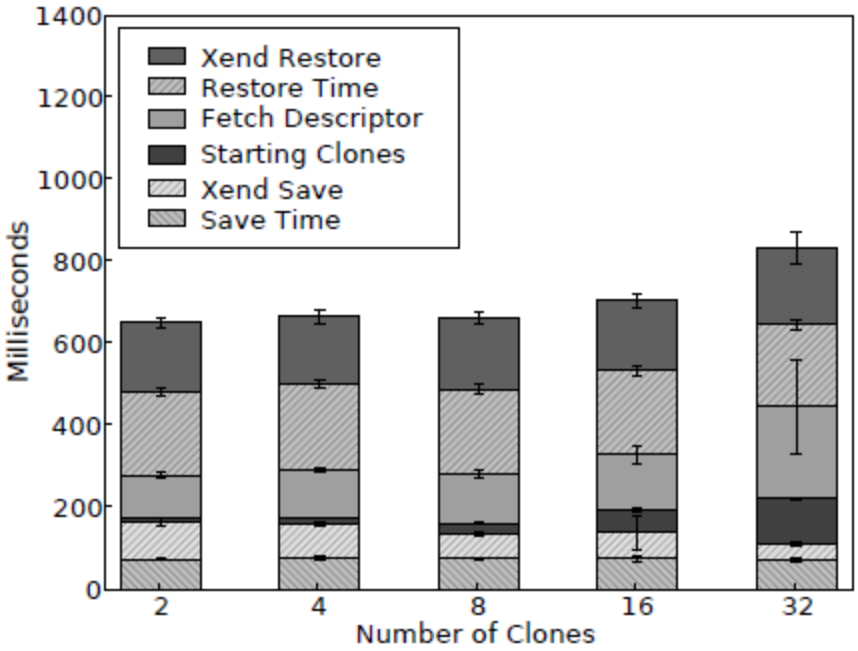


Figure 3: Fast Clone Creation. Legend order matches bar stacking from top to bottom.

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 hosts 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.

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 first instruction the VM tries to execute upon resume will be missing. Snow Flock'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.

We use Xen shadow page tables to allow the hypervisor to trap memory accesses to pages that have not yet been fetched. 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 micro benchmark 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 micro benchmark runs totaling ten thousand page fetches are displayed in Figure 4(a).

The overhead of page fetching is mcdist, 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.
>
> "Memtap Logic" is the time spent by the memtap internals, consisting mainly of mapping the faulting VM page.
>
> "Network" 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.

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 macro benchmark 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 sub section. Experiments with smaller memory footprints and different numbers of clones show similar results and are therefore not shown.

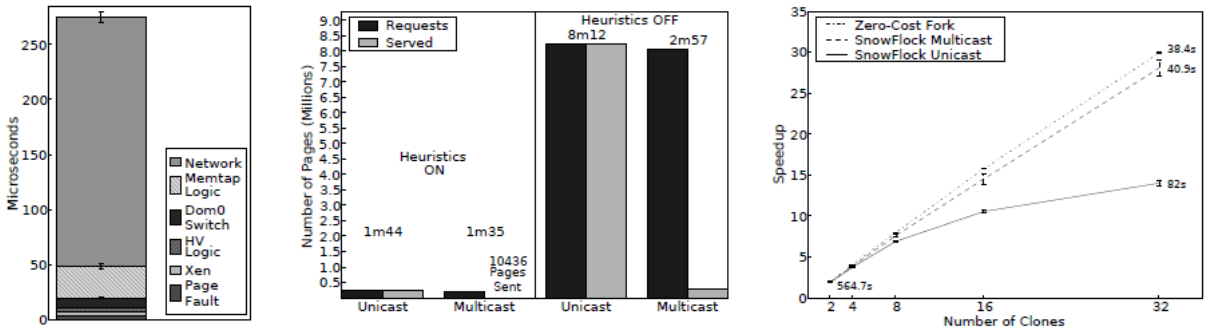
The avoidance heuristics result in substantial benefits, in terms of both runtime and data transfer. Nearly all of SHRiMP's memory footprint is allocated from scratch when the inputs are loaded. The absence of heuristics forces the VMs to request pages they do not really need, inflating the number of 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.

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 coexist in the cloud.

The modest server design is minimalistic, containing only switch programming and flow control logic. No atomicity or 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 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.



(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

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

To maximize throughput, 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 back its rate limit. 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 Snow Flock 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.

Virtual Disk

The virtual disks of Snow Flock VMs are implemented with a block tap [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 file systems 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.

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 Snow Flock 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 Snow Flock 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.

Application Evaluation

Our evaluation of Snow Flock 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 such as BLAST [NCBI] or ClustalW [EBI], and other Internet services such as 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 Power Edge 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 Snow Flock 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 Power Connect 5324 gigabit switches. All results reported are the means of five or more runs, and error bars depict standard deviations.

Applications

We tested Snow Flock with three typical applications from bioinformatics and three applications representative of the fields of graphics rendering, parallel compilation, and financial services. We devised workloads for these applications with run times 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 sav-ignyi, 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, and so forth. 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.

Results

We executed the above applications in Snow Flock, 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 Snow Flock compare to other methods for instantiating VMs?
- How close does Snow Flock come to achieving optimal application speedup?
- How scalable is Snow Flock? 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 Snow Flock provides in terms of efficient VM cloning and application performance. The table shows results for SHRiMP using 128 processors under three configurations: Snow Flock 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 Snow Flock 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, Snow Flock 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), Snow Flock 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.

Application Performance: Figure 5 compares Snow Flock 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 Snow Flock VMs in terms of kernel version, disk contents, RAM, and number of processors.

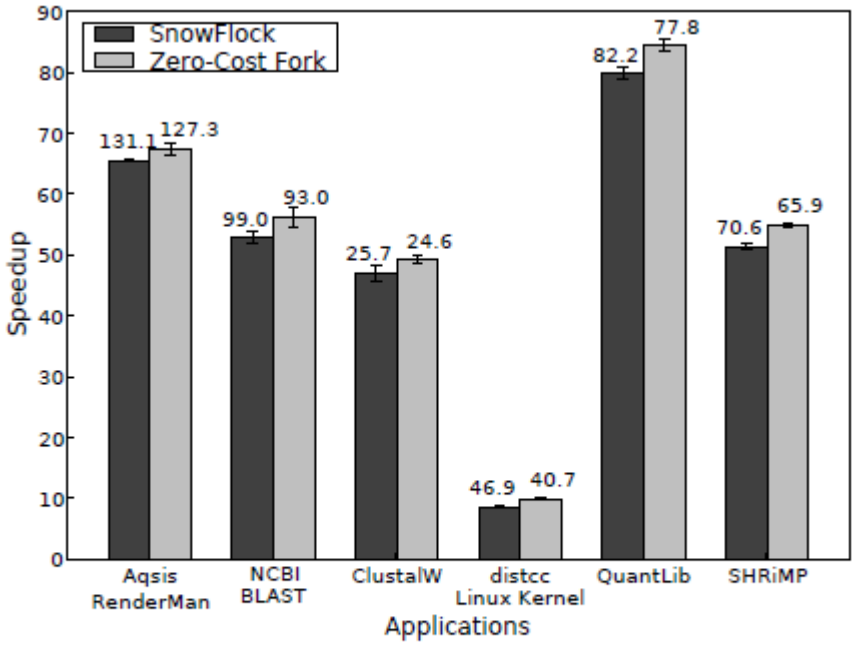


Figure 5: Application Benchmarks. Applications run with 128 threads: 32 VMs × 4 cores. Bars show speedup vs. a single thread zero-cost baseline. Labels show time to completion in seconds.

Snow Flock performs extremely well and succeeds in reducing execution time from hours to tens of seconds for all the benchmarks. Moreover, Snow Flock 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 is small. Crustal, in particular, yields the best results with less than two seconds of overhead for a 25 second task. This shows that tighter coupling of Snow Flock into application logic is beneficial.

Scale and Agility: We address Snow Flock'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 Snow Flock 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.

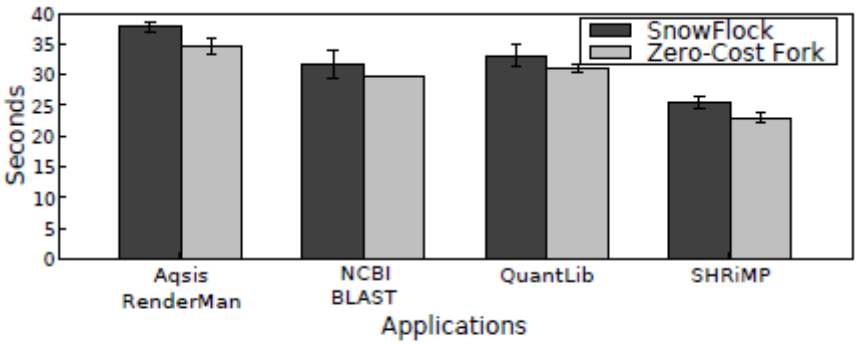


Figure 6: Concurrent Execution of Multiple Forking VMs. For each task we allocate 32 threads (32 VMs × 1 core), and cycle cloning, processing and joining repeatedly.

The results, shown in Figure 6, demonstrate that Snow Flock 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, Snow Flock 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.

Conclusion and Future Directions

Remote execution of resource intensive applications for resource poor hardware is a well known approach in mobile/pervasive computing. All remote execution work carefully designs and partitions applications between local and remote execution, and runs a simple visual, audio output routine at the mobile device and computation intensive jobs at a remote server. Rudenko, Flinn, and Satyanarayanan explored saving power via remote execution. Cyber foraging uses surrogates (untrusted and unmanaged public machines) opportunistically to improve the performance of mobile devices. For example, both data staging and Slingshot use surrogates. In particular, Slingshot creates a secondary replica of a home server at nearby surrogates. ISR provides an ability to suspend on one machine and resume on another machine by storing virtual machine images in a distributed storage system. Coign automatically partitions a distributed application composed of Microsoft COM components.

Clone Cloud uses nearby computers or data centers to speed up your smartphone applications, bringing the power of cloud computing to your fingertips. Currently, this involves exploring polymorphic execution that makes it possible to execute applications of resource-starved devices such as smartphones by opportunistically offloading computation to available cloud resources in nearby data centers. The idea is simple: clone the entire set of data and applications from the smartphone onto the cloud and selectively execute some operations on the clones, reintegrating the results back into the smartphone. One can have multiple clones for the same smartphone, clones pretending to be more powerful smartphones, and so on. Clone Cloud makes smartphones more powerful, secure, reliable, and power-efficient. At present, Clone Cloud exists as a prototype that runs on Google's Android mobile OS.

We can execute very expensive operations via cloud cloning such as object recognition, virus scanning, and data leak detection without requiring application designers to explicitly plan for cloning, without eating up the smartphone's battery power, and with significant performance improvement.

The augmentation and merging of Snow Flock may give rise to a new era of smartphones, tablets, and smart tabs which can make all things possible fitted right into your pocket. Security concerns can be overcome by RSA security and the virtualization platform can be provided by VMware merged with Snow Flock.

In this article, we introduced the primitive of VM fork and Snow Flock, 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 I/O 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, Snow Flock has broad applicability to other applications: flash crowd handling, execution of untrusted code components, instantaneous testing, and so forth.

Snow Flock 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.

Snow Flock is an active open source project [Snow Flock]. Our future work plans involve adapting Snow Flock to big data applications. We believe there is fertile research ground studying the interactions of VM fork with data parallel APIs such as Map Reduce [Dean 2004]. For example, Snow Flock'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.

Snow Flock'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, I wish to explore applying the Snow Flock 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, Snow Flock 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. Snow Flock 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, Snow Flock is of immediate relevance to users wishing to test and deploy parallel applications in the cloud.

Appendix

1. AUGMENTATION[1] berkeley.intel-research.net/bgchun/clonecloud-hotos09.pdf

2. ARCHITECTURE[2] www.usenix.org/event/hotos09/tech/full_papers/chun/chun.pdf

3. SNOW FLOCK[3]: Swift VM Cloning for Cloud Computing. http://sysweb.cs.toronto.edu/snowflock.

4. SAndroid dev phone 1. code.google.com/android/dev-devices.html.

5. Apple iPhone. www.apple.com/iphone.

6. Blackberry smart phones. na.blackberry.com/eng/.

7. Google desktop. desktop.google.com/.

8. Mcafee. www.mcafee.com.

9. Moblin. http://moblin.org.

10. Nokia N810 Internet tablet. www.nseries.com/index.html#1=products,n810.

11. Picasa. picasa.google.com/.

12. VMware ESX. www.vmware.com/products/vi/esx/.

13. Xen. www.xen.org.

14. R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In ACM SIGOPS European Workshop, 2002.

15. B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In SOSP, 2007.

16. B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Tiered Fault Tolerance for Long-Term Integrity. In FAST, 2009.

17. B. Cully et al. Remus: High availability via asynchronous virtual machine replication. In NSDI, 2008.

18. Diamedic. Diabetes Glucose Monitoring Logbook. www.martoon.com/diamedic/.

19. J. Diaz. iPhone Heart Monitor Tracks Your Heartbeat Unless You Are Dead. gizmodo.com/5056167/iphone-heart-monitor-tracks-your-heart-beat-unless-% you-are-dead, 2008.

20. J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In HotOS, 2001.

21. J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In SOSP, 1999.

22. J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data staging for untrusted surrogates. In USENIX FAST, 2003.

23. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In ASPLOS, 1996.

24. S. Garriss et al. Trustworthy and personalized computing on public kiosks. In MobiSys, 2008.

25. G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In OSDI, 1999.

26. J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In NDSS, 2005.

27. E. B. Nightingale, P. Chen, and J. Flinn. Speculative execution in a distributed file system. In SOSP, 2005.

28. E. B. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the sync. In OSDI, 2006.

29. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. Saving portable computer battery power through remote process execution. MCCR, 1998.

30. M. Satyanarayanan et al. Pervasive personal computing in an internet suspend/resume system. IEEE Internet Computing, 2007.

31. Y.-Y. Su and J. Flinn. Slingshot: Deploying stateful services in wireless hotspots. In MobiSys, 2005.

32. Young et al. Protium, an infrastructure for partitioned applications. In HotOS, 2001.