University of Toronto – Department of Computer Science
Technical Report CSRG-TR578
Impromptu Clusters for Near-Interactive Cloud-Based Services

H. Andrés Lagar-Cavilla, Joseph Whitney, Adin Scannell,
Stephen M. Rumble, Eyal de Lara, Michael Brudno, M. Satyanarayanan†
University of Toronto and Carnegie Mellon University†

Abstract

We introduce Impromptu Clusters (ICs), a new abstraction that makes it possible to leverage cloud-based clusters to execute short-lived parallel tasks, for example Internet services that use parallelism to deliver near-interactive responses. ICs are particularly relevant for resource-intensive web applications in areas such as bioinformatics, graphics rendering, computational finance, and search. In an IC, an application encapsulated inside a virtual machine (VM) is swiftly forked into multiple copies that execute on different physical hosts, and then disappear when the computation ends. SnowFlock, our IC prototype, offers near-interactive response times for many highly-parallelizable workloads, achieves subsecond parallel VM clone times, and has negligible runtime overhead.

1 Introduction

Over the past several years we have seen the development of two trends pertaining to parallelism and resource availability. One trend is the growth in web-based services that leverage parallel execution on compute clusters to deliver near-interactive performance (seconds to a few minutes) for resource-intensive applications. This trend is easy to observe in several fields, including bioinformatics. NCBI-BLAST, maintained by the National Center for Biotechnology Information (NCBI), is perhaps the most widely used bioinformatics tool. The BLAST service accepts DNA or protein sequences as queries, and allows biologists to quickly learn what other known biological sequences are similar. BLAST typically delivers search results within a few seconds through the use of cluster computing [23]. Many other examples of compute-intensive Internet services exist in diverse domains such as finance, graphics rendering, and search. Unfortunately, the hardware and staffing costs of creating and operating a large computational center form a high barrier to entry for new services of this genre.

The second trend is a growing interest in "cloud computing," suggesting the emergence of a utility model [2]. Here, the large fixed costs of operating the data center are borne by a third party service provider, such as Amazon or Yahoo. The creator of a new cloud-based service – a small organization or even an individual – only has to cover the marginal cost of actual resource usage, lowering the barrier to entry for a cloud-based service. Unfortunately, the programming interfaces available in today's cloud infrastructure are not a good match for parallel Internet services such as BLAST. Specifically, they lack support for rapidly creating an execution instance of an application with a very high degree of parallelism. Without such support, the ability to deliver near-interactive performance for such applications is lost.

We propose the *Impromptu Cluster (IC)*, an abstraction that lies at the nexus of these two trends and has the potential to speed their convergence. In an IC, an application encapsulated inside a virtual machine (VM) is swiftly forked into multiple copies that execute on different physical hosts, and then disappear when the computation ends. ICs simplify the development of parallel applications and reduces management burden by enabling the instantiation of new stateful computing elements: workers that need no setup time because they have a memory of the application state achieved up to the point of forking. This approach combines the benefits of cluster-based parallelism with those of running inside a VM. The latter include security, performance isolation, flexibility of running in a custom environment configured by the programmer, and the ability to migrate between execution sites. ICs make development of high performance applications accessible to a broader community of scientists and engineers by streamlining the execution of parallel applications on cloud-based clusters.

Our prototype implementation of the IC abstraction, called *SnowFlock*, provides swift parallel VM cloning that makes it possible for Internet applications to deliver near-interactive performance for resource-intensive highly-parallelizable tasks. SnowFlock makes use of four

key techniques: *VM descriptors* (condensed VM images that allow for sub-second suspension of a running VM and resumption of a number of replicas); a *memory-on-demand* subsystem that lazily populates the VM's memory image during runtime; a set of *avoidance heuristics* that minimize the amount of VM memory state to be fetched on demand; and a *multicast distribution* system for commodity Ethernet networking hardware that makes the overhead of instantiating multiple VMs similar to that of instantiating a single one.

Experiments conducted with applications from the domains of bioinformatics, quantitative finance, rendering and parallel compilation confirm that SnowFlock offers sub-second VM clone times with a runtime overhead of less than 7 % for most workloads. This allows IC applications to scale in an agile manner, and to offer near-interactive response times for many highly-parallelizable applications.

The remainder of this paper is organized as follows. Section 2 describes the Impromptu Cluster paradigm and motivates the need for fast parallel VM cloning. In Section 3 we present SnowFlock, our implementation of the IC abstraction. Section 4 describes the representative applications used to evaluate SnowFlock in Section 5. We discuss related work in Section 6, and conclude the paper in Section 7.

2 Impromptu Cluster

Today's cloud computing interfaces lack support for rapid instantiation of VMs needed by highly parallel applications. We propose the *Impromptu Cluster (IC)* abstraction to address this shortcoming. In IC, an application running inside a VM takes advantage of cluster computing by forking multiple copies of its VM, which then execute independently on different physical hosts. Use of an IC preserves the isolation and ease of software development associated with VMs, and greatly reduces the performance overhead of creating a collection of identical VMs on a number of physical machines.

A key property of an IC is the ephemeral nature of the replicas. Forked VMs are transient entities whose memory image and virtual disk are discarded once they exit. A VM fork in an IC is similar in nature to a traditional process fork in that the forked VMs have identical views of the system. However, each forked VM has its own independent copy of the operating system and virtual disk, and state updates are not propagated between VMs. We refer to the original VM that was initially forked to create the IC as the *master,* and to the other resulting VMs as *slaves.* Due to the transient nature of the slaves, 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 master VM, for example by message passing or via a distributed file system.

ICs can leverage the growing popularity of multiprocessor architectures by providing parallel execution as a combination of VM and process-level replication. VM replication is performed first, and process-level replication afterward. This allows for the creation of VMs which span multiple physical hosts, and processes which span the cores within each host.

We envision the creation of ICs as a highly dynamic task, driven by workload characteristics or resource availability. The IC abstraction is particularly apt for deploying Internet services that respond to requests by executing parallel algorithms. More broadly, impromptu parallelism lends itself well to applications with unpredictable request arrival and therefore varying resource demand.

2.1 Usage Model

Table 1 describes the IC API. VM forking has two stages. In the first stage, the application places a reservation for the desired number of clones with an `ic_request_ticket` call. Due to spikes in load or management policies, the cluster management system may allocate fewer nodes than the number requested. In this case the application has the option to re-balance the computation to account for the smaller node allocation. In the second stage, we fork the VM across the nodes provided by the cluster management system with the `ic_clone` call. When a slave VM finishes its part of the parallel computation, it executes an `ic_exit` operation that removes it from the IC. A parent VM can wait for its slaves to terminate by making an `ic_join` call, or force their termination with `ic_kill`. In section 4 we show API usage examples in the context of representative applications.

Hierarchical replication is optionally available to applications. In this mode, VMs are forked to span physical hosts, and process-level replication is used to span the processors within a host. Depending on the allocated resources, an IC might not obtain strictly symmetrical VMs, e.g. one four-processor VM and two dual-processor VMs for a reservation of size eight. The ticket object returned by an allocation request describes how the *m* process-level clones are distributed among the forked VMs. An application writer may wish to only leverage the VM cloning aspect of ICs and retain control over process-level replication. This can be achieved by simply disabling hierarchical replication in the ticket request.

When a child VM is forked, it is configured with a new IP address and is placed on the same virtual subnet as the VM from which it was created. In principle, IC VMs are only visible and can only communicate with their IC peers. This requirement can be relaxed to allow visibility of certain IP addresses on the physical cluster or on the Internet. This enables external access to the IC, for exam-

`ic_request_ticket (n, hierarchical)`	Requests an allocation for `n` clones. If `hierarchical` is true, process forking will be considered on top of VM forking. Returns a `ticket` describing an allocation for $m \leq n$ clones.
`ic_clone(ticket)`	Clones, using the allocation in the `ticket`. Returns the clone `ID`, $0 \leq ID \leq m$.
`ic_exit()`	For slaves ($1 \leq ID \leq m$), deallocates the slave.
`ic_join(ticket)`	For the master ($ID = 0$), blocks until all slaves in the `ticket` reach their `ic_exit` call. At that point all slaves are deallocated and the `ticket` is discarded.
`ic_kill(ticket)`	Master only, immediately deallocates all slaves in `ticket` and discards the `ticket`.

Table 1: The Impromptu Cluster API

ple through a web-based frontend, or to a dataset hosted by the physical infrastructure.

Finally, to enable domain-specific configuration of the IC, we provide a callback interface for registering command-line scripts or functions to be called as part of the VM forking process. For example, this feature can be used for simple file system sharing across an IC: after replication the master starts an NFS server and the rest of the VMs mount the NFS partition.

2.2 The Need for Agile VM Fork

We anticipate that VM fork will be a frequent operation. For instance, some servers will create impromptu clusters on many incoming requests. In this context, the instantiation of new VMs must be a rapid operation introducing low runtime overhead.

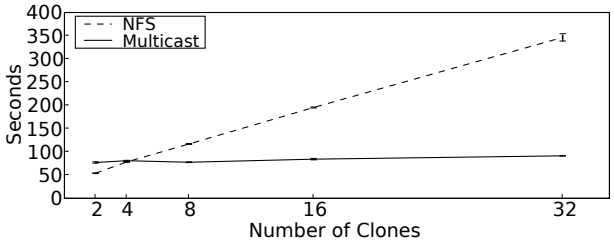


Figure 1: Latency for creating ICs by suspending a VM to disk and distributing the image over NFS and multicast. Experiments are conducted with Xen 3.0.3 and a VM with 1 GB of RAM.

Figure 1 shows that a straightforward approach (creating an IC by suspending a VM and resuming the saved image on a set of remote hosts) does not provide the necessary agility required for near-interactive response times. Overall, the latency in our testbed (see Section 5) for suspending and resuming a VM to and from local storage ranges from 9 to 22 seconds and from 6 to 38 seconds, respectively. The figure shows that latency grows linearly to the order of hundreds of seconds when NFS is used to distribute the VM image, as the source node quickly becomes a bottleneck. Using a multicast networking technique (described in Section 3.4) to distribute the VM image results in better scalability. However, multicast still averages more than a minute to replicate a VM and thus fails to provide the required swift setup time.

At first glance, an alternative to on-demand VM creation is to set up long-lived worker VMs that idle while they wait for user requests. However, because statistical multiplexing lies at the heart of cloud computing, the idle VMs are likely to be consolidated on a smaller number of physical hosts. As a result, when the worker VMs are needed to address a user request they have to be migrated first to free physical hosts – time-sharing a physical processor between VMs running a high-performance application is counter-productive. Moreover, because the memory images of the consolidated VMs diverge over time, VM migration has to be handled with individual point-to-point transmissions, creating hotspots that further impact application response time. Thus, for services that do not experience constant peak demand (the common case and the motivation for cloud computing), this approach will exhibit similar performance characteristics to the results in Figure 1.

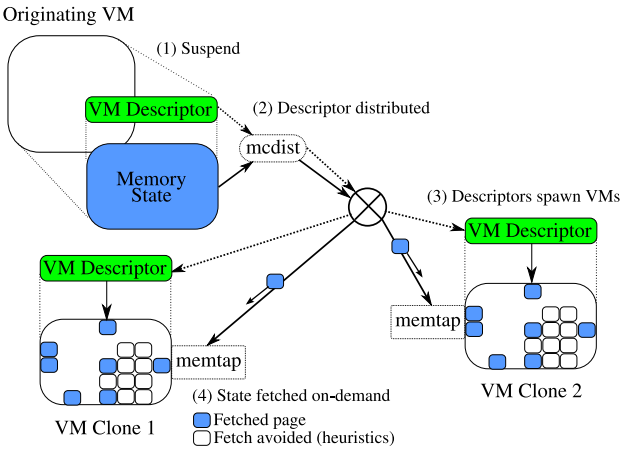
In summary, deployment of services with near-interactive response times in cloud computing settings requires an ability to swiftly instantiate new parallel workers that none of the existing techniques is capable of providing. In the next section we introduce our solution to this problem.

3 SnowFlock

SnowFlock is our prototype implementation of the IC abstraction. At its core lies a low-latency VM replication facility capable of instantiating new VM workers in sub-second time, with negligible runtime overhead. SnowFlock extends the Xen 3.0.3 Virtual Machine Monitor (VMM) [4]. 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 through a control toolstack. SnowFlock additionally consists of a client library running inside VMs, and a daemon running in each physical host's domain0.

API calls from Table 1 are available to workflow scripts or via C and python bindings. The client library marshals such calls and transfers them via a shared memory interface to the SnowFlock daemon. Daemons keep track of the current composition of each IC, including the location and status of each VM. The process of VM replication and deallocation is coordinated by the local daemon assigned

to the master VM; other local daemons interact with the extended Xen VMM in each host and spawn helper processes. SnowFlock defers VM allocation decisions to a resource manager plug-in. Cluster resource management, allocation and accounting fall outside of the scope of this work, and the purpose of our plug-in architecture is to leverage existing work such as Usher [16], Moab [18], or Sun Grid Engine [11]. The experiments we report in this paper use a simple first-come first-served resource manager.



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

Figure 2: Agile VM Replication Architecture

Our low-latency VM replication mechanism uses four techniques. First, the Master VM is temporarily suspended to produce a condensed VM image we call a *VM descriptor*. A VM descriptor is a small file that only contains basic VM metadata and the guest kernel memory management data structures, not the bulk of the VM's memory state. A VM descriptor can be created, distributed to other physical hosts, and used to spawn a new VM in less than a second. Our second technique is *memtap*, a memory-on-demand mechanism that lazily fetches the rest of the VM memory state as execution proceeds. Our third technique is the introduction of *avoidance heuristics* which substantially reduce the amount of memory that needs to be fetched on-demand by augmenting the guest kernel. Our fourth technique is *mcdist*, a multicast distribution system that uses the multicast features of commodity Ethernet network cards and switches. mcdist optimizes the on-demand propagation of VM state by leveraging the fact that the forked VMs take part in parallel computation, which results in substantial temporal locality in memory accesses across VMs. The interactions between these four mechanisms are depicted in Figure 2.

In this section, we describe in detail each of these four techniques. We then discuss the specifics of the virtual I/O devices of a SnowFlock VM, namely the virtual disk and network isolation implementations.

3.1 VM Descriptors

We observe that a VM suspend and resume cycle can be distilled to the minimal operations required to begin execution of a VM replica on a separate physical host. We thus modify the standard Xen suspend and resume process to yield *VM Descriptors*, condensed VM images that allow for swift clone spawning.

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 from execution. When the hypercall succeeds, a privileged process in domain0 maps the suspended VM memory to populate the descriptor. The descriptor contains metadata describing the VM and its virtual devices, a few memory pages shared between the VM and the Xen hypervisor and control tools, the registers of the remaining VCPU, and the contents of the guest kernel memory management data structures.

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

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

The resulting descriptor is multicast to multiple physical hosts using the mcdist library we describe in section 3.4, and used to spawn a VM replica on each host. The metadata is used to allocate a VM with the appropriate virtual devices and memory footprint. All state saved

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

We evaluate the VM descriptor mechanism in Section 5.2.1. In summary, the VM descriptors can be used to suspend a VM and resume a set of 32 replicas in less than a second, with an average descriptor size of roughly one MB for a VM with one GB of RAM.

3.2 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 VM replica's memory image with state fetched from the originating host, where an immutable copy of the VM's memory from the time of cloning is kept.

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

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

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

Certain paravirtual operations need the guest kernel to be aware of the memory-on-demand subsystem. When interacting with virtual I/O devices, the guest kernel hands *page grants* to domain0. A grant authorizes domain0 to behave as a DMA unit, by performing direct I/O on VM pages and bypassing the VM's page tables. To prevent domain0 from reading inconsistent memory contents, we simply touch the target pages before handing the grant, thus triggering the necessary fetches.

Our implementation of memory-on-demand is SMP-safe. The shared bitmap is accessed in a lock-free manner with atomic (`test_and_set`, etc) operations. When a shadow page table write triggers a memory fetch via memtap, we pause the offending VCPU and buffer the write of the shadow page table. Another VCPU using the same page table entry will fault on the still empty shadow 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 queued as depending on the very first fetch. When memtap notifies completion of the fetch, all pending shadow page table updates are applied, and all queued VCPUs are allowed to proceed with execution.

3.3 Avoidance Heuristics

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

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

The second heuristic addresses the case where a virtual I/O device writes to the guest memory. Consider block I/O: the target page is typically a kernel buffer that is being recycled and whose previous contents do not need to

be preserved. Again, the guest kernel can set the corresponding present bits and prevent the fetching of memory that will be immediately overwritten.

In section 5.2.3 we show the substantial improvement that these heuristics have on the performance of SnowFlock for representative applications.

3.4 Multicast Distribution

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

We use IP-multicast in order to send data to multiple hosts simultaneously. Multiple IP-multicast groups may exist simultaneously within a single local network and mcdist dynamically chooses a unique group in order to eliminate conflicts. Multicast clients subscribe by sending an IGMP protocol message with the multicast group to local routers and switches. The switches then relay each message to a multicast IP address to all switch ports that subscribed via IGMP. Off-the-shelf commercial Gigabit hardware supports IP-multicast.

In mcdist, the server is designed to be as minimal as possible, containing only membership management and flow control logic. No ordering guarantees are given by the server and requests are processed on a first-come first-served basis. Ensuring reliability thus falls to receivers, through a timeout based mechanism. In this section, we describe the changes to memtap necessary to support multicast distribution. We then proceed to describe two domain-specific enhancements, lockstep detection and the push mechanism.

3.4.1 Memtap Modifications

Our original implementation of memtap used standard TCP networking. A single memtap process would receive only the pages that it had asked for, in exactly the order requested, with only one outstanding request (the current one) per VCPU. Multicast distribution forces memtap to account for asynchronous receipt of data, since a memory page may arrive at any time by virtue of having been requested by another VM clone. This behaviour is exacerbated in push mode, as described in section 3.4.3.

On receipt of a page, the memtap daemon executes a hypercall that maps the target page of the VM in its address space. The cost of this hypercall can prove prohibitive if executed every time page contents arrive asynchronously. Furthermore, in an SMP environment, race conditions may arise when writing the contents of pages not explicitly requested: a VCPU may decide to use any of these pages without fetching them.

Consequently, in multicast mode, memtap batches all asynchronous responses until a threshold is hit, or a page that has been explicitly requested arrives and the mapping hypercall is needed regardless. To avoid races, all VCPUs of the VM are paused. Batched pages not currently mapped in the VM's physical space, or for which the corresponding bitmap entry is already set are discarded. A single hypercall is then invoked to map all remaining pages; we set our batching threshold to 1024 pages, since this is the largest number of pages mappable with a single context switch to the hypervisor. The page contents are copied, bits are set, and the VCPUs un-paused. The impact of this mechanism is evaluated in section 5.2.2.

3.4.2 Lockstep Detection

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

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

3.4.3 Push

Push mode is a simple enhancement to the server which sends data pro-actively. This is done under the assumption that the memory access patterns of a VM exhibit spatial locality. Our algorithm works as follows: when a request for a page comes to the server, in addition to providing the data for that request, the server adds a counter starting immediately above the requested page to a pool of counters. These counters are cycled through and incremented in turn, with the appropriate data sent for each one. As data, both requested and unrequested, is sent out by the server, a bit is set in a large bitmap. No data is sent twice due to the automated counters, although anything may be explicitly requested any number of times by a client, as in pull mode. Experimentally, we found that using any sufficiently large number of counters (e.g., greater than the number of clients) provides very similar performance.

Flow Control

Sending data at the highest possible rate quickly overwhelms clients, who face a significant cost for mapping pages and writing data. A flow control mechanism was designed for the server which limits and adjusts its sending rate over time. Both server and clients estimate their send and receive rate, respectively, using a weighted average of the number of bytes transmitted each millisecond. Clients provide explicit feedback about their current rate to the server in request messages. The server increases its rate limit linearly. Experimental evidence indicates that a 10 KB/s increment every 10 milliseconds is appropriate. When a loss is detected implicitly by a client request for data that has already been sent, the server scales its rate limit back to three quarters of its estimate of the *mean* client receive rate.

3.5 Virtual I/O Devices in an IC

Outside of the four techniques addressing fast and scalable VM replication, our IC implementation needs to provide a virtual disk for the cloned VMs, and must guarantee the necessary network isolation between ICs.

3.5.1 Virtual Disk

The virtual disks of VMs in an IC are implemented with a blocktap[29] driver. Multiple views of the virtual disk are supported by a hierarchy of copy-on-write slices located at the site where the master VM runs. Each clone operation adds a new COW slice, rendering the previous state of the disk immutable, and launches a disk server process that exports the view of the disk up to the point of cloning. Slaves 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 most cases the virtual disk is not heavily exercised in an IC. Most work done by slaves is processor intensive, resulting in little disk activity that does not hit kernel caches. Further, thanks to the heuristic previously mentioned, writes generally do not result in fetches. Our implementation largely exceeds the demands of many realistic tasks and did not cause any noticeable overhead for the experiments in section 5. For more complex scenarios, however, disk infrastructures such as Parallax [17] may be more suitable.

3.5.2 IC Network Isolation

In order to prevent interference or eavesdropping between ICs on the shared network, either malicious or accidental, we employ a mechanism to isolate the network for each IC 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 IC and clone identifiers. Simple filtering rules are used by all hosts to ensure that no packets delivered to a VM come from an IC other than its own. Conversely, when a packet is delivered to a SnowFlock VM, the destination MAC address is rewritten to be as expected by the VM, rendering the entire process transparent. Additionally, a small number of special rewriting rules are required for protocols with payloads containing MAC addresses, such as ARP. Despite this, the overhead imposed by filtering and rewriting is imperceptible and full compatibility at the IP level is maintained.

4 Applications

To evaluate the generality and performance of SnowFlock, we tested several usage scenarios involving 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 single-processor machine, but which can be substantially reduced to near interactive response times if provided enough resources. In general, 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. Results are dumped to temporary files which the clones send to the master before reaching an `ic_exit` call. Once the master successfully completes an `ic_join`, the results are collated. The exception to this technique is ClustalW in section 4.3, where we modify the application code directly.

4.1 NCBI BLAST

The NCBI implementation of BLAST[1], the Basic Local Alignment and Search Tool, is perhaps the most popular computational tool used by biologists. BLAST searches a database of *biological sequences* – strings of characters representing DNA or proteins – to find sequences similar to a query. Similarity between two sequences is measured by an *alignment* metric, that is typically similar to edit distance. BLAST is demanding of both computational and I/O resources; gigabytes of sequence data must be read and compared with each query sequence, and parallelization can be achieved by dividing either the queries, the sequence database, or both. We experimented with a BLAST search using 1200 short protein fragments from the sea squirt *Ciona savignyi* to query a 1.5GB por-

tion of NCBI's non-redundant protein database, a consolidated reference of protein sequences from many organisms. VM clones access the database, which is a set of plain text files, via an NFS share. Database access is parallelized across VMs, each reading a different segment, while query processing is parallelized across process-level clones within each VM.

4.2 SHRiMP

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

4.3 ClustalW

ClustalW [14] 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 [9]. ClustalW uses progressive alignment – a greedy heuristic that significantly speeds up the multiple alignment computation, but requires precomputation of pairwise comparisons between all pairs of sequences – to build a guide tree. The pairwise comparison is computationally intensive and embarrassingly parallel, since each pair of sequences can be aligned independently.

We have modified the standard ClustalW program to allow parallelization with SnowFlock: Figure 3 shows the integration of API calls into the ClustalW code. After cloning, each slave 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 master, before joining the IC. Replacing IC cloning with process forking yields an equivalent parallel program confined to executing within a single machine. Using this implementation we conducted experiments performing guide-tree generation by pairwise alignment of 200 synthetic protein sequences of 1000 amino acids (characters) each.

4.4 QuantLib

QuantLib [22] is an open source development toolkit widely used in quantitative finance. It provides a vast set of models for stock trading, equity option pricing, risk analysis, etc. Quantitative finance programs are typically single-program-multiple-data (SPMD): a typical task using QuantLib runs a model over a large array of parameters (e.g. stock prices,) and is thus easily parallelizable by splitting the input. In our experiments we ran a set of Monte Carlo, binomial and Black-Scholes variant models to assess the risk of a set of equity options. Given a fixed maturity date, we processed 1024 equity options varying the initial and striking prices, and the volatility. The result is the set of probabilities yielded by each model to obtain the desired striking price for each option.

```
sequences = ReadSequences(InputFile)
ticket = ic_request_ticket(n, hierarchical=true)
m = ticket.allocation
ID = ic_clone(ticket)
for i in sequences:
   for j in sequences[i+1:]:
      if ((i*len(sequences)+j) % m == ID):
         PairwiseAlignment(i, j)
if (ID > 0):
   RelayScores()
   ic_exit()
else:
   PairsMatrix = CollatePairwiseResults()
   ic_join(ticket)
BuildGuideTree(PairsMatrix, OutputFile)
```

Figure 3: ClustalW Pseudo-Code Using the IC API

4.5 Aqsis – Renderman

Aqsis [3] is an open source implementation of Pixar's RenderMan interface [20], an industry standard widely used in films and television visual effects since 1989. This renderer accepts scene descriptions produced by a modeler and specified in the RenderMan Interface Bitstream (RIB) language. Rendering also belongs to the SPMD class of applications, and is thus easy to parallelize: multiple instances can each perform the same task on different frames of an animation. For our experiments we fed Aqsis a sample RIB script from the book "Advanced RenderMan: Creating CGI for Motion Pictures".

4.6 Distcc

Distcc [7] is software which distributes builds of C/C++ programs over the network for parallel compilation. It operates by sending preprocessed code directly to a compiling process on each host and retrieves object file results back to the invoking host. Because the preprocessed code includes all relevant headers, it is not necessary for each compiling host to access header files or libraries; all they need is the same version of the compiler. Distcc is different from our previous benchmark in that it is not embarrassingly parallel: actions are tightly coordinated by a master host farming out preprocessed files for compilation by slaves. In our experiments we compile the Linux kernel (version 2.6.16.29) from `kernel.org`.

5 Evaluation

We first examine the high-level performance of SnowFlock with experiments using representative application benchmarks described in section 4. In section 5.2, we turn our attention to a detailed micro evaluation of the different aspects that contribute to SnowFlock's performance and overhead.

All of our experiments were carried out on a cluster of 32 Dell PowerEdge 1950 blade servers. Each machine had 4 GB of RAM, 4 Intel Xeon 3.2 GHz cores, and a Broadcom NetXtreme II BCM5708 gigabit network adaptor. 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. All machines were connected to two daisy-chained Dell PowerConnect 5324 gigabit switches. Unless otherwise noted, all results reported in the following subsections are the means of five or more runs, error bars depict standard deviations, and all VMs were configured with 1124 MB of RAM.

In several experiments we compare SnowFlock's performance against an "ideal" baseline. The ideal results are obtained with VMs previously allocated with all necessary resources and in an idle state, ready to process the jobs alloted to them. These VMs are vanilla Xen 3.0.3 domains, with no cloning or state-fetching overhead, similarly configured in terms of kernel version, disk contents, RAM, and number of processors. We note that the ideal results are not representative of cloud computing environments, in which aggressive consolidation of VMs is the norm.

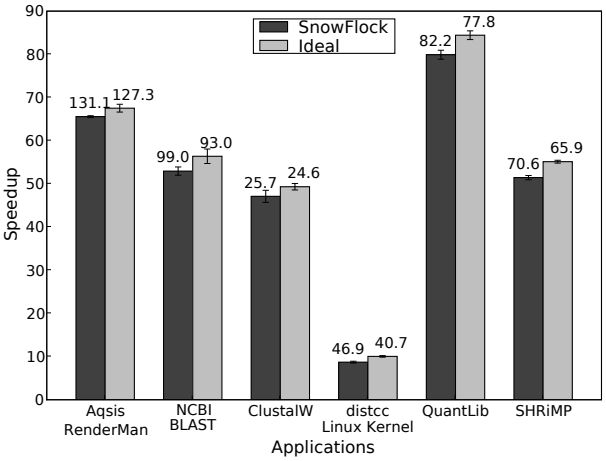
5.1 Macro Evaluation

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

- What is the performance of SnowFlock for the representative applications described in section 4?
- How capable is SnowFlock of delivering near-interactive response times when supporting multiple concurrent ICs, and how does it react to stressful conditions that include repeated cloning and adverse VM allocation patterns?

5.1.1 Application Results

We test the performance of SnowFlock with the applications described in section 4 and the following features enabled: memory on demand, multicast without push, avoidance heuristics, and hierarchical cloning. For each application we spawn 128 threads of execution (32 4-core SMP VMs on 32 physical hosts) in order to fully utilize our testbed. SnowFlock is tested against an ideal with 128 threads to measure overhead, and against an ideal with a single thread in order to measure speedup.



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

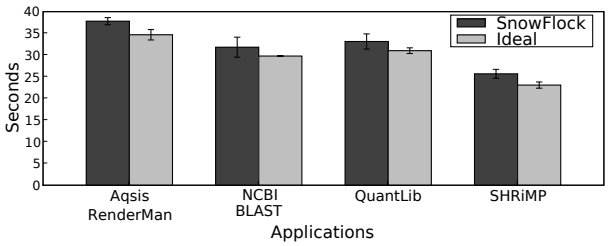
Figure 4: Application Benchmarks

Figure 4 shows the results. We obtain speedups very close to the ideal, and total time to completion no more than five seconds (or 7%) greater than the ideal. The overheads of VM replication and on-demand state fetching are small. These results demonstrate that the execution time of a parallelizable task can be reduced, given enough resources, to a near-interactive range. We note three aspects of application execution. First, ClustalW yields the best results, showing that tighter coupling of SnowFlock into application logic is beneficial. Second, while most of the applications are embarrassingly parallel, the achievable speedups are below the maximum: some slaves may finish ahead of others, so that "time to completion" effectively measures the time required for the slowest VM to complete. Third, distcc's tight synchronization results in underutilized slaves and low speedup, although distcc still delivers a near-interactive time to completion.

5.1.2 Scale and Agility

In this section we address SnowFlock's capability to support multiple concurrent ICs. We launch four VMs that each simultaneously spawn an IC of 32 uniprocessor VMs. To further stress the system, after completing a parallel task each IC joins and deallocates its slaves and immediately launches another parallel task, repeating this cycle five times. Each IC runs a different application. We selected the four applications that exhibited the highest degree of parallelism (and slave occupancy): SHRiMP, BLAST, QuantLib, and Aqsis. To even 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 for four ICs. The

"ideal" results were obtained with an identical distribution of VMs; since there is no state-fetching performed in the ideal case, the actual allocation of VMs does not affect those results.



For each task we cycle cloning, processing and joining repeatedly.

Figure 5: Concurrent Execution of Multiple ICs

The results, shown in Figure 5, demonstrate that SnowFlock is capable of withstanding the increased demands of multiple concurrent ICs. As we will show in section 5.2.3, this is mainly due to the small overall number of memory pages sent by the multicast server, when pushing is disabled and heuristics are enabled. The introduction of multiple ICs 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.

5.2 Micro Evaluation

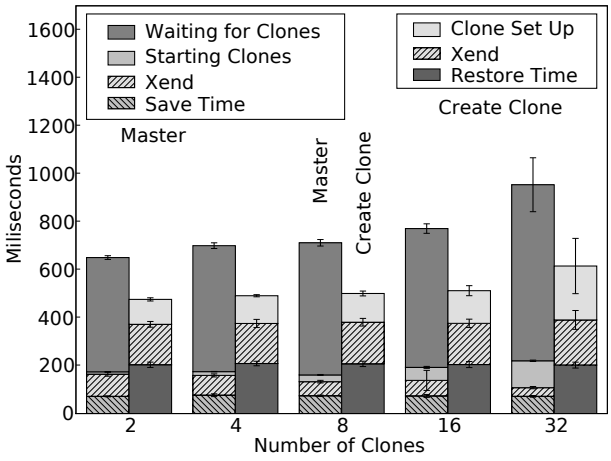
To better understand the behaviour of SnowFlock, and to isolate the factors which cause the small overhead seen in the previous section, we performed experiments to answer several performance questions:

- How fast does SnowFlock spawn an IC? How scalable is the cloning operation?
- What are the sources of overhead when SnowFlock fetches memory on demand to a VM?
- How is the performance of SnowFlock sensitive to the use of avoidance heuristics and the choice of networking strategy?

5.2.1 Fast VM Replica Spawning

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

Recall that in section 3, the process used to replicate a VM was introduced. For the purposes of evaluating the



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

Figure 6: Time To Create Clones

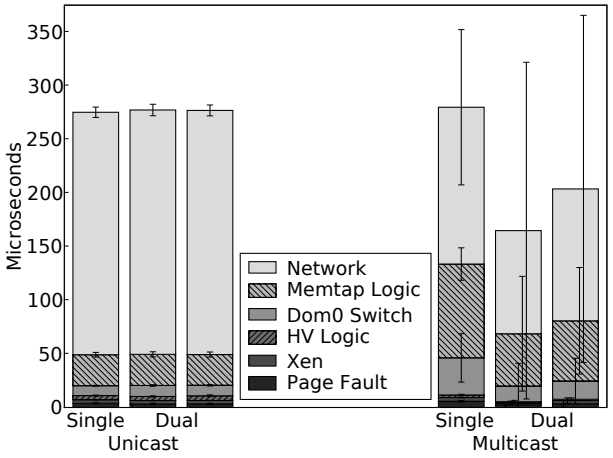
time required for VM cloning, we decompose this process into the following steps corresponding to bars in Figure 6: 1) Suspending the running VM and generating a VM descriptor. These are the "Save Time" and "Xend" components in the "Master" bar; "Xend" stands for unmodified Xen code we leverage. 2) Contacting all target physical hosts to trigger VM instantiation ("Starting Clones"). 3) Each target host pulls the VM descriptor via multicast ("Clone Set up" in the "Create Clone" bar). 4) Spawning each clone VM from the descriptor ("Restore Time" and "Xend"). 5) Set up, including network isolation, for each clone. ("Clone Set Up"). 6) Waiting to receive notification from all target hosts ("Waiting for Clones", which roughly corresponds to the total size of the corresponding "Create Clone" bar).

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

5.2.2 Memory On Demand

To understand the overhead involved in our memory-on-demand subsystem, we devised a microbenchmark in which a VM allocates and fills in a number of memory pages, invokes SnowFlock to have itself replicated and then touches each page in the allocated set. We instrumented the microbenchmark, the Xen hypervisor and memtap to timestamp events during the fetching of each page. The results for multiple microbenchmark runs to-

talling ten thousand page fetches are displayed in Figure 7. The "Single" columns depict the result when a single VM fetches state. The "Dual" columns show the results when two VM clones concurrently fetch state.



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

Figure 7: Page Fault Time

We split a page fetch operation into six components. "Page Fault" indicates the hardware overhead of using the shadow page tables to detect first access to a page after VM resume. "Xen" is the cost of executing the Xen hypervisor shadow page table logic. "HV Logic" is the time consumed by our logic within the hypervisor (bitmap checking and SMP safety.) "Dom0 Switch" is the time spent while context switching to the domain0 memtap process, while "Memtap Logic" is the time spent by the memtap internals, consisting mainly of mapping the faulting VM page. Finally, "Network" depicts the software and hardware overheads of remotely fetching the page contents over gigabit Ethernet.

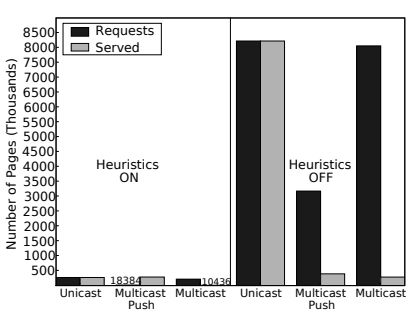
The overhead of page fetching is modest, averaging 275 μs with unicast (standard TCP). Our implementation is frugal, and the bulk of the time is spent in the networking stack. With multicast, substantially higher variances are observed in three components. As explained in section 3.4.1, memtap fully pauses the VM and batches multiple page mappings in a single hypercall with multicast, making the average operation more costly. Also, mcdist's logic and flow control are not as optimized as TCP's, and they run in user space. The need to perform several system calls results in a high scheduling variability. A final contributor to multicast's variability is the effectively bimodal behaviour caused by implicit prefetching. Sometimes, the page the VM needs may already be present, in which case the logic defaults to a number of simple checks. This is far more evident in the dual case – requests by one VM result in prefetching for the other, and explains the high variance and lower overall "Network" averages for the multicast case.

5.2.3 Sensitivity Analysis

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

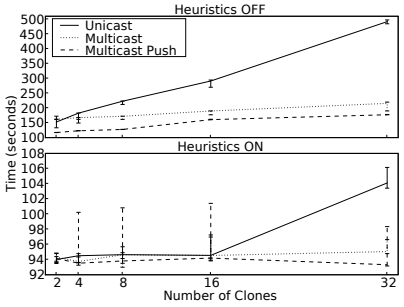
Figure 8 (a) illustrates the memory-on-demand activity for a memory footprint of 1 GB and 32 VM clones. Consistent results were observed for all experiments with smaller numbers of clones, and with the 512 MB footprint. The immediate observation is the substantially beneficial effect of the avoidance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

Three aspects of multicast execution are reflected in Figure 8 (a). First, even under the extreme pressure of disabled heuristics, the number of pages served is reduced dramatically. This enables a far more graceful scaling than with unicast, as seen in the upper portion of Figure 8 (b). Second, 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 three of the four multicast experiments. Third, push mode increases the chances of successful prefetching and decreases the overall number of requests from all VMs, at the cost of sending more pages. The taller error bars (up to 8 seconds) of the push experiments, however, reveal the instability caused by the aggressive distribution. Constantly receiving pages keeps memtap busy and not always capable of responding quickly to a page request from the VM, hurting the runtime slightly and causing



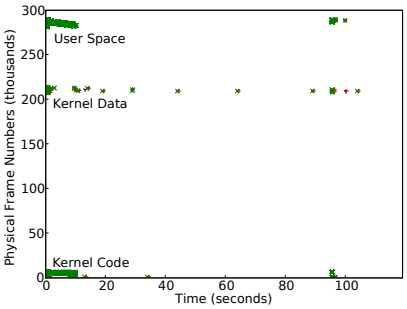
Comparison of aggregate page requests from 32 clones vs. number of pages sent by the memory server.

(a) Pages Requested vs. Served



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

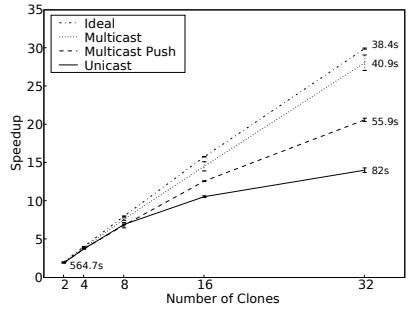
(b) Time To Completion



Page requests over time for three randomly selected VMs out of 32 clones. Multicast with heuristics enabled.

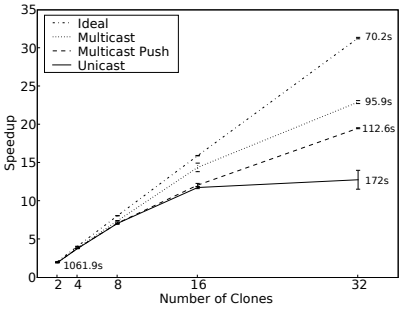
(c) Pages Requested vs. Time

Figure 8: Sensitivity Analysis with SHRiMP – 1 GB Memory Footprint



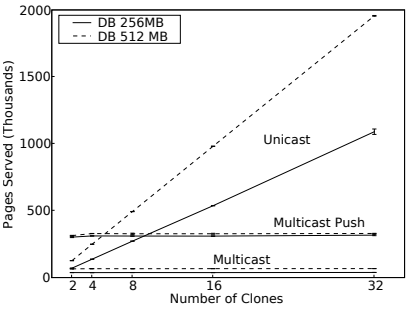
Speedup against a single thread, with labels showing absolute times.

(a) Speedup – 256 MB DB



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

(b) Speedup – 512 MB DB



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

(c) Memory Pages Served

Figure 9: Sensitivity Analysis with NCBI BLAST

VMs to lag behind.

Figure 8 (c) shows the requests of three randomly selected VMs as time progresses during the execution of SHRiMP. Heuristics are enabled, the footprint is 1 GB, and there are 32 VM clones using multicast. Page requests cluster around three areas: kernel code, kernel data, and previously allocated user space code and data. Immediately upon resume, kernel code starts executing, causing the majority of the faults. Faults for pages mapped in user-space are triggered by other processes in the system waking up, and by SHRiMP forcing the eviction of pages to fulfill its memory needs. Some pages selected for this purpose cannot be simply tossed away, triggering fetches of their previous contents. Due to the avoidance heuristics, SHRiMP itself does not directly cause any fetches, and takes over the middle band of physical addresses for the duration of the experiment; fetches are only performed rarely when kernel threads need data.

5.2.4 Heuristic-adverse Experiment

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

Figure 9 (a) plots the speedups achieved for a DB of 256 MB by cloning a uniprocessor VM to n replicas. Speedups are against an ideal with a single VM, and the labels indicate the runtime for 32 VMs. We see an almost linear speedup for multicast, closely tracking the speedup exhibited with ideal execution, while unicast ceases to scale after 16 clones. Multicast push is better than unicast but unable to perform as well as multicast, due to memtap and network congestion. With a larger database (Figure 9 (b)) even multicast starts to suffer a noticeable overhead, showing the limits of the current implementation.

Figure 9 (c) shows the number of pages fetched vs. the number of clones for the three networking substrates. The information is consistent with that in Figure 8 (a). The linear scaling of network utilization by unicast leads di-

rectly to poor performance. Multicast is not only the better performing, but also the one with the least network utilization, allowing for better co-existence with other SnowFlock ICs. These observations, coupled with the instabilities of push mode shown in the previous section, led us to choose multicast with no push as SnowFlock's default behaviour for the macro-evaluation in section 5.1.

6 Related Work

To the best of our knowledge, we are the first group to address the problem of low-latency replication of VMs for cloud-based services that leverage parallelism to deliver near-interactive response times. A number of projects have worked to develop efficient multiplexing of many VMs on a single machine. The Potemkin project [27] 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. Denali [30] dynamically multiplexes VMs that execute user-provided code in a webserver, with a focus on security and isolation.

Emulab [12] uses virtualization to efficiently support large-scale network emulation experiments in their testbed. The "experiment" notion in Emulab bears a resemblance to SnowFlock's IC: it is a set of VMs connected by a virtual network. Experiments are long-lived and statically sized: the number of nodes does not change during the experiment. Instantiation of all nodes takes tens to hundreds of seconds. Emulab uses Frisbee [13] as a multicast distribution tool to apply disk images to nodes during experiment setup. Frisbee and mcdist differ in their domain-specific aspects: for instance, Frisbee uses filesystem-specific compression techniques which do not apply to memory state; conversely the lockstep problem in memory sending addressed by mcdist does not apply to Frisbee's disk distribution.

One objective of SnowFlock is to complement the capabilities of a shared computing platform. The Amazon Elastic Compute Cloud [2] (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 [26]: consolidation via memory sharing [28] or ballooning, resuming from disk, live migration [6], etc. Amazon's EC2 claims to instantiate multiple VMs in "minutes" – insufficient performance for the agility objectives of SnowFlock.

The term "virtual cluster" has been employed by many projects [8, 10, 24] with semantics differing from SnowFlock's IC. The focus has been almost exclusively on the resource provisioning and management aspects. Chase et al. [5] present dynamic virtual clusters in which nodes are wiped clean and software installed from scratch to add them to a long-lived on-going computation. Nishimura's virtual clusters [19] are statically sized, and created from a single image also built from scratch via software installation. Usher [16] is a modular manager of clusters of VMs; Usher's virtual clusters can be dynamically resized via live migration or VM suspend and resume. Usher could complement SnowFlock's architecture by acting as the resource manager and implementing policies for VM allocations. Another alternative is Platform EGO [21].

We view the use of high-speed interconnects, if available, as a viable alternative to multicasting. Huang et al. [15] demonstrate very fast point-to-point VM migration times with RDMA on Infiniband. We note that latency increases linearly, however, when a single server pushes a 256 MB VM to multiple receivers, which is a crucial operation to optimize for use in SnowFlock.

7 Conclusion and Future Directions

In this work, we introduced *Impromptu Clusters* (IC), a new abstraction that streamlines the execution of parallel applications on cloud-based clusters. ICs preserve the isolation and ease of development associated with executing inside a VM, while allowing applications to take advantage of cluster computing by forking multiple copies of their VM, which then execute independently on different physical hosts.

SnowFlock, our IC prototype, can clone a VM over a large number of physical hosts in less than a second. SnowFlock's fast parallel cloning makes it possible for web-based services to leverage cloud-based clusters to deliver near-interactive response times for resource-intensive, highly parallel applications.

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. The rest of the VM's memory image can be fetched efficiently on-demand. Moreover, simple modifications to the guest kernel significantly reduce network traffic by eliminating the transfer of pages that will be overwritten. Second, the parallel nature of the application being forked results in a locality of memory accesses that makes it beneficial to distribute the VM image using multicasting. This allows for the instantiation of a large number of VMs at a cost similar to that of forking a single copy.

We observe that the IC paradigm is not only useful to utility computing operations but can also be used by organizations that prefer to share internal compute resources instead. For example, an animation studio can use SnowFlock to share a large cluster among several animation teams, or a financial company can share a large cluster between their financial derivatives groups.

In this paper we have remained within the scope of embarrassingly parallel applications, which need little explicit synchronization. A natural progression is to consider problems that demand tighter sharing, particularly of areas of memory. Similarly, developing a SnowFlock-friendly version of MPI, in which the task of spawning an IC is transparently triggered by "mpirun" invocation, holds the promise of enabling transparent and binary compatibility for a large number of existing applications. Our initial experience with MPI versions of BLAST and ClustalW has thus far been positive.

In closing, by combining the benefits of parallelization with the strengths of VM technology, ICs will make high performance applications accessible to a much broader community of scientists and engineers. From a broader perspective, ICs may prove to be a catalyst for scientific discovery as it will speed up the adoption of new applications that will enable scientists to conduct experiments that are currently not possible.

Acknowledgments

This research was supported by the National Science Foundation (NSF) under grant number CNS-0509004, the National Science and Engineering Research Council (NSERC) of Canada under a strategic grant and a Canada Graduate Scholarship, by the Canadian Foundation for Innovation (CFI), and the Ontario Innovation Trust (OIT) under grant number 7739, and by a research grant from Paltform Computing Corp. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, NSERC, CFI, OIT, Platform Computing Corp, Carnegie Mellon University, or the University of Toronto.

We thank Young Yoon for his help with QuantLib, Ryan Lilien for his early involvement in this work, and Lionel Litty for comments on earlier drafts.

References

[1] ALTSCHUL, S. F., MADDEN, T. L., SCHAFFER, A. A., ZHANG, J., ZHANG, Z., MILLER, W., AND LIPMAN, D. J. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res. 25* (1997), 3389–3402.

[2] AMAZON.COM. Amazon Elastic Compute Cloud (Amazon EC2). `http://www.amazon.com/gp/browse.html?node=201590011`.

[3] Aqsis. `http://aqsis.org/`.

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proc. of the* 17th *Symposium on Operating Systems Principles (SOSP)* (Bolton Landing, NY, Oct. 2003).

[5] CHASE, J. S., IRWIN, D. E., GRIT, L. E., MOORE, J. D., AND SPRENKLE, S. E. Dynamic Virtual Clusters in a Grid Site Manager. In *Proc. 12*th *IEEE International Symposium on High Performance Distributed Computing (HPDC)* (Washington, DC, 2003).

[6] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proc. 2*nd *Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005).

[7] distcc: a Fast, Free Distributed C/C++ Compiler. `http://distcc.samba.org/`.

[8] EMENEKER, W., AND STANZIONE, D. Dynamic Virtual Clustering. In *Proc. IEEE International Conference on Cluster Computing (Cluster)* (Austin, TX, Sept. 2007).

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

[10] FOSTER, I., FREEMAN, T., KEAHEY, K., SCHEFTNER, D., SOTOMAYOR, B., AND ZHANG, X. Virtual Clusters for Grid Communities. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid)* (Singapore, May 2006).

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

[12] HIBLER, M., RICCI, R., STOLLER, L., DUERIG, J., GURUPRASAD, S., STACK, T., WEBB, K., AND LEPREAU, J. Feedback-directed Virtualization Techniques for Scalable Network Experimentation. Tech. Rep. FTN-2004-02, University of Utah, May 2004.

[13] HIBLER, M., STOLLER, L., LEPREAU, J., RICCI, R., AND BARB, C. Fast, Scalable Disk Imaging with Frisbee. In *Proc. of the USENIX 2003 Annual Technical Conference* (San Antonio, TX, June 2003).

[14] HIGGINS, D., THOMPSON, J., AND GIBSON, T. Clustal w: improving the sensitivity of progressivemultiple sequence alignment through sequence weighting,position-specific gap penalties and weight matrix choice. *Nucleic Acids Res. 22* (1994), 4673–4680.

[15] HUANG, W., GAO, Q., LIU, J., AND PANDA, D. K. High Performance Virtual Machine Migration with RDMA over Modern Interconnects. In *Proc. IEEE International Conference on Cluster Computing (Cluster)* (Austin, TX, Sept. 2007).

[16] MCNETT, M., GUPTA, D., VAHDAT, A., AND VOELKER, G. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In *Proc. 21*st *Large Installation System Adminstration Conference (LISA)* (Dallas, TX, Nov. 2007).

[17] MEYER, D., AGGARWAL, G., CULLY, B., LEFEBVRE, G., HUTCHINSON, N., FEELEY, M., AND WARFIELD, A. Parallax: Virtual Disks for Virtual Machines. In *Proc. Eurosys 2008* (Glasgow, Scotland, Apr. 2008).

[18] MOAB. Moab Cluster Suite, Cluster Resources Inc., 2008. `http://www.clusterresources.com/pages/products/moab-cluster-suite.php`.

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

[20] Pixar's RenderMan. `https://renderman.pixar.com/`.

[21] PLATFORM COMPUTING, INC. Technical Whitepaper: Integrating Enterprise Infrastructures with Platform Enterprise Grid Orchestrator (EGO), 2006.

[22] QuantLib: a Free/Open-source Library for Quantitative Finance. `http://quantlib.org/index.shtml`.

[23] RPS-BLAST. `http://www.ncbi.nlm.nih.gov/Structure/cdd/cdd_help.shtml`.

[24] RUTH, P., MCGACHEY, P., JIANG, J., AND XU, D. VioCluster: Virtualization for Dynamic Computational Domains. In *Proc. IEEE International Conference on Cluster Computing (Cluster)* (Boston, MA, Sept. 2005).

[25] SHRiMP - SHort Read Mapping Package. `http://compbio.cs.toronto.edu/shrimp/`.

[26] STEINDER, M., WHALLEY, I., CARRERA, D., GAWEDA, I., AND CHESS, D. Server Virtualization in Autonomic Management

of Heterogeneous Workloads. In *Proc. 10th Integrated Network Management (IM) conference* (Munich, Germany, 2007).

[27] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A., VOELKER, G., AND SAVAGE, S. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. In *Proc. of the 20th Symposium on Operating Systems Principles (SOSP)* (Brighton, UK, Oct. 2005).

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

[29] WARFIELD, A., HAND, S., FRASER, K., AND DEEGAN, T. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference* (Anaheim, CA, Apr. 2005).

[30] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5th USENIX Symposium on Operating System Design and Implementation (OSDI)* (Boston, MA, Dec. 2002).