Kaleidoscope: Cloud Micro-Elasticity via VM State Coloring

Roy Bryant[1] Alexey Tumanov[1] Olga Irzak[1] Adin Scannell[1]
Kaustubh Joshi[2] Matti Hiltunen[2] H. Andrés Lagar-Cavilla[2] Eyal de Lara[1]

[1]University of Toronto, [2]AT&T Labs Research

Abstract

We introduce cloud micro-elasticity, a new model for cloud Virtual Machine (VM) allocation and management. Current cloud users over-provision long-lived VMs with large memory footprints to better absorb load spikes, and to conserve performance-sensitive caches. Instead, we achieve elasticity by swiftly cloning VMs into many transient, short-lived, fractional workers to multiplex physical resources at a much finer granularity. The memory of a micro-elastic clone is a logical replica of the parent VM state, including caches, yet its footprint is proportional to the workload, and often a fraction of the nominal maximum. We enable micro-elasticity through a novel technique dubbed VM state coloring, which classifies VM memory into sets of semantically-related regions, and optimizes the propagation, allocation and deduplication of these regions. Using coloring, we build Kaleidoscope and empirically demonstrate its ability to create micro-elastic cloned servers. We model the impact of micro-elasticity on a demand dataset from AT&T's cloud, and show that fine-grained multiplexing yields infrastructure reductions of 30% relative to state-of-the art techniques for managing elastic clouds.

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

General Terms Design, Experimentation, Measurement, Performance

Keywords Virtualization, Cloud Computing

1. Introduction

Cloud computing caters to bursty Internet workloads with a utility model that emphasizes pay-per-use and elasticity of provisioning. As with all utilities, there is a granularity associated with service delivery and billing. For Infrastructure as a Service (IaaS) clouds, the granule is the virtual machine (VM). Adopting virtualization as a building block yields distinct advantages in security, isolation and ease of management, but this coarse granularity imposes inefficient patterns that harm both users and providers.

In an ideal cloud, 'elastic' servers grow and shrink in tight concert with user demand. Currently, a load balancer adjusts the size of a pool of full-sized worker VMs [Amazon a;b] that are booted from scratch from a template. Unfortunately, this heavy-weight mechanism is a poor match for the operation model of an efficient utility. Creation is slow – new servers take a while to boot because it's a laborious I/O bound task. Moreover, this latency is hard to predict – instantiation latencies in Amazon's EC2 cloud have been observed to fluctuate sharply around a two-minute mean [Hyperic]. Furthermore, once booted, the server's performance-critical application and OS caches are essentially empty, which degrades performance when it is most needed to service demand spikes. Finally, VMs claim a full memory footprint even if they are required for only short periods of time and much of their memory is not actually used.

Therefore, server owners have incentives to keep VMs active for long periods, both to provide slack resources during long instantiation latencies, and because servers with large, warm buffers become crucial to overall performance and are too valuable to sacrifice. This practice may explain why the proportion of EC2 'Extra Large' instances (15 GiBs of RAM) has grown from 12% to 56% in a year, even while the total number of servers has tripled. Further, the proportion of servers running longer than a month has nearly doubled [RightScale]. These behaviors detract from the cloud vision of matching resource usage to actual demand, inflate user costs while still failing to achieve a good QoS if the load exceeds expectations, and curtail providers' ability to consolidate and optimize infrastructure use.

This paper proposes a vision of *cloud micro-elasticity*, in which cloud server elasticity is achieved through short-lived, transient clone VMs, which are copies of a running VM instance and allocate resources (memory, disk) only on demand. To enable this, we introduce color-based fractional

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Eurosys '11 April 10–13, Salzburg, Austria.
Copyright © 2011 ACM 978-1-4503-0634-8/11/04. . . $10.00

VM cloning, a new technique that allows the fine-grained management of VM state, and enables the swift instantiation of stateful VMs that allocate resources in proportion to use. By cloning a warm, running VM instead of booting a new one, our workers inherit their parent VM's state and do not require warming. They come online faster, reach peak performance sooner, and because short-lived worker VMs typically access only a fraction of their state, they can service transient spikes in load from within a smaller footprint.

Color-based fractional VM cloning uses a novel VM state replication technique. Instead of blindly treating the VM as a uniform collection of pages, it bridges the semantic gap between the Virtual Machine Monitor (VMM) and the guest OS by examining architectural information (e.g., page table entries) and other clues to glean a more detailed understanding of the guest's state. This higher-quality knowledge allows the VMM to optimize the propagation of state to clones by identifying semantically related regions. Specifically, we use VM state coloring to tailor the prefetching of kernel vs user space regions, code vs data regions, and to optimize the propagation of the file system page cache. Finally, coloring provides hints that guide memory consolidation by identifying regions with a high likelihood of content similarity.

To evaluate the performance of color-based fractional VM cloning, we implemented *Kaleidoscope*, an elastic server that reacts to transient load spikes by spawning fractional VMs. Experiments using elastic Web and Online Analytical Processing (OLAP) workloads show that Kaleidoscope significantly improves on the current state of the art. First, Kaleidoscope instantiates new stateful clones in seconds, and nearly matches the runtime performance of an idealized cloning strategy that uses zero-latency eager full state replication. Second, by bridging the semantic gap, Kaleidoscope is effective in finding state that is likely to be needed by the new clone. For example, for the OLAP workload, it achieves 2.9 times the query throughput with 43% less waste than color-blind cloning. Third, Kaleidoscope's fractional VM workers grow only as needed to satisfy new allocations or hold newly transferred state. In our experiments, the memory footprint of workers reached only 40% to 90% of their parent's memory allocation.

To further evaluate the advantages of color-based fractional VM cloning, we simulated its effects on traces collected from AT&T's hosting operation. The simulation shows that the finer-grained handling of VM state drastically reduces infrastructure use. With VM cloning to rapidly create new workers, a server can scale faster, and therefore requires much less slack capacity to deal with load increases. Also, the reduced memory footprint translates into a denser packing of VMs on physical infrastructure. The net result is a 30% reduction in infrastructure, which creates energy and money savings that can be shared with end-users via more attractive fine-grained pricing schemes.

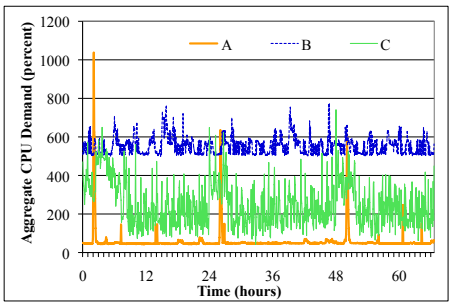


Figure 1. Aggregate CPU demand for sample customers

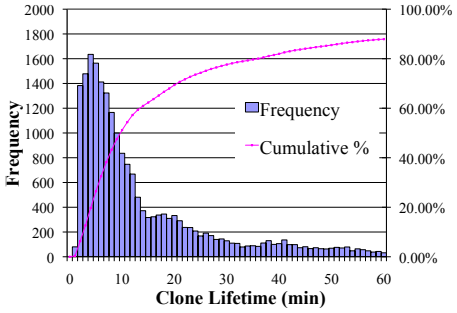


Figure 2. Elastic workers are typically short-lived

We also discuss the ways in which Kaleidoscope ensures correct and consistent behavior. Although VM cloning is not universally applicable without tuning, many legacy server applications, including the OLAP database and Ecommerce Apache Web server used in our evaluation, function correctly without modification.

This paper makes the following contributions. First, we introduce the notion of VM state coloring as a general mechanism to bridge the semantic gap and glean high-quality information on the runtime state of a VM. Second, we show how state coloring can be implemented efficiently by exploiting x86 architectural properties and guest kernel introspection extensions. Third, we present Kaleidoscope, a micro-elastic server that uses state coloring to optimize the replication and sharing of VM state, and delivers a QoS that approaches that of fully over-provisioned servers, while consuming resources proportional to the immediate demand. And fourth, we quantify the benefits of deploying Kaleidoscope servers using a data set of multi-customer demand extracted from AT&T's hosting operation, showcasing substantial savings for users and providers.

2. Real Data Motivates Micro-Elasticity

To use resources efficiently, elastic servers should grow and shrink in tight concert with user demand. We examine the potential for such elasticity using a month of demand data from AT&T's hosting operation (see Section 9 for more details about the data traces). AT&T hosting is a traditional hosting business in which customers buy rack-space to stat-

ically provision web-facing multitier applications, such as portals, shopping sites, and enterprise services.

Need for Elasticity. Analysis shows that over the whole month, an average customer tier has a mean demand of only 15.3% of its peak, thus indicating ample long-term fluctuations. Furthermore, demand elasticity also percolates to smaller timescales. Figure 1 shows the total CPU demand for three sample customer tiers. While the characteristics can be very different across customers, they all exhibit significant short-term variations, and thus could benefit from fine-grained elasticity.

Elastic Workers are Short Lived. Figure 2 shows the rate of creation and the lifetimes of elastic server workers, if we were to closely follow demand by maintaining a CPU utilization of between 70% and 90% across all workers. The results show a frequent creation of very short-lived workers: 23,214 workers would be created for a set of only 248 elastic servers. The mean worker lifetime would be only a little over 10 minutes, with over 85% of the workers needed for less than an hour. These workers, therefore, are essentially single-purpose entities that are frequently created to service a narrow workload during short periods of demand pressure, and have limited time to grow their active memory footprint. A mechanism that could allow them to be created cheaply, quickly, and with an allocation proportional to their use would be of great benefit.

3. Designing Efficient Micro-Elasticity

We achieve micro-elasticity by building upon live VM cloning, and augmenting its capabilities through two separate techniques: coloring of VM state to improve its propagation and sharing, and fractional VM allocations to minimize state footprint. Through the combination of these techniques we enable the swift instantiation of fractional, stateful VMs that are virtual copies of an existing server instance, but which are allocated resources (memory, disk, network) proportional to their actual use. In this section we provide background on cloning, illustrate its limitations and motivate the introduction of state coloring and fractional VM allocations.

3.1 Live VM Cloning with SnowFlock

SnowFlock [Lagar-Cavilla 2009] introduced the concept of live VM cloning across a cluster or cloud of physical machines. Cloning requires replicating the state of a VM, and SnowFlock achieves this with on-demand paging. In this approach, a clone is quickly created from a small architectural VM descriptor containing metadata, virtual device (NIC, disk) specifications, and architectural data structures such as page tables, segment descriptors, and virtual CPU (VCPU) registers. The clone VM then triggers page faults as it encounters missing pages of memory or disk, and any referenced state is lazily transferred by a copy-on-demand mechanism.

SnowFlock complements copy-on-demand with multicasting of VM state. Multicast enhances network scalability and results in implicit prefetching, as clones will receive replies to requests issued by sibling clones created at the same time, and presumably accessing similar code or data. SnowFlock's multicast need not guarantee delivery of state to all clients, only to the client explicitly requesting it.

3.2 The Challenges of State Propagation

SnowFlock adopted on-demand paging to minimize instantiation time and optimize resource usage. With on-demand paging, there is space to apply 'late-binding' optimizations that may overlap or hide the overhead of state propagation with useful work performed by the clone. Unfortunately, for many servers, on-demand paging as implemented by SnowFlock results in an extended warmup period in which performance of the new instance is significantly degraded due to blocking waiting for the working set to be fetched (discussed in Section 8.1). Our experimental results show that on-demand fetching is so inefficient that it negates the benefits derived from warm caches.

An alternative to on-demand paging is eager full replication; this approach is similar to traditional VM migration, with the difference that at the end, there are two VMs running. Unfortunately, eager full replication places heavy demands on the network and results in long instantiation times. In addition, it requires that memory be allocated for all the parent's state, much of which may not be used. On the upside, because all state is fetched eagerly, once started, the new worker can quickly achieve peak performance.

3.3 Color-Based Fractional VM Cloning

To achieve the benefits of both eager and on-demand propagation (fast VM instantiation, short warmup period, resource allocation proportional to use) without their respective shortcomings, we use two novel mechanisms that optimize VM cloning performance: *VM state coloring*, which discriminates otherwise uniform VM state into semantically-related regions allowing state to be efficiently prefetched and shared; and *fractional allocation*, which dynamically allocates memory to accommodate only the state that is actually accessed by the new worker.

Color-based fractional VM cloning makes it possible for users to achieve high resource utilization while still accommodating transient load increases at a low rate of service violations. Users keep just enough server instances to deal with the average short-term demand placed on the service (e.g., keep average worker utilization at 80%), and instantiate new transient workers to deal with any sudden load increases. As shown in Section 2, short-term demand spikes are prevalent and typically result in the creation of workers that are needed for ten or less minutes. When demand for the service subsides, the transient workers are shut down and their resources returned to the cloud. Our approach significantly reduces memory footprints relative to static overprovision-

ing, but it should be noted that this benefit is not entirely free. It incurs a modest network cost for the fractional state transfer.

By cloning a warm, running VM, new workers come online within a few seconds, and by inheriting their parent VM's state, they do not require warming. Using VM state coloring, we efficiently prefetch state to mitigate the page fault blocking associated with on-demand VM cloning, which significantly boosts performance. By allocating memory on demand and sharing identical pages, fractional VM workers save space, and short-lived workers service transient spikes in load from within a smaller footprint.

4. VM State Coloring

Historically, VM state has been treated as uniform binary state, enabling virtualization to simplify many tasks. For example, by saving the entire RAM of a VM as one flat binary file, computation migration can be implemented robustly [Satyanarayanan 2005]. However, the limited information that the VMM has about the state of the guest, referred to as the semantic gap [Chen 2001], can constrain the effectiveness of system services such as I/O scheduling or malware detection [Jones 2006a;b, Litty 2008].

We have devised a set of *VM state coloring* mechanisms that allow us to classify the memory of a VM into a set of semantically meaningful regions. We color VM state by inspecting *architectural information*, such as that contained in page table entries, and performing *introspection* on the guest kernel's data structures. Without adding significant overhead, the discrimination of otherwise opaque VM state into semantically-related regions allows us to optimize VM cloning performance.

4.1 Architecture-based Coloring

Page table entries in x86 contain a wealth of information regarding memory pages. First, pages are tagged as executable or not by the NX bit. Second, pages can be tagged as belonging to the kernel or user space with a 'user' bit. Even if the OS (or the VMM) chooses not to use this bit, a walk of the page tables allows for the reconstruction of a corresponding virtual address: both 32 and 64 bit OSes typically allocate the lower portion of virtual addresses for user space and the higher portion for kernel space. Third, commodity x86 OSes present a bijective mapping between user-space processes and root page table pages, allowing for the discrimination of state unique to a given process.

Using this architectural information we can color memory in roughly four groups: kernel code, kernel data, user code and user data. Further granularity can be achieved by splitting the user colors on a per-process basis, which is left for future exploration. We also note that using x86 architectural information in this fashion makes our coloring robust, as its usefulness is completely independent of the software stack.

A complication arises in 64bit OSes such as Linux, which install a contiguous mapping of the entire physical memory into the kernel data space for expedited access to physical memory addresses. This mapping is called the 'direct map'. As a result, all pages including free ones, have at least one page table mapping. Many kernel data structures are only reached via their address in the direct map, without any other mappings needed. To tell apart kernel data from free (or uncolored) pages, we need to go beyond page table analysis.

4.2 Introspective Coloring

At the root of the semantic gap problem is the fact that the OS has a more complete knowledge of system resources than the VMM. Both the Xen VMM and the Linux OS maintain a 'frame table,' an array of compact records describing the properties of a page. Among other things, a Xen frame table entry indicates the owner VM and the number of page table mappings of a page across all VMs in the system.

A Linux frame table entry has more useful information. First, page records corresponding to frames of memory that belong to a file lead to a radix tree containing all fellow pages mapping the same file. Memory frames can thus be colored as belonging to the file system page cache, and for further granularity, we can group the pages used for each individual file. We underscore that the latter requires no knowledge of the actual file attributes nor file system internals.

Second, the page record indicates whether the page is being used by the guest, or is free. It reflects 'real' usage of the page and is intended to capture all usage, through a combination of a reference count and a set of flags. We thus rely on a page structure record with all of its fields and count reset to identify pages as free. This enables differentiation of free and kernel data pages. Given the circumstances, we find this to be the most conservative method and also the most robust, as the notion of a page structure record is broadly applicable.

We note that the original implementation of VM cloning [Lagar-Cavilla 2009] performs a form of paravirtual coloring. By instrumenting the kernel page allocator it could identify victim pages used for new memory allocations, and prevent the clone VM from issuing a request for a victim page which will be immediately overwritten. There is naturally a very high correlation between victim and unused pages.

4.3 VM State Coloring for Efficient Propagation

We apply VM state coloring at the point of cloning. During the generation of the architectural descriptor, all page table entries have to be processed to turn MFNs (machine frame numbers, i.e., the memory frames of a host) into PFNs (physical frame numbers, i.e., the memory frames of a guest VM) – this translation is later undone when the clone VM is created in a different host. At this point, architectural coloring is applied to partition memory into four disjoint regions; the kernel and user space regions are each subdivided by data vs executable.

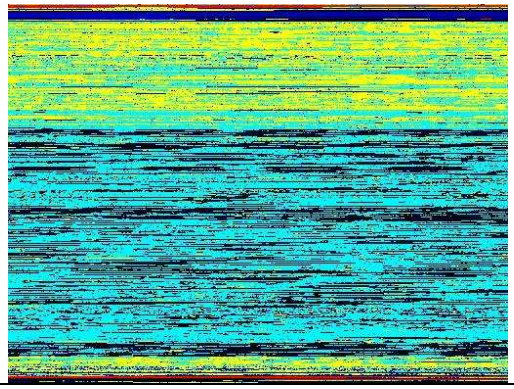


Figure 3. Color map Rendering of a memory snapshot of a VM running the SPECweb Support workload. X axis is page number and wraps around for presentation. Legend: Page Cache - yellow; User and Kernel Data - light and dark blue; User and Kernel Code - light and dark red; Free - black.

After the architectural descriptor is generated, a *memory server* is left running on the parent VM's host machine. This memory server keeps a map of the parent VM's memory frozen at the point of cloning (a 'checkpoint'), and uses copy-on-write to allow the parent VM to proceed with execution while serving the frozen image to clones. The memory server is aware of the architectural coloring information, and is able to examine at will the VM's frozen memory, and in particular the guest kernel's frame table. In this way, the memory server performs introspection to identify free pages, which are re-colored to form a fifth region. Further, the memory server identifies pages belonging to the file system page cache, and also re-colors these pages on a per-file basis. For pages that have multiple colors, for example an executable page that is also in the file system page cache, the specific color that will be used is application-dependent and configurable.

To better understand the advantage of coloring state, consider the color map (Figure 3) of a Web server at the point of cloning. For the sake of presentation, we have not refined the coloring to be per-file. A key observation is the interspersing of different colors in the physical memory space of the VM due to virtual-to-physical translations and memory fragmentation.

Each clone is aided by a *memtap* process in charge of obtaining the memory the clone needs. Memtap's objective is to keep the clone's VCPUs blocked as little as possible. A blocked VCPU not only affects the QoS of the request it is serving, but also effectively disables the guest kernel's ability to multiprocess and service any other requests with that VCPU. When a clone faults on a missing page and requests it from the parent, it also asks for suggestions of related pages that are likely to be needed soon. The memory server uses the principle of spatial locality within the color in which the explicit request falls, so the pages prefetched may be scattered across physical memory.

Table 1. Kaleidoscope's prefetching is tuned by color.

Color	Window	Color	Window
Kernel Code	4	Kernel Data	12
User Code	4	User Data	16
		Page Cache Data	8

A naïve alternative to per-color prefetching is 'color-blind' prefetching, which lumps together multiple unrelated colors (including free pages, which cannot be distinguished without the use of introspective coloring) in the same prefetch block. Because prefetched pages are allocated even if they are not used, color-blind's less targeted approach wastes memory. We show in Section 8 how the color map increases the accuracy (fewer 'wasted' fetches of unneeded pages) and efficiency (more faults avoided) of prefetching, and how it outperforms color-blind.

In the Kaleidoscope prototype, we tuned the prefetch strategy by semantic region to further improve efficiency (see Table 1). We obtained our per-color policies by post-processing the state propagation activity on a set of experiments with coloring turned on, but no prefetching enabled. We simulated lookahead and pivot policies with different window sizes, and chose the most effective ones for the evaluation in Section 7. Kaleidoscope's primary distinction is to use a reduced prefetch window for executable pages, regardless of whether they reside in the file cache. Secondarily, we found that by refining the window size by data page properties provided a modest improvement. By increasing the prefetch window for user data and reducing the window for data pages in the file system page cache, efficiency and accuracy improved by 5.5% and 0.7% respectively, compared to a uniform window size of 12 for all data pages. We leave for future work the online prediction of prefetching policies using techniques similar to our post-processing profiling.

5. Fractional Allocation

When a typical VM is created, all of its backing memory is pre-allocated. This is also the case in the SnowFlock VM cloning implementation, where the cloned VM's memory is allocated eagerly, and subsequently populated with the state that is fetched from the parent VM on-demand. In contrast, the fine-grain, per-page usage knowledge we extract through coloring opens up the opportunity to optimize the memory footprint of the cloned VM.

First, on-demand fine-grained propagation of the memory of a clone VM calls for the on-demand allocation of its memory frames, a technique we call *fractional allocation*. Second, the hints extracted through memory coloring can direct content-based sharing of memory pages across VMs with great efficiency and modest effort.

5.1 Implementing Fractional Footprints

Fractional allocation is achieved by allocating on-demand the underlying pages of memory of a cloned VM. This is re-

alized through the concept of a 'ghost MFN'. A ghost MFN has the property of serving as a placeholder that encodes the clone's PFN that it backs, and a flag indicating absence of actual allocation. The ghost MFN is placed in lieu of an allocated MFN in the page tables, and the PFN-to-MFN translation table that each Xen paravirtual guest maintains. The first guest access to the PFN triggers a shadow page fault in the hypervisor, which is trapped and handled by allocating the real MFN to replace the ghost. Note that the very same page fault is already handled to draw missing state from the parent VM. Separately, as state is prefetched by the memtap memory daemon, the daemon itself can request the allocation of the MFNs needed to store prefetched content. Finally, we avoid fragmenting the host's free page heap by increasing the granularity of requested memory chunks to, for example, 2 GiBs or 512 pages at a time, while still replacing ghost MFNs one at a time.

5.2 VM State Coloring for Memory Deduplication

The color map's semantic hints allow clone VMs to significantly reduce their memory footprint for very little cost. Certain colors, specifically page cache pages and executable pages (kernel or user-space), yield a relatively high probability of inter-VM sharing within the same host. Other colors are typically populated with data (stacks, unaligned buffers, heap pointers) that all but nullify the chances of sharing. A similar principle is exploited in related para-virtual sharing work [Milosz 2009].

During construction of the descriptors, we calculate the 128-bit hash values [Hsieh 2004] of pages in candidate colors and include them for use by the clone. Each host maintains a content-addressable store (CAS) of shareable pages it has previously fetched for different clones, with in-use pages stored once in physical memory and referenced by each live clone that needs it. Using coloring to guide content sharing is more efficient than previous work [Gupta 2008, Waldspurger 2002] for three reasons. Because hash values are calculated once on the parent and passed to every clone, we efficiently avoid repetitions of brute force traversals of memory and hash calculations – Section 8 shows that our color-directed sharing captures most of the sharing opportunities among VMs with an order of magnitude less overhead. Also, cloned worker VMs have 'fate determinism': they are single-purpose, transient, and seldom start new processes or significantly change their behavior. Thus the expense of periodically re-scanning the memory for sharing opportunities is not warranted. Finally, sharing is applied only to seldom-updated executable and file system page cache pages, which minimizes the overhead cost incurred by breaking sharing of pages that are updated.

Sharing pages with identical contents complements fractional allocation to reduce the footprint of clone VMs. The net effect is a clone footprint that grows as a function of the workload. Because the footprint reflects state fetched as-needed, minus color-directed sharing, it allows the clone to perform the work of a fully stateful VM with an effective footprint which is much less than what is typically achievable via ballooning or brute-force memory deduplication.

It should be noted that both footprint reduction mechanisms have a welcome performance side-effect. Sharing hits prevent round-trips to the server to fetch the necessary page. Coloring prevents the needless propagation of free pages, as their actual contents are irrelevant – with fractional allocation we can simply take a free page in the host and scrub it. We note that in an environment without fractional allocation, coloring could enable the mapping of all free pages to the same underlying physical frame. This would work as automatic ballooning, without the need for guest collaboration, and with instantaneous self-regulation as shares are broken. It would also yield a higher sharing ratio than content-based sharing by disregarding the actual (unused) contents. We leave exploration of this opportunity for future work.

6. The Kaleidoscope Prototype

Kaleidoscope is a prototype elastic server that uses live VM cloning, state coloring, and fractional footprints to create VM workers in response to bursts in load. New workers are created in seconds and inherit the warm state of their parent VM. We describe the architecture of a Kaleidoscope server, how the worker pool is managed, and how new clones interact with secondary storage. We close the section with a discussion on the mechanisms we provide to guarantee correctness of live-cloned servers.

6.1 Kaleidoscope Server Architecture

A Kaleidoscope elastic server is a dynamically-resizing cluster of VMs. There are three roles in the cluster. First, there is a parent VM which is a traditional (i.e., booted from scratch) VM containing the necessary software stack. Second, fractional worker VMs are cloned from the parent as transient workers to handle load fluctuations. Third, a gateway VM interfaces the cluster with the outside world and manages the load using the Linux IP Virtual Server (IPVS). It routes client requests to workers, monitors the number of incoming client connections, and spawns new clones when a high water-mark threshold is exceeded. In this manner, no server in the pool, parent included, is ever overloaded – provided there are physical resources available to create more clones. Conversely, when the load drops, extraneous workers are starved of new connections and discarded once their work is complete.

Kaleidoscope currently creates a fresh checkpoint for each generation of clones to ensure the warmth of the inherited file system cache, although this could be tuned to reuse 'master' checkpoints to conserve resources if slightly cooler buffers provide sufficient performance. Similarly, its scaling speed can be tuned by configuring how many new workers are created simultaneously. Because Kaleidoscope can multicast VM state, in a highly bursty environment with

flash crowds we can aggressively clone multiple workers at each step, and subsequently scale back quickly if the large step proves unwarranted.

The local disk of the parent VM is cloned to all child VMs. Typically, this is the root disk with application binaries and libraries, while high-volume application data is served by a storage backend. The semantics of disk cloning are identical to memory: clones see the same disk, although modifications to it remain private, and are discarded upon clone termination. We have not seen the necessity to improve upon the original disk cloning implementation [Lagar-Cavilla 2009]. The local disk is provided purely on-demand, but is rarely used by transient clones who find most of their requests satisfied by in-memory kernel caches, whose propagation we do optimize. We also note that the virtual disk is implemented as a sparse flat file for clones. This implicitly guarantees fractional allocation of disk blocks as a function of use, mirroring the behavior of a clone's memory footprint.

Multi-tier stacks rely on a variety of engines for data backends: RDBMSes like MySQL, caching layers like memcached, infrastructure key-value stores like SimpleDB [Amazon c], and user-deployed key-value stores such as Cassandra [Lakshman 2009]. With the complexity associated with deploying a data backend for a given application, we have decided to stay clear of any specific storage backend architecture in this work. We assume that, in most cases, a backend can be found that is fast enough to render the processing servers the bottlenecks. In this paper, workers get their static data from an NFS server.

6.2 Correctness and Consistency

Spontaneously cloning random, unsuspecting servers may yield unsatisfactory results. For instance, if an email server were cloned in the middle of sending a queue of messages, the clones would unwantedly send duplicates of the messages still in the queue of the parent at the time of cloning. As another simple example, if an application server keeps a count for the number of sessions it has handled, and periodically commits it to an underlying database, the count will drift off in each clone, and conflicting numbers will be committed by each clone to the database.

So when is it safe to use Kaleidoscope? Kaleidoscope's cloning is not intended to be a provider-driven primitive that is applied to an unmodified server. A power user or system administrator who is familiar with the server's behavior should decide whether Kaleidoscope should be deployed, when it is safe to clone, and whether code must be modified. We assume providers still exert ultimate control by limiting the resources available to users, in order to thwart DoS attacks, 'fork bomb'-analogous attacks, and other threats.

Despite the nominal necessity for expert intervention, there is a large set of applications for which Kaleidoscope cloning is trivial and harmless to apply. In our experience, many common legacy server applications work correctly with minor or no modifications. Our Apache Web server with static files worked fine, as did OLAP database analytics, and the SPECweb suite of benchmarks, which store their client account information in a back end. Because Kaleidoscope does not hand off live connections from the parent to clone, latency sensitive applications such as video servers should also work well.

6.2.1 In-Flight Requests

Kaleidoscope can be applied only to servers with built-in support for maintaining state consistency amongst a dynamically changing pool of load balanced nodes. Beyond this basic requirement, Kaleidoscope servers need to handle the case where a server instance is processing a request at the time of cloning (a likely scenario in a busy server), because each of the created clones will also be doing so when it comes alive.

Kaleidoscope servers need to ensure that operations that are under way at the time of cloning are treated as follows: (1) read-only operations to clone state need no extra handling; (2) operations that modify state cached on the clone that needs to remain consistent must either finish or abort on all nodes. An alternative is to sidestep the issue by queueing new write operations at the load balancer and cloning only after all ongoing writes have committed at the master (a 'write barrier'); (3) requests that may modify cached data that need not be kept consistent across nodes need to be routed consistently. Many servers distribute load by assigning an account or session persistently to a node, and the load balancer routes requests observing the node-session binding. Sessions handled by the parent VM at the point of cloning need to be discarded by the clones, and the load balancer must still route requests through the parent VM; (4) external side effects (a write to a database server, file system) need to happen only once. Conveniently, this is the default behavior of many applications, such as database servers, which roll back in-flight transactions submitted over dropped connections. Alternatively, an intermediary arbitration layer could handle clone-born duplicates and ensure that that visible external side-effects are emitted only once.

6.2.2 Consistency Mechanisms

Kaleidoscope provides four mechanisms that help applications maintain correctness and consistency across a VM clone operation:

Programatic Integration One of the major appeals of live VM cloning is the ability to integrate cloud fan-out decisions into program logic, with an interface similar to other privileged system calls. Much like UNIX `fork()`, the process within the server invoking the clone call will receive, upon success, a unique clone ID. This allows programs, modified or written from scratch, to immediately react to cloning as part of their flow control logic. We note that throughout this paper cloning is triggered by a load-balancer; the clones

themselves are not modified to perform an explicit call, but can retrieve their ID through a `proc` interface.

IP Address Adjustment Upon cloning, a clone's IP address is automatically reconfigured before any inbound or outbound packets are allowed. Clones share an internal private network with their parent and other select entities such as the load balancer or the backend server. Clones are assigned a new IP address within the private network as a function of their ID. The reconfiguration of IP addresses requires no developer intervention.

On the parent's side, network connections that are open at the point of cloning remain open and working. If the parent VM accepts client connections, Kaleidoscope does not attempt to hand these off to the clone. On the clone VM, the connection is inherited but the assignment of a new IP address during cloning forces all inbound and outbound connections to drop – in many cases, this will result in automatically discarding session state that the parent handles entirely. However, the new clone has to graciously deal with broken connections to system resources, such as the load balancer and backend storage. Once those connections are re-established, the new, cloned workers require no further network-plane intervention to ensure correctness.

Reconfiguration Hook Much like the Linux hotplug infrastructure calls user-space scripts through the creation of `kobjects` and `uevents`, the guest kernel will automatically invoke a reconfiguration script after cloning, if one has been registered. As previously discussed, our only modification to the servers used in this paper was to have clones deal with automatic IP reconfiguration by remounting their NFS connections and subscribing to the load balancer through a simple script invoked through the reconfiguration hook.

SIGCLONE For more involved scenarios, a special asynchronous signal can be sent to processes once a clone comes alive. Handling of this signal will require code level changes, and is only intended for complex situations – none of our experiments in this paper used SIGCLONE. Processes explicitly subscribe to receive SIGCLONE, which will be sent as a one of the available POSIX real-time signal and thus terminate the process if unhandled.

7. Experimental Setup

As we anticipated, it is much faster to clone workers than to boot them (five seconds vs. two minutes). Thus, our experiments look past this major advantage, and instead examine the value of micro-elasticity and warmed caches by comparing cloning to an idealized cloud where new workers boot instantly at no cost. We measure the initial behavior of six worker prototypes under five workloads, and examine the benefits of fractional footprints and page sharing.

7.1 Worker Prototypes

Each of the six prototypes strikes a different balance between performance and efficiency. All prototypes run on identical hardware, and all applications, including Apache and MySQL, run without modification. Where indicated, the prototypes are previously warmed by a related workload (different random seeding). All cloned worker VMs run on a different physical host than their parent. The prototypes are:
Warm Static Previously warmed, this statically overprovisioned server sets the upper bound for achievable performance. This worker is equivalent to an idealized zero-latency cloning strategy that uses eager full replication.
Cold Standby Recently booted and held on cold standby, it represents an idealized version of current elastic clouds, where copying and booting workers is instant and free. Cold standby results, if extended by an average of two minutes, are comparable to results from today's commercial clouds.
Minimal Clone Newly cloned using basic VM cloning, this worker faults on every page it needs with no wasted fetches.
Kaleidoscope Newly cloned worker that uses the color map to prefetch close PFNs within the same semantic region. The prefetch window is tuned by semantic region (see Section 4.3 and Table 1).
Aggressive Clone This color-blind clone is tuned for higher performance and prefetches the next 16 PFNs.
Conservative Clone Color-blind clone tuned for efficiency by constraining its prefetch window to four pages.

7.2 Web and Database Workloads

We tested each prototype under five workloads.

OLAP Server In the absence of an accepted standard OLAP benchmark, we drive the MySQL database server with ad hoc decision-support queries drawn from the TPC-H [TPC-H] benchmark. We use only 17 of the 22 TPC-H queries because five are long running and are poorly suited to evaluate a server's initial, transient performance. Our elastic OLAP server is read-only and does not support database writes, which is consistent with standard industry practice, where complex decision-support queries are typically run against a read-only copy of a business's main OLTP database to avoid performance interference.

Web Server with Static Content Apache Web server is driven by Httperf [Mosberger 1998], requesting 3,000 files selected randomly from 7,500 static HTML files (random sizes, 1 to 128 KiB). Where warmed, all files are loaded in random order into the file system cache.

Web Server with Dynamic Content Apache is driven by requests for dynamic content by the industry-standard SPECweb 2005 [SPECweb 2005] benchmark. This benchmark consists of three standard workloads: *Banking*, where users check balances and manipulate accounts through secure connections; *Ecommerce*, where users search, browse and purchase products; and *Support* where users browse and search product listings and download files through insecure

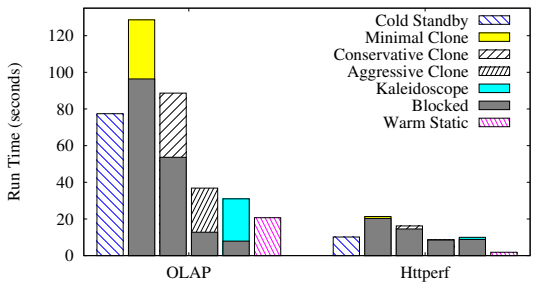


Figure 4. Prefetching reduces page faults and VCPU blocking, so benchmarks finish sooner. Legend top-to-bottom matches columns left-to-right.

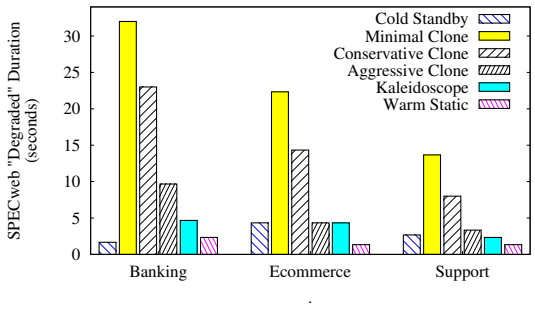


Figure 5. Prefetching reduces the length of time that prototypes fail to meet SPECweb's minimum acceptable QoS. Legend top-to-bottom matches columns left-to-right.

connections. Each workload is a closed-loop simulation of user visits based on Markov chains derived from the web logs of typical sites. The workloads draw from many representative files and model a typical distribution of file sizes and overlaps in file access patterns.

All workers are implemented as Xen 3.4.0 VMs running 64-bit Linux (Debian Core 5) on eight identical Sun Fire X2250s (each with eight Xeon cores, 8 GiB RAM, and dual Gigabit Ethernet). Workloads are generated on five Dell servers (four Xeon cores, 4 GiB RAM, Gigabit Ethernet). OLAP workers run MySQL 5.1.47 in a 2 GiB VM. Web workers run Apache 2.2.9 in a 768 MiB VM. All workload data files are accessed through NFS.

8. Results

In this section we discuss the performance, efficiency and resource use of the six prototypes under the five workloads.

8.1 Performance

In our experiments, we measure performance in two ways. For the OLAP and Httperf benchmarks, which impose a fixed volume of work, we measure the run time to assess performance (Figure 4). For SPECweb, however, the run time is fixed because the benchmarks sustain a specified load over time, and instead we measure whether the server's QoS is

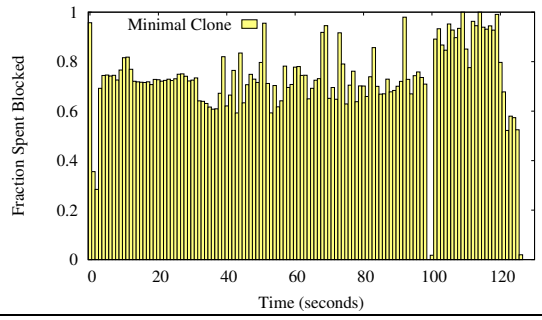


Figure 6. VM cloning copies state on demand, leaving the VCPU frequently blocked while page faults are serviced.

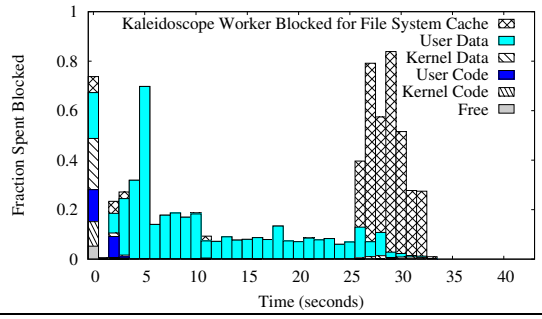


Figure 7. With fewer faults and less blocking, new Kaleidoscope workers run database queries faster. No faults were registered after the first 33 seconds.

'acceptable', which is defined as meeting a minimum latency for a specified percentage of requests. For these benchmarks we measure the 'degraded duration', the number of seconds for which SPECweb finds the instantaneous QoS 'unacceptable' (Figure 5). Another difference between the two groups of benchmarks is the value of their accrued state. For OLAP and Httperf, the warmed memory state speeds future work, as evidenced by the margin at which Warm Static outperforms Cold Standby (factors of 3.7 and 5.6, respectively). For the SPECweb benchmarks, the dynamic content varies for each request, which renders cached state much less valuable and causes Cold and Warm to perform nearly equally.

The Minimal Clone's lazy state propagation exacts a heavy price, resulting in the worst performance on every benchmark. Minimal Clone is unable even to match the Cold Standby performance, showing that the cost of copying state purely on demand negates the entire benefit of warmed memory state. Figure 6 provides insight into Minimal Clone's poor performance. The plot shows that fetching state purely on demand leaves the VCPU frequently blocked while page faults are serviced.

In contrast, Kaleidoscope nearly matches the performance of Warm Static for all benchmarks. For OLAP, it leverages the inherited warm state to outperform Cold Standby by a factor of 2.5, and ran only 10 seconds slower than Warm Static, achieving 67% of its best-possible through-

put. By comparison, Cold Standby achieved only 27% throughput during its 77 second run time. (Note that these performance gains would be even more dramatic if the latency and overhead of booting the cold VM were included.) Figure 7 shows that Kaleidoscope's color-directed prefetching largely eliminates the state transfer cost that so hampered Minimal Clone. *In summary, Kaleidoscope's approach to state replication achieves the fast instantiation time associated with on-demand cloning while coming very close to matching the runtime performance of eager full replication.*

Both color-blind prototypes performed better than Minimal Clone, but failed to materially beat Kaleidoscope on any benchmark. The key is that prefetching a page before it is needed eliminates faults, reduces VCPU blocking and boosts performance, whereas prefetching an unneeded page wastes network bandwidth, and in our fractional footprint environment, memory allocation. Figure 9(a) shows the relative fault reduction and waste of blind prefetching for various window sizes, with conservative strategies toward the top left, and aggressive toward the lower right.

By bridging the semantic gap, Kaleidoscope is better at estimating whether pages are likely to be needed (Figure 9(b)), and eliminates more faults with less waste. For the OLAP workload, it outperforms the Conservative Clone and achieves 2.9 times the throughput with 43% less waste. The Aggressive Clone is tuned for higher performance, but still falls 16% short of Kaleidoscope's OLAP throughput and wastes seven times as much prefetch bandwidth and clone memory allocation.

8.2 Scalability

To maintain an acceptable QoS for large spikes in load, it is important that Kaleidoscope's performance scale well with the number of simultaneous clones. Figure 8 shows that, for up to eight simultaneous new workers under the OLAP workload, Kaleidoscope scales well and outperforms Cold Standby. Kaleidoscope's inheritance of the parent's warmed caches is efficient, and multicast is effective in distributing the universally needed pages to the sibling clones. In fact, each clone's fetch act as prefetch for others, which boosts overall performance. In contrast, Cold Static workers tend to slow each other as they contend for the central database files.

Larger-scale Kaleidoscope elasticity can be achieved by cloning multiple parents in parallel. Although not implemented in the current prototype, clones older than several minutes could be converted into cloneable parents by transferring the full VM image. This configuration should support an exponentially increasing load that doubles every minute.

8.3 Lightening the Backend Load

Clones, with their inherited warm caches, sometimes exert less load on the backend storage than cold workers that warm their caches from scratch. This secondary benefit is most pronounced for the Httperf benchmark, where static HTML files stored in the page cache save 200MB of reads,

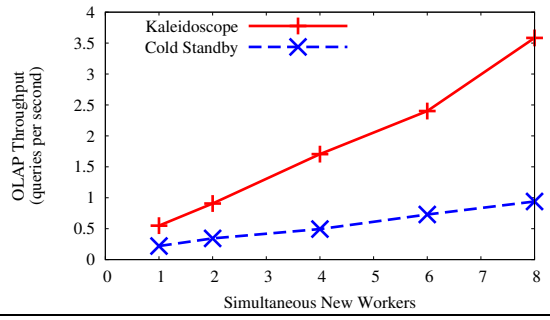
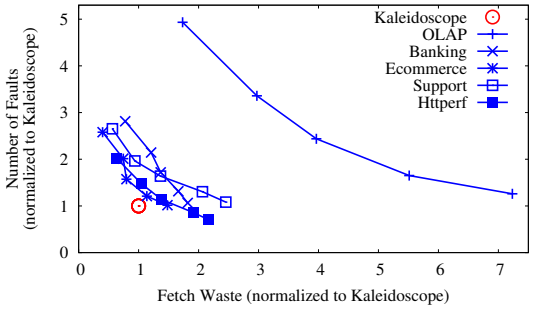


Figure 8. The database's OLAP throughput scales well with the number of simultaneous clones.



(a) Locality-based fetching fails to match Kaleidoscope's fault reduction to waste ratio. Windows from 4 (top left, less waste more faults) to 16 (bottom right, less faults more waste) pages.

Workload	Pages Needed	Faults Avoided	Pages Fetched Unnecessarily
OLAP	193,788	163,987	8,195
Banking	97,451	76,675	21,341
Ecommerce	75,044	58,734	17,352
Support	65,786	51,387	13,589
Httperf	58,850	39,980	8,625

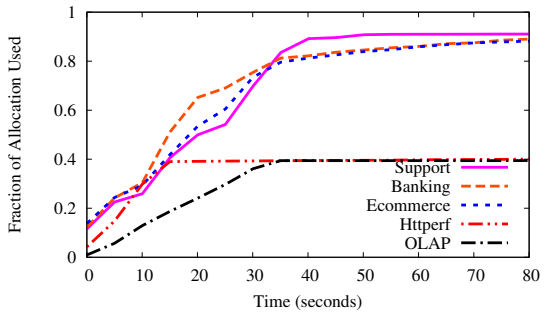
(b) Kaleidoscope's color-directed prefetching efficiently avoids most faults with relatively few wasted fetches.

Figure 9. Prefetch Efficiency Kaleidoscope's use of semantic hints yields better fault reduction with less waste.

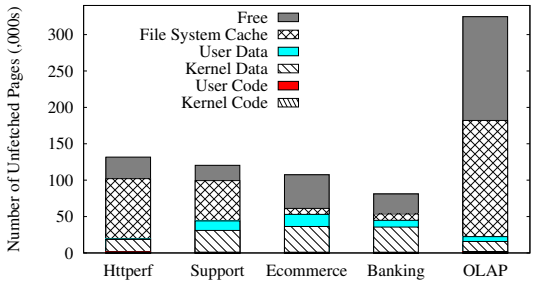
and for OLAP, where MySQL's user space data structures are sufficient to satisfy future queries, eliminating 890MB of backend reads. The least benefit is derived for the SPECweb workloads. Because they randomly selects files from a very large set, cached results are statistically unlikely to help.

8.4 Fractional Footprints

Kaleidoscope's fractional VM workers grow only as needed to satisfy new allocations or hold newly transferred state, as illustrated for the five workloads (with page sharing disabled) in Figure 10(a). For the SPECweb workloads with their large numbers of files and simulated users, the workers grow to claim approximately 90% of their allocation within the first minute. The Httperf and OLAP benchmarks are quite different, and under their intense but narrower loads,



(a) Kaleidoscope workers allocate memory as necessary, an advantage when spikes in load are short lived. Traces include transferred state plus new allocations. Legend matches plots top-to-bottom.



(b) Kaleidoscope workers avoid unneeded pages, mostly from the file system cache and free list. Legend matches bars top-to-bottom.

Figure 10. Fractional Footprints Kaleidoscope's workers efficiently avoid fetching or allocating unnecessary pages.

Kaleidoscope workers reached only 40% and 39% of their nominal memory size, respectively. Figure 10(b) shows the distribution of the unfetched parent state by semantic region.

The 3,000 static HTML files of the Httperf benchmark are easily cached with capacity to spare, and since future requests hit the same files, the allocation remains stable. As may be common given the coarse sizing of VMs available in today's commercial clouds, the worker is oversized for the load requirements, and Kaleidoscope offers the attractive possibility of efficiently retaining the unused capacity for other purposes. The OLAP worker is similar. It, too, is oversized, and clones of the parent VM are even more so – fractional footprint results in over one GiB of savings. Whereas the parent temporarily needed memory to cache the database files while it populated its user space data structures, the cache is later not needed to service requests and therefore remains unfetched and unallocated by the clone.

8.5 Sharing Identical Pages

A 'fair' evaluation of the content-addressable store (CAS) is important because results are dependent on the experimental design. For example, clones of the same parent and from the same, or even different, generations are extremely similar, and for read-only workloads could be nearly identical. This would skew the analysis of sharing opportunities. We therefore assess Kaleidoscope's page sharing with clones of the MySQL and Apache parents as follows. We assume that a Kaleidoscope deployment would avoid placing siblings on the same host, but that hosts might be used for workers of several distinct parents. To determine the 'initial sharing opportunities' we consider two clones and their initial state as inherited from each parent, and count the duplicate pages present in the union of their states. As the clones service their workloads (OLAP and Banking respectively), they update some of their pages, which may require that shares be broken. After both workloads are complete, we consider the 'net savings', the count of duplicate pages that are still shared.

Table 2. Color-Directed vs Blind CAS

	Kaleidoscope	Blind CAS
Size of CAS Cache	22,507	248,717
Pages Hashed	26,166	258,072
Saved by Sharing	5,522	9,355
Lost to Divergence	1,848	2,413
Net Savings	3,674	6,942

Kaleidoscope provides very high CAS efficiency by tracking hash values only for those pages that are the most likely to be sharable, and are the least likely to subsequently diverge. Table 2 shows the initial and longer-term sharing opportunities for the Banking and OLAP benchmarks running on the same host. By tracking only free, kernel (except the file system cache), and executable pages (including those in the file system cache), *Kaleidoscope captures 53% of the sharing opportunities while computing hashes for only 10% of pages, and performs an average of 7.1 hash calculations per page saved. Blind CAS netted additional pages at 10 times that cost.*. Kaleidoscope is not intended to compete with general CAS systems, but is rather taking advantage of savings it encounters at nearly zero cost. It should be noted that Kaleidoscope has relatively few sharing opportunities because it has already eliminated the wholesale waste from free pages by avoiding their allocation entirely.

As a further benefit, the local CAS cache can be used to service clone page faults without incurring the round trip cost of fetching the page from the parent VM. This could happen frequently where hosts are used for subsequent generations of workers servicing later spikes in load.

9. Implications for Cloud Data Centers

Finally, we examine the implications of deploying Kaleidoscope for the QoS, resource use, and infrastructure requirements of elastic clouds. We conduct a simulation-driven study using one month of CPU and memory demand data collected from AT&T hosting in January 2010. The data is for a subset of 248 customers' tiers hosted on a total of 1,740 statically allocated physical processors (PPs) and collected at 5 minute intervals. The selection of customer tiers was based on those customers who had instrumentation of processor and memory consumption turned on – from that set

we retained only those PPs devoted to web and application server tiers. We consider a scenario in which this demand is served by a hypothetical cloud data center that contains identical physical machines (PMs) with 16 CPU threads and 24 GiB of RAM. VMs are packed into PMs using a first-fit bin-packing algorithm - a newly created VM is allocated to the first PM which has sufficient memory and CPU capacity available.

CPU demand (% PP used) is aggregated over all PPs belonging to a customer tier at every time interval and then divided equally amongst the number of VMs. When the aggregate utilization (i.e., aggregate CPU demand/CPU capacity across all VMs belonging to the tier) for a customer exceeds the high threshold T_H, sufficient additional VMs are created to reduce utilization back below T_H. When the aggregate utilization falls below low threshold T_L, enough VMs are removed to bring utilization back between T_L and T_H. The thresholds determine how efficiently the cloud's resources are utilized. High values will delay new VM creation and hasten the destruction of underutilized VMs, and fewer VM's will be needed to satisfy a given demand.

The additional capacity of a newly instantiated VM is brought online only after an *instantiation interval* whose duration depends on the VM creation mechanism. We consider an 'overload' an event in which the CPU demand at that point in time exceeds the current VM allocation. Overloads happen because the VM creation mechanism used is too slow and VMs are not created with sufficient anticipation. We measure QoS degradation due to overloads by the accumulated value of the 'unmet demand' in CPU-seconds - i.e., one CPU-second implies a shortfall of an entire CPU core's worth of demand for the period of one second.

We estimate the seconds of unmet demand, the CPU-hours used by all the customers, and number of physical machines required in the cloud for the following scenarios:

Warm Static: Each physical processor (PP) is mapped to a single VCPU VM (100% demand) with the PP's peak memory allocation. VMs are statically allocated at the beginning of the simulation. This represents traditional over-provisioning with the best performance, and meets all demands at the expense of a large infrastructure commitment.

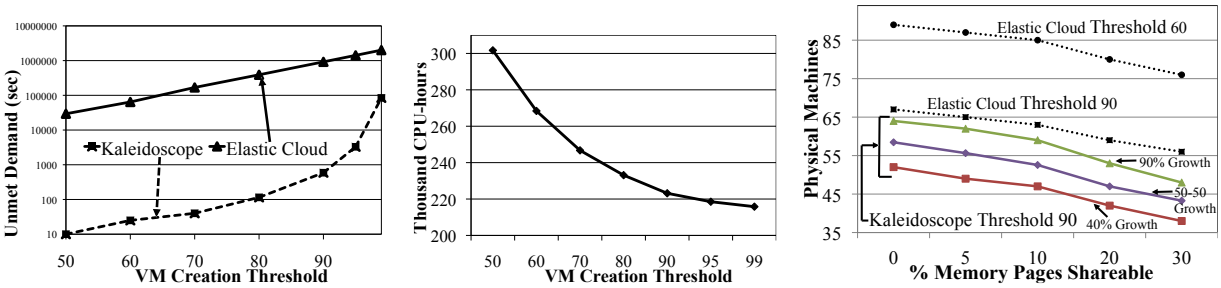
Elastic Cloud: Simulates current elastic clouds with dynamic addition and removal of VMs. Each VM is allocated memory equal to the peak demand of any PP in the tier over the entire month. An instantiation interval of 2 minutes is used based on current clouds. Page sharing is optimistically simulated by reducing the parent VM's memory by a fixed tunable percentage throughout the whole simulation. Extra workers do not share pages due to their transient nature.

Kaleidoscope: Similar to the Elastic Cloud with an instantiation interval of 5 seconds, based on performance reported in Section 7. A new clone's memory starts at 20% of its final size and grows dynamically for 40 seconds according to one of two memory growth profile's from Figure 10(a) - a large footprint clone corresponding to the Support, Ecommerce, and Banking workloads that grows to 90% of the full memory allocation, and a small footprint corresponding to the Httperf and OLAP workloads that grows to 40% of the full memory allocation. Finally, the final memory size on all VMs (parents and clones) are reduced by a fixed tunable percentage when simulating memory sharing. To account for missed sharing opportunities in the transient clones, we reduce sharing percentage by 47% for clones (according to Table 2). We do not take into account the improvements in sharing overhead caused by Kaleidoscope's state coloring.

Figure 11(a) shows the substantial impact of Kaleidoscope on QoS by plotting cumulative unmet demand across all customers over the entire one-month period. We used $T_L = T_H - 20\%$ and interpolated the demand data to 5 second intervals. The plot shows that QoS degrades exponentially with the slower VM creation time of the Elastic Cloud and higher CPU thresholds. In practice, VM cold booting used in the Elastic Cloud will result in even higher QoS violations for many applications due to the performance degradation, caused by cold caches, that is ignored in these results. In comparison, Kaleidoscope with $T_H = 90\%$ has over three orders of magnitude less unmet demand than the Elastic Cloud. Thus, slow VM creation imposes the adoption of more conservative VM creation and deletion thresholds, and is, in effect, the modus operandi in today's IaaS operations.

Higher CPU thresholds lead to better utilization of resources and increased IaaS efficiency. Figure 11(b) shows the number of CPU-hours actually allocated to all customers over the one-month period, as a function of the CPU threshold. CPU hours are counted (with a five-second granularity) from the moment an allocation decision is made, and thus are independent from the instantiation latency. The corresponding Warm Static allocation, whose CPU-hour count is independent of creation thresholds, is over an order of magnitude higher (4.2 million CPU-hours) and is not shown in the figure. By combining the results from Figure 11(b) with Figure 11(a) we see that Kaleidoscope with a $T_H = 90\%$ outperforms current Elastic Clouds with $T_H = 50\%$ by resulting in a 98% reduction in the number of seconds of unmet demand, while still requiring 26% fewer resources to be purchased by cloud users.

Finally, Figure 11(c) shows the number of physical machines required as a function of memory page sharing. We compare the Elastic Cloud and Kaleidoscope scenarios. For Elastic Cloud, we consider two creation thresholds, an aggressive one that minimizes resource usage (90%), and a conservative one that minimizes unmet demand (60%). For Kaleidoscope we only consider the aggressive 90% threshold, and compare three different memory growth profiles for clones: Support (growth of up to 90% of the footprint), OLAP (growth of up to 40%), and a 50-50 Mix with customers randomly assigned either profile. For both Elastic Cloud and Kaleidoscope we also factor in an overall



(a) Unmet demand – a proxy for QoS – as a function of VM creation and destruction thresholds. Log scale, higher is worse

(b) VM CPU hours billed as a function of VM creation thresholds. Note that this is independent of instantiation latency.

(c) Infrastructure use, as a function of memory consolidation ratio. For Kaleidoscope we consider three fractional allocation profiles.

Figure 11. Simulations using AT&T Hosting data show that Kaleidoscope improves cloud QoS, resource use, and efficiency.

page sharing success percentage. Even with identical CPU thresholds, Kaleidoscope significantly reduces the number of physical machines needed by the IaaS provider by 5% to 30%. If the VM boot techniques used in the Elastic Cloud setup require a conservative threshold of 60%, then even higher infrastructure reductions of up to 50% are possible.

10. Related Work

We propose VM state coloring as another way of bridging the semantic gap between VM management and OS knowledge, and is very different from 'memory coloring', which has been used in the past for techniques improving the performance of processor memory caches. The problem of the semantic gap in virtualized environments was first formulated by Chen and Noble [Chen 2001]. Patagonix [Litty 2008] and Antfarm [Jones 2006a] perform a form of architecture-based semantic gap bridging, using x86 page table knowledge to identify user-space processes inside a VM. Self-migration [Hansen 2004], is an approach to migration that demands tight collaboration between the host VMM and guest VM, pervasively bridging the semantic gap.

Geiger [Jones 2006b] targets semantic gap issues related to the OS page cache, and presents an OS-independent alternative to our page cache identification mechanism. Geiger, however, requires the use of approximation heuristics to detect when a page in the page cache has been given a different use. It also needs to reverse-engineer the underlying filesystem to detect block device transactions pertaining to the journal or other non-file structures, and to group guest pages according to their backing file.

Potemkin [Vrable 2005] implements a different form of VM cloning: Potemkin clones are short-lived lightweight VMs residing in the same host as the parent (or template) and sharing memory via copy-on-write. The canonical work in memory management of VMs is Waldspurger's [Waldspurger 2002]. Conceptually, Difference Engine [Gupta 2008] extends these ideas by adding sub-page sharing. Both systems perform repeated cycles of brute-force fingerprinting of the entire host memory to achieve memory deduplication. Satori [Milosz 2009] is similar to our work in that it uses an efficient source of VM introspection, virtual disk DMA operations, to guide the sharing mechanism.

To the best of our knowledge, we are the first to evaluate the use of VM cloning to dynamically scale servers and preserve QoS during spikes in load. Prior related work has focused on optimizing application QoS within a static VM allocation. Approaches include the utilization of VM migration to relieve datacenter hot spots [Wood 2006], workload management and admission control to optimize QoS and resource use [Elnikety 2004, Urgaonkar 2008a], and allowing applications to barter resources [Norris 2004].

We close by highlighting related work in the broad area of dynamic resource provisioning. Typically, dynamic VM provisioning is achieved with copy-and-boot techniques [Murphy 2009, Urgaonkar 2008b]. Another technique is to keep a pool of pre-configured machines on standby, and to bring these generic hot spares to bear as required [Fox 1997]. These are all examples of approaches that tie up computing resources in reserve, or present prolonged instantiation latencies leading in many cases to subpar performance.

11. Conclusions

In this paper we have introduced the notion of cloud micro-elasticity, in which servers react to load by swiftly spawning transient, short-lived cloned VM's with warm application caches and a tightly adjusted fractional memory footprint. Micro-elasticity is achieved by tailoring propagation and sharing policies to the different types of memory in a VM, a technique we call VM state coloring. VM state coloring can spawn stateful clones with warm application caches at a fraction of the cost (and footprint) of state-of-the-art techniques. By simulating cloud micro-elasticity on traces collected from AT&T's multi-tenant hosting environment, we obtain reductions in infrastructure use of roughly 30%, which benefit both providers and users.

There are two important paths for future work. First, the consistency assumptions governing some multi-tiered applications might be subverted by the impromptu addition

of cloned VMs. This has not been the case for our Web and OLAP workloads; nonetheless, we need to further explore the suitability of our sanitization mechanisms for other workloads. Second, the principles of VM state coloring can be very useful in WAN VM migration; improvements on the efficiency of content-addressing, prefetching, and even fetch avoidance are all valuable optimizations to migration.

Acknowledgments

We thank Michael Mior for his input throughout this project. We thank Philip Patchin, John Wilkes, Angela Demke-Brown, Bianca Schroeder, and Mary Fernandez for their feedback prior to submission. Eleftherios Koutsofios enabled access to the data used in the simulation study. We thank the anonymous Eurosys reviewers, and our shepherd Andreas Haeberlen for helping improve the quality of the final version of this paper. This research was partially supported by the National Science and Engineering Research Council of Canada (NSERC) under grant number 261545-3, and a hardware donation from Oracle Inc.

References

[Amazon a] Amazon. Auto Scaling. `http://aws.amazon.com/autoscaling/`.

[Amazon b] Amazon. Elastic Load Balancing. `http://aws.amazon.com/elasticloadbalancing/`.

[Amazon c] Amazon. SimpleDB. `http://aws.amazon.com/simpledb/`.

[Chen 2001] P. Chen and B. Noble. When Virtual is Better Than Real. In *Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Elmau/Oberbayern, Germany, May 2001.

[Elnikety 2004] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proc. 13th WWW*, pages 276–286, New York City, NY, May 2004.

[Fox 1997] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based Scalable Network Services. In *Proc. 16th SOSP*, pages 78–91, Saint Malo, France, October 1997.

[Gupta 2008] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proc. 8th OSDI*, San Diego, CA, December 2008.

[Hansen 2004] J. A. Hansen and E. Jul. Self-migration of Operating Systems. In *11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[Hyperic] Hyperic. CloudStatus. `cloudstatus.com`.

[Jones 2006a] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proc. Usenix ATC*, Boston, MA, June 2006.

[Jones 2006b] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *Proc. 12th ASPLOS*, San Jose, CA, October 2006.

[Lagar-Cavilla 2009] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proc. 4th EuroSys*, pages 1–12, Nuremberg, Germany, April 2009.

[Lakshman 2009] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. In *Proc. 3rd LADIS*, Big Sky, MT, October 2009.

[Litty 2008] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proc. 17th Usenix Security*, San Jose, CA, July 2008.

[Milosz 2009] G. Milosz, D. Murray, S. Hand, and M. Fetterman. Satori: Enlightened Page Sharing. In *Proc. Usenix ATC*, San Diego, CA, July 2009.

[Mosberger 1998] D. Mosberger and T. Jin. httperf—a Tool for Measuring Web Server Performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[Murphy 2009] M. A. Murphy, B. Kagey, M. Fenn, and S. Goasguen. Dynamic Provisioning of Virtual Organization Clusters. In *Proc. 9th CCGRID*, pages 364–371, Shanghai, China, 2009.

[Norris 2004] J. Norris, K. Coleman, A. Fox, and G. Candea. OnCall: Defeating Spikes with a Free-Market Application Cluster. In *Proc. 1st ICAC*, pages 198–205, Washington, DC, USA, 2004.

[Hsieh 2004] P. Hsieh. Super Fast Hash function, 2004. `http://www.azillionmonkeys.com/qed/hash.html`.

[RightScale] RightScale. More Servers, Bigger Servers, Longer Servers, and 10x of That. `http://blog.rightscale.com/2010/08/04/more-bigger-longer-servers-10x/`.

[Satyanarayanan 2005] M. Satyanarayanan, M. Kozuch, C. Helfrich, and D. O'Hallaron. Towards Seamless Mobility on Pervasive Hardware. *Pervasive and Mobile Computing*, 1(2), 2005.

[SPECweb 2005] SPECweb. Standard Performance Evaluation Corp., 2005. `http://www.spec.org/web2005/`.

[TPC-H] TPC-H. Transaction Processing Performance Council. `http://www.tpc.org/tpch/`.

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

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

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

[Waldspurger 2002] C. A. Waldspurger. Memory Resource Management in VMWare ESX Server. In *Proc. 5th OSDI*, Boston, MA, 2002.

[Wood 2006] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and Gray-box Resource Management for Virtual Machines. In *Proc. 2nd NSDI*, Boston, MA, 2006.