INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Shrinker: Efficient Wide-Area Live Virtual Machine Migration using Distributed Content-Based Addressing

Pierre Riteau — Christine Morin — Thierry Priol

N° 7198

Février 2010

Distributed and High Performance Computing

ISSN 0249-6399 ISRN INRIA/RR--7198--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Shrinker: Efficient Wide-Area Live Virtual Machine Migration using Distributed Content-Based Addressing

Pierre Riteau* , Christine Morin† , Thierry Priol†

Theme : Distributed and High Performance Computing
Équipe-Projet Myriads

Rapport de recherche n° 7198 — Février 2010 — 21 pages

Abstract: Live virtual machine migration is a powerful tool offered by virtualization technologies. However, current implementations restrict its usage to local area networks. Extensions to wide-area networks have been proposed with systems for transferring disk state and transparently migrating network connections. Still, wide-area live migration remains expensive to use because of the bandwidth bottleneck, especially when considering migrations of large clusters of virtual machines rather than single instances. Previous work has shown that VMs running identical or similar operating systems have a significant portion of their memory containing the same data. In this paper, we introduce Shrinker, a live virtual machine migration system leveraging this common data to improve migration efficiency between data centers interconnected by wide-area networks. Shrinker uses distributed content-based addressing to detect memory pages of the migrated VM that are already available on the destination site, removing the need to transfer these pages over a wide-area link. We have implemented Shrinker in the KVM hypervisor and present performance evaluation in a distributed environment. We show that it is able to reduce wide-area bandwidth usage by 30 to 40% and migration time by 20%.

Key-words: Virtualization, Live Migration, Wide-Area Networks, Cloud Computing

* Université de Rennes 1, IRISA, Rennes, France – Pierre.Riteau@irisa.fr
† INRIA Rennes - Bretagne Atlantique, Rennes, France – firstname.lastname@inria.fr

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Shrinker: Migration à chaud efficace de machines virtuelles dans des réseaux étendus

Résumé : La migration à chaud de machines virtuelles est un outil puissant offert par les technologies de virtualisation. Cependant, les mises en œuvres actuelles limitent son utilisation aux réseaux locaux. Des extensions aux réseaux étendus ont été proposées avec des systèmes permettant de transférer l'état du support de stockage virtuel et de migrer de façon transparente les connexions réseaux. Toutefois, la migration à chaud dans les réseaux étendus reste coûteuse à cause de la limitation de la bande passante, en particulier dans le cas d'une migration d'une large grappe de machines virtuelles plutôt qu'une unique instance. Il a été montré que des machines virtuelles exécutant des systèmes d'exploitation identiques ou similaires ont une part non négligeable de leur mémoire contenant des données identiques. Dans cet article, nous introduisons Shrinker, un système de migration à chaud de machine virtuelle tirant parti de ces données identiques afin d'améliorer l'efficacité de la migration entre des centres de données connectés par des réseaux étendus. Shrinker utilise une table de hachage distribuée fondée sur l'adressage par contenu pour détecter les pages mémoires de la machine migrée qui sont disponibles sur le site destination, évitant ainsi de les transférer sur le réseau étendu. Nous avons mis en œuvre Shrinker dans l'hyperviseur KVM et nous présentons une évaluation de ses performances dans un environnement distribué. Nous montrons qu'il est capable de réduire l'utilisation du réseau étendu de 30 à 40% et le temps de migration par 20%.

Mots-clés : Virtualisation, Migration à chaud, Réseaux étendus, Cloud Computing

1 Introduction

The complete encapsulation of full execution environments (applications combined together with their underlying OS) in virtual machines (VMs) allowed the development of live virtual machine migration technologies [7, 23]. These mechanisms relocate the memory and virtual device state of a VM from one physical machine to another, with no noticeable downtime of the VM. This offers interesting advantages for data center management, including:

Load balancing: VMs can be dynamically migrated depending on workload, offering more efficient usage of computing resources.

Power efficiency: VMs with low workload can be consolidated to fewer physical machines, allowing to power off resources.

Transparent infrastructure maintenance: Before machines are brought down for maintenance, administrators can relocate VMs to other nodes without any noticeable interruption of service for users.

Current implementations of live virtual machine migration in popular hypervisors, such as Xen [7] or VMware [23], target single data center scenarios: VMs can only be migrated between nodes of a single site. This restriction comes from two limitations of these implementations. First, they depend on shared storage to avoid transferring persistent state of VMs, which can be expensive (in the order of several gigabytes). Shared storage is usually not accessible across several data centers. Second, they require that the destination node be in the same IP network than the source node. This allows migrated VMs to leave their IP address unmodified, keeping TCP connexions opened after migration to a new host machine.

However, live migration of virtual machines in wide-area networks (WANs) would offer interesting features to both administrators and users of cloud computing environments. Administrators of multiple geographically distributed data centers would be able to get the same benefits brought by live migration in local networks. It would allow them to load balance workload between several data centers, and offload VMs to another site whenever a site-wide maintenance is occurring. Users could also benefit from wide-area live migration. Users with access to *private clouds* (private computing infrastructures managed using a cloud computing stack) could seamlessly migrate VMs between private clouds and public clouds depending on resource availability. Even though provisioning fees in clouds are currently fixed, this could change in the future to a model where pricing can be variable, e.g. depending on infrastructure load. Wide-area live migration could enable users to leverage competition between cloud providers by migrating from one to another.

Several research works focused on enabling wide-area live migration by solving the two aforementioned issues. They proposed systems to transfer disk state [5, 19, 13] and to transparently migrate network connections [5, 11]. However, the large size of virtual machines still remains an important issue for wide-area live migration, especially when considering migrations of large clusters of virtual machines rather than single instances.

Previous work has shown that VMs running identical or similar operating systems have a significant portion of their memory containing the same

data [30, 9]. This is caused by VMs having the same versions of programs, shared libraries or kernels loaded in memory, or common files loaded in buffer cache. In this paper, we introduce Shrinker, a live virtual machine migration system leveraging this common data to improve migration efficiency between data centers interconnected over wide-area networks. Shrinker uses distributed content-based addressing to detect memory pages of the migrated VM that are already available on the destination site, removing the need to transfer these pages over a wide-area link. We present the design and implementation of Shrinker, which is developed as a modification of KVM, the Linux Kernel Virtual Machine [15].

This paper is organized as follows. In Section 2, we cover the architecture of Shrinker. Then, in Section 3, we describe our implementation in the KVM hypervisor. In Section 4, we present our experiment methodology and analyze the results. In Section 5, we cover related work. Finally, we conclude and give work perspectives for the future in Section 6.

2 Architecture

In this section we present the design of our system. To improve live migration efficiency, Shrinker detects memory pages of the migrated VM that are already available in VMs running on the destination site. First, we present the general idea behind Shrinker. Second, we describe how duplicate memory is detected, using content-based addressing combined with a site-wide distributed hash table at the target site. Then, we present how this distributed hash table is populated by periodic memory indexing of VMs. Finally, we cover security issues generated by our utilization of content-based addressing.

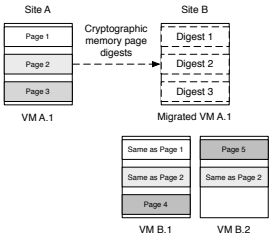
We consider live migration of virtual machines between data centers interconnected by wide-area networks. A migration occurs from a *source* or *original* node to a *target* or *destination* node. The original node is located in a *source* or *original* site while the target node is located in a different *target* or *destination* site.

To reduce bandwidth usage of live migration, Shrinker takes advantage of memory pages in the migrated VM that are present in the memory of VMs executing on the destination site. This can be the case for several reasons:

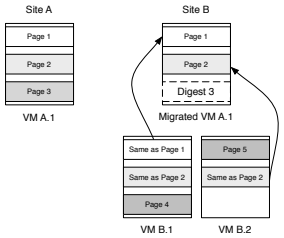
- VMs on the destination site are executing the same versions of programs, shared libraries or kernels as the migrated VM, and thus have identical pages of code loaded in memory. This is usually the case when VMs are created using the same install media or the base image such as a common Amazon EC2 Machine Image (AMI).
- Common file system data is loaded in buffer cache.
- A program running on the migrated VM and on VMs of the destination site uses identical data aligned to page boundaries.

When such pages are identified, they are transferred using the local network of the destination site instead of the wide-area network. In order to identify these pages, Shrinker leverages content-based addressing (also called content-based hashing) using cryptographic hash functions. Cryptographic hash functions map a block of data of almost arbitrary length to a hash value (or digest) of

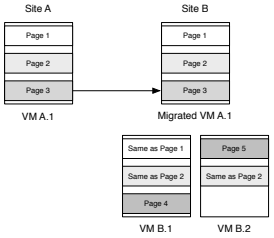
fixed size (usually smaller than the original data). These functions differ from ordinary hash functions in that they are designed to render practically infeasible to find the original block of data from a hash value, modify a block of data while keeping the same hash value, and find two different blocks of data with the same hash value. Memory page digests use only a fraction of the VM's memory size. For example, with the SHA-1 [22] cryptographic hash function which maps a block of data to a 160-bit (20 bytes) hash value, digest use only about 0.5% of a x86 VM's memory size (since the x86 architecture has 4 KB memory pages).



(a) Transfer of cryptographic digests



(b) Lookup of existing data in the target site



(c) Transfer of missing pages from the source

Figure 1: Overview of Shrinker's migration protocol: initially, only cryptographic digests of each page are sent to the destination site (site B). Pages that are already present in VMs of site B are transferred over LAN. Remaining pages are requested to the source node over WAN.

Figure 1 presents a high-level overview of Shrinker's migration protocol. Instead of sending all memory pages from the source node to the target node, Shrinker's migration protocol initially only transfers each page's hash value (Figure 1(a)). Each digest received by the target machine identifies a memory page. The target machine uses digests to look up pages in the destination site. If pages are found on the destination site, they are transferred using the local network instead of the wide-area network (Figure 1(b)). Otherwise, they are requested to the original node over WAN (Figure 1(c))

2.1 Distributed Content-Based Addressing

To be able to fetch memory pages from the memory of other VMs in the site, the destination node needs to discover which VMs in the network have a copy of a specific page. Since these other VMs are running and their memory content

changes over time, this information has to be discovered dynamically. To solve this problem in Shrinker, we use two subsystems. The first is a site-wide distributed hash table allowing to locate nodes having a copy of a given page using its hash value. The second is a periodic memory indexer added to hypervisors to populate the distributed hash table.

2.1.1 Site-Wide Distributed Hash Table

To look up memory pages by their hash values, Shrinker leverages a site-wide distributed hash table (DHT) built on a peer-to-peer (P2P) network. All VM hosts in the site participate in the P2P network. Similarly to classic DHTs such as Chord [27], the P2P network forms a ring. Each node joining the network is identified by the hash of its IP address and port, which defines its position in the ring. A node is responsible for indexing memory pages of the site with hash values that immediately follow its ID.

However, contrarily to Chord and similar DHTs, Shrinker's DHT uses a $O(1)$ design. Participants of the P2P network are aware of all other nodes. We choose this because the primary requirement for Shrinker's DHT is performance. As a matter of fact, consider a 512 MB VM being migrated. This VM is identified by 131,072 page hashes, each requiring a lookup to determine if and where the corresponding page is available on the target site. The very high number of lookups impose stringent requirements on the DHT latency and throughput. Since Shrinker targets data centers that are relatively stable environments, resistance to high churn is not the main requirement for our DHT. Moreover, scalable $O(1)$ peer-to-peer overlays have been presented in the literature [8]. Finally, $O(n)$ DHTs are designed to scale to millions of participating nodes. However, data centers with millions of physical machines are not common.

Figure 2 presents an example of DHT used by Shrinker. For simplicity's sake, hash values in this figure are only two hexadecimal digits in length. Nodes are ordered on a ring from value 00 to value FF. In our example, three nodes are participating in the DHT: nodes 20, 70 and 95. Node 20 is hosting a VM composed of two pages with hash values 78 and $A5$. Node 70 hosts a VM with three pages: $0D$, $A5$ and $C3$. Node 95 is not hosting any running VM but is the target node of the migration of a VM with three pages: page 78, page $A5$ and page $F4$. Since nodes are responsible for the area of the ring that follows them, 95 is responsible for indexing page $0D$. It knows that page $0D$ is available from node 70. Similarly, node 70 indexes pages 78, $A5$ and $C3$.

During the live migration, node 95 receives a stream of memory page digests. It uses them to look up pages in the DHT. The responsible for page 78 is node 70, so node 95 contacts it to look up the page. When 70 receives the request, it forwards it to the node hosting a copy of page 78, node 20. Node 20 answers the request by sending the content of page 78 to node 95. Next, node 95 looks up page $A5$, which is also indexed by node 70. Page $A5$ is present on nodes 20 and 70, so node 70 can either send directly the page to node 95, or ask node 20 to send it. Finally, node 95 looks up page $F4$, for which it is responsible, and discovers that it is not available. Node 95 thus requests the original node to send the content of page $F4$. In all cases where a page is coming from local peers, the target node checks that the page content is consistent with the corresponding hash value, to avoid intentional injection of corrupted data.

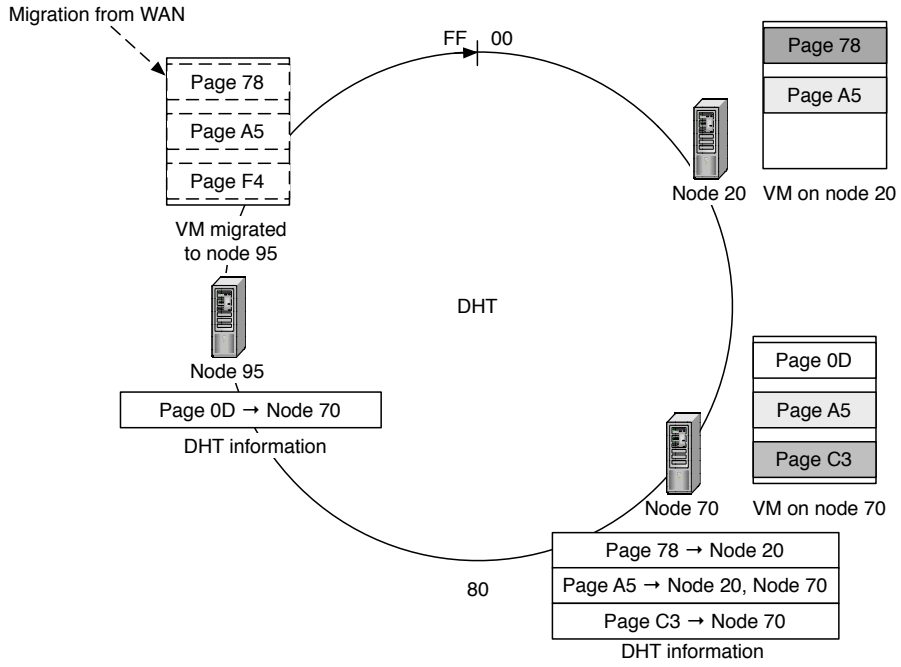


Figure 2: Shrinker's Distributed Hash Table

2.1.2 Periodic Memory Indexing

In order to populate the DHT, hypervisors on VM hosts are modified to periodically index the memory of their VM guests. They compute the hash value of each page and communicate them to responsible nodes in the DHT to inform them that page copies are available. When a memory page has changed, the hosting node also contacts the node responsible for the old version of the page to inform it that the copy is not accessible anymore, to avoid keeping outdated information in the DHT.

The most basic mode of operation for memory indexing would be to periodically scan the entire memory. However, hypervisors generally allow to track memory changes in VMs. This is actually required to efficiently use the *pre-copy* live migration technique [28] which iteratively copies memory on the target node as it is modified on the source node. Using this tracking system, hypervisors can restrict their indexing to memory pages that have been modified since the last index run.

To avoid indexing memory pages that are going to be modified in the near future, Shrinker implements page idleness detection, similarly to [9]. Instead of scanning all pages that were modified since the last run, hypervisors keep track of the number of consecutive rounds during which a page has not been modified. The page is indexed only when this counter has exceeded a threshold. The memory overhead of this mechanism is insignificant: using one byte per page to keep track of 256 possible rounds requires only 0.025% of the VM's memory.

It must be stressed that strong consistency is not required between the index of pages in the DHT and the actual pages residing on VM hosts. If a page indexed by the DHT is not available anymore on the corresponding VM host, the protocol simply falls back to a transfer from the original node.

2.2 Security Considerations

Since the number of unique memory pages (2^{4096}) is bigger than the number of possible hash values (2^{160} for SHA-1), the use of cryptographic hash functions opens the door to collisions: it is possible that different memory pages involved in a live migration (on the sender side and/or in the DHT) map to the same hash value. In this case, a node querying the DHT for the page identified by this hash value may receive the wrong page. However, the properties offered by cryptographic hash functions allow us to use hash values as unique identifiers of memory pages with a high confidence. The probability p of one or more collisions occurring is bounded by the following equation, where n is the number of objects in the system and b the number of bits of hash values:

$$p \leq \frac{n(n-1)}{2} \times \frac{1}{2^b} \tag{1}$$

If we consider a distributed system with 1 exabyte (2^{60} bytes) of 4 KB memory pages indexed by Shrinker using the SHA-1 hash function, the collision probability is around 10^{-20} [24]. This is considered to be much less than probability of data corruption undetectable by ECC memory for example.

However, this is the probability of an accidental collision. Although the theoretical number of operations to find a collision is approximately $2^{\frac{n}{2}}$ (birthday attack), attackers can exploit weaknesses in the hash function design to find collisions more easily. For example, researchers have shown attacks against SHA-1 that decrease the number of operations from 2^{80} to 2^{63}.

In the context of Shrinker, an attacker with access to VMs participating in the DHT (which, in public cloud computing environments, would require only to provision VMs for a small fee) could inject colliding pages in the DHT. These pages could then be used by a live migration process to populate the memory of the migrated VM, injecting wrong data into it. This issue can be mitigated by migrating to stronger cryptographic hash functions, such as SHA-224, SHA-256, SHA-384, and SHA-512. The main downside is that they are more expensive to compute. Another solution is to restrict participation to the DHT. For example, in a cloud environment, each group of trusted users could have its own DHT. This may however decrease Shrinker's ability to find existing data since less memory pages would be shared.

3 Implementation

We implemented Shrinker in the KVM hypervisor [15]. Unlike Xen [2], KVM requires virtualization capabilities of recent x86 processors (AMD-V or Intel VT-x) to run guest VMs. KVM is built from two separated components. The first part runs in the kernel, and consists of two loadable modules: one is generic while the other is processor-dependent. The second part is a modified version

of QEMU [3] running in user space to provide all other virtualization features, such as device emulation or live migration.

Our initial implementation of Shrinker only modifies the user space component of KVM version 0.11 and is about 2,000 lines of C code. The only additional dependency is the OpenSSL library, which is used to compute hash values. Figure 3 presents a high-level overview of our implementation and the relations between the subsystems. The first subsystem implements periodic memory indexing (c.f. Section 2.1.2). This subsystem communicates with the kernel part of KVM. The second one is a simple $O(1)$ distributed hash table to allow indexing and requesting pages with peer hypervisors (c.f. Section 2.1.1). Finally, the live migration code, which is already present in KVM, is slightly modified to allow transferring page hashes between the original and the target node, and calling the other subsystems when needed.

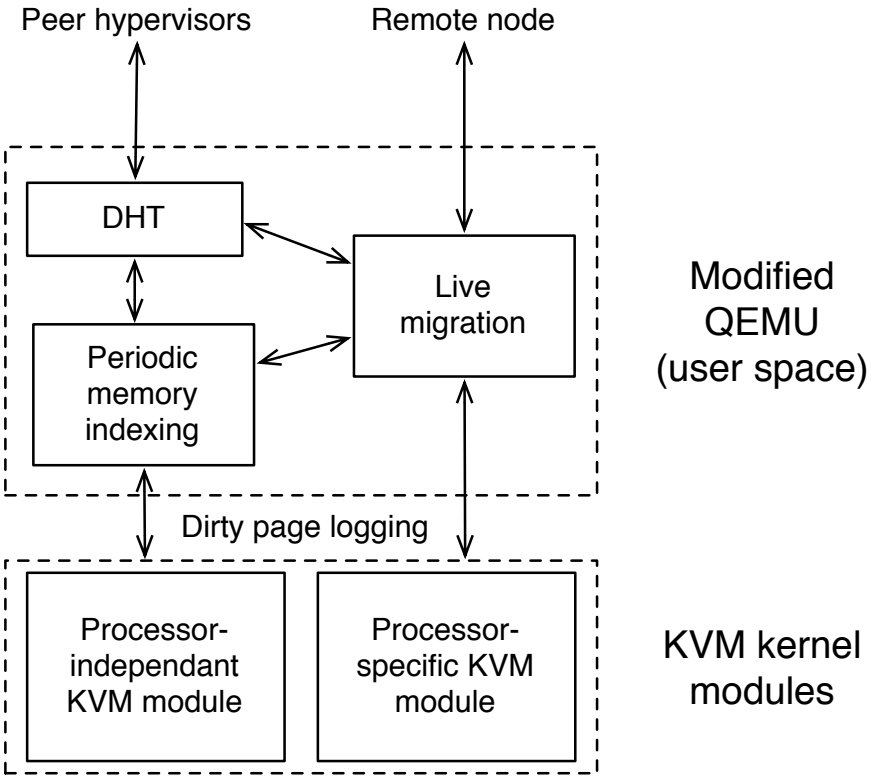


Figure 3: Overview of Shrinker's subsystems and their relations

3.1 Periodic Memory Indexing

We implemented periodic memory indexing in KVM using features built for the existing live migration infrastructure. Shrinker uses the KVM kernel interface to keep track of page modifications (*dirty page logging*). The kernel module marks VM pages as read-only. Whenever the VM tries to write to a page, the

operation is trapped by the kernel and the KVM module marks the page as dirty. The page then stays in a writable state until the user space component resets the dirty state. This means that writes to a page can only be trapped once between two index runs, which is much better performance-wise than logging all write attempts.

When periodic memory indexing is run, Shrinker requests the kernel for a bitmap of the VM's pages state, to check which ones are dirty. When a page has been unmodified for a number of indexing runs (both the frequency of the indexing and the idleness threshold can be tuned), it is considered idle enough to be indexed in the DHT. In this case, the hypervisor needs to be able to answer page requests from a peer and maintain the DHT when the page is modified. For this, we keep track of idle pages using two data structures, presented in Figure 4 (again, for simplicity, hash values are two hexadecimal digits long). First, a hash table allows to efficiently locate a page's content from its digest, whenever a peer requests it. Note that identical pages in memory map to the same digest. This means the hash table stores a list of addresses to local pages. In Figure 4, three idle pages (two uniques) are registered. Page 0 and 3 are identical and map to the same hash value AB, while page 1 maps to 54. When a request for a page arrives from a peer, this hash table is used to find one of the corresponding memory page and send its data as the answer. We also use a table to keep track of the indexed hash value of every memory page. This is required because the user space part of the hypervisor is not notified immediately when a page is modified by a VM, and needs to track the old digest to be able to unregister the page from the local hash table and from the DHT. For example, if page 1 is modified, at the next index run the hypervisor will remove the entry 54 from the local hash table, and contact the node responsible for 54 to remove the corresponding reference from the DHT.

Since dirty page logging happens only in the kernel, the indexing state in user space can be out of date compared to the real content of VM pages. This is not considered a significant issue: the DHT works in a best-effort mode. In our current implementation of Shrinker, hypervisors do not check whether the page is still valid before sending it as an answer to a page request. An alternative would be to do a copy of the page (to be sure that it is not modified by the VM) and compute its hash value to verify whether we are sending valid data. Choosing one solution over another depends on what is the most important factor: CPU usage or local network utilization. In all cases, when a page indexed in the DHT cannot be found or does not match the answer of a peer, Shrinker transfers the page from the original node.

3.2 Live Migration Protocol

The live migration protocol implemented in KVM is based on pre-copy, like the implementations in the Xen [7] and VMware [23] hypervisors. The protocol simply streams memory and device state as a sequence of bytes to the target node. When memory is sent, each piece of data is preceded by an address combined with a flag. When a page is sent, the address is its address in the VM's physical memory and the flag indicates the page type (normal or compressed). The protocol implements one optimization: before a page is sent, all bytes of the page are compared. If all bytes are identical, the page type is marked as compressed and only the first byte is sent. This allows to efficiently send pages

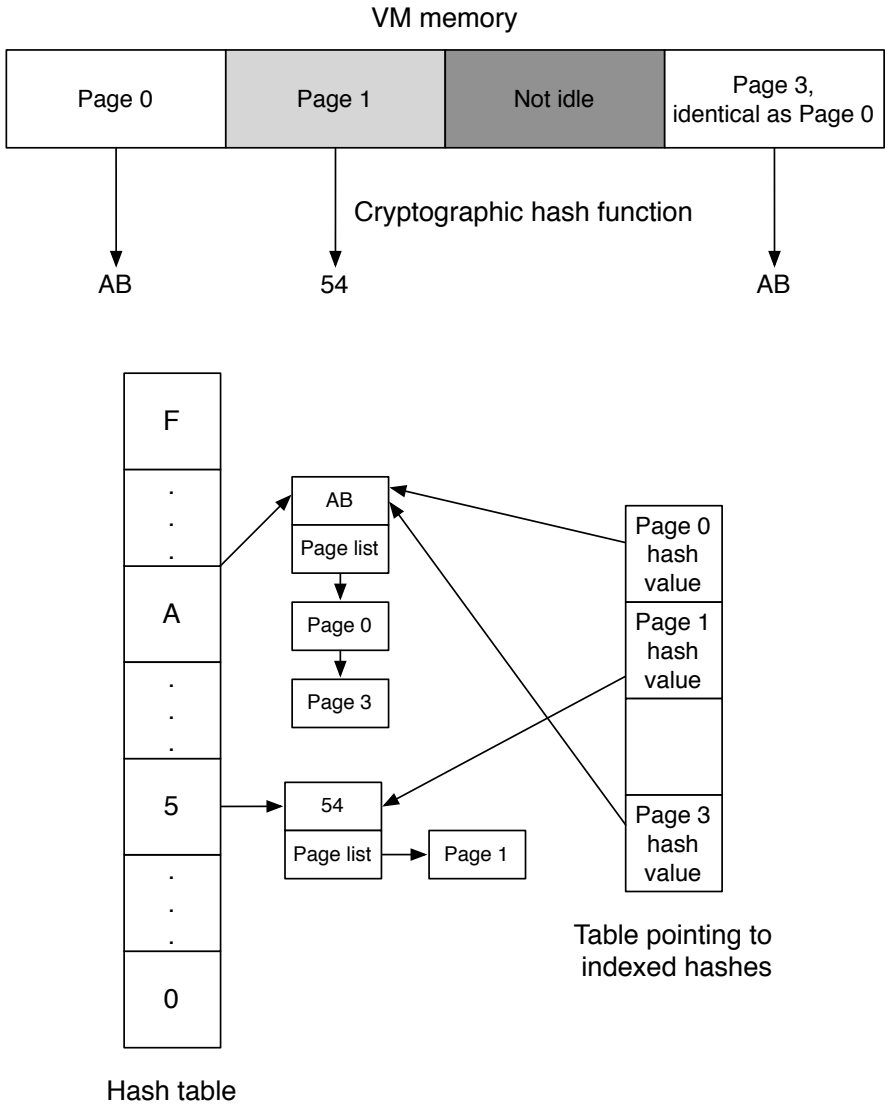


Figure 4: Data structures used by periodic memory indexing

composed of only zero bytes. Our implementation adds a new type of page to the flag: *hashed page*. In this case, the data following the address is a hash value instead of the complete page content. When this type of data is received, it triggers a DHT lookup on the destination node. It should be noted that Shrinker still uses the compressed flag. This means zeroed pages are *not* looked up in the DHT.

3.3 Distributed Content-Based Addressing

Our current implementation of a DHT and P2P overlay is very simple. Periodically, hypervisors send an announcement to a predefined multicast IP address. All hypervisors also listen for announcements. This allows them to create a list of network peers. When a peer has not sent any announcement for some time, it is removed from the list. All requests and replies in the DHT are done using UDP. We avoided TCP because scaling to a large number of nodes would require to set up a large number of connexions: since nodes are uniformly distributed over the ring and hash values are usually also uniformly distributed, looking up all pages of a migrated VM would require to set up connexions with many peers in the network. There are many ways in which this architecture could be improved. However, the DHT and P2P overlay design is not the main contribution of this paper.

When a hash value is received on the target node, the node first searches the local hash table for the page. This allows to exploit intra-VM data commonality without requiring any DHT request. If the page cannot be found locally, the node determines which peer is responsible for the page. This is done locally, since all peers are aware of each other. The node then sends the hash value to the responsible node. It also creates a timeout in order to fall back to the original node if the answer never arrives. This could happen for several reasons: packet loss, crash of a peer, etc.

When the responsible node receives the hash value, it looks for it in a local index. This index is similar to the local hash table used by periodic memory indexing, except that it keeps track of peers instead of page addresses, using their IP addresses and ports. If the page cannot be found, the responsible node answers the request to inform the target node that the page is not available. In this case, the target node requests the page from the original node. If the page is found, the responsible node directly contacts one of the node hosting the page with the hash value and the IP information of the target node. This allows the node hosting the page to send it directly to the target node. To keep live migration efficient not only with regard to bandwidth but also time, these lookups are done in parallel with the incoming digest stream from the original node. The destination node keeps track of sent requests using a queue. When answers are received or when timeouts are triggered, the corresponding requests are removed from the queue. If the incoming stream is too fast and the queue becomes too big, the stream is throttled down in order for the queue to catch up.

4 Performance Evaluation

In this section we study performance of our Shrinker implementation. We evaluate two types of metrics. First, we measure its influence on WAN bandwidth utilization and migration time. Second, we study the performance overhead caused by periodic memory indexing. These experiments are performed in a real distributed environment. We use two sites of the Grid'5000 platform, Nancy and Rennes, which are both located in France and separated from more than 600 kilometers. The sites are interconnected using a 10 Gbit/s link. On the original site (Nancy), each physical node involved in the experiment is hosting a VM. On the destination site (Rennes), the same number of physical machines is used as target nodes for live migration. Additional nodes are running VMs on site B to populate the DHT before the migration.

Although persistent state migration and network transparency mechanism for wide-area live migration have been proposed, they are not available in the current implementation of KVM. Since Shrinker itself does not implement these techniques, we used two workarounds to enable inter-site live migration. First, each node on the original site has a local copy of the VM virtual disk. Target nodes on the destination site access the virtual disk on the original node using SSHFS. To allow VMs to communicate between each other even after migration, they are configured to use private IPs available on both sites. The VMs are configured to use bridged networking to allow seamless communication between each other. Each host also has an interface alias with a private IP in the VM subnet. This allows us to control VMs from a frontend machine using OpenSSH's ProxyCommand.

Physical nodes use Intel Xeon processors with frequencies ranging from 1.6 GHz to 2.5 GHz. All nodes have a minimum of 2 GB of RAM. Physical nodes and VMs use the *x86_64* port of Debian 5.0 (Lenny) as their operating system. VMs are configured to use 1024 MB of memory.

We use two suite of benchmarks for our performance evaluation. The NAS Parallel Benchmarks (NPB) [1] are derived from computational fluid dynamics applications. Dacapo [4] is a Java benchmarking suite which consists of a set of real world applications with *non-trivial memory loads*.

4.1 Live Migration Performance

To evaluate live migration performance, we migrate a cluster of VMs from site *A* to site *B*. Firstly, we run the `cg` benchmark from NPB in 64 VMs on the original site. On the remote site, we use 64 nodes as target nodes for the migration, and 16 additional VMs (running on other physical machines) run randomly chosen benchmarks from the Dacapo suite. Secondly, we run Dacapo in the migrated VMs and `cg` in the additional VMs. Figure 5 presents the migration time of the 64 VMs from site *A* to site *B* for the two experiments.

We observe that migration time is improved by 17% and 25% respectively. We also measure the average number of pages sent over the wide-area network. Figure 6 shows that Shrinker reduces the number of pages transferred over WAN by 37% for `cg` and 40% for Dacapo.

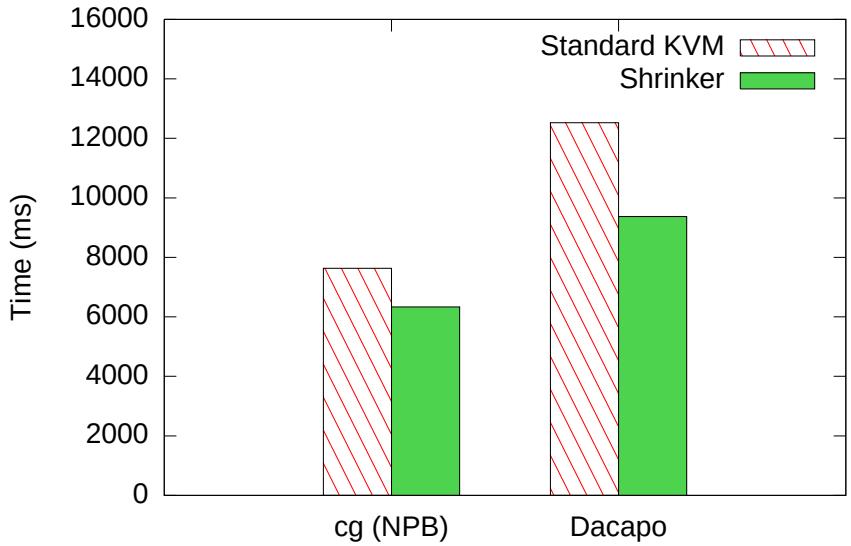


Figure 5: Average migration time of 64 VMs running `cg` (NPB) and random benchmarks from Dacapo

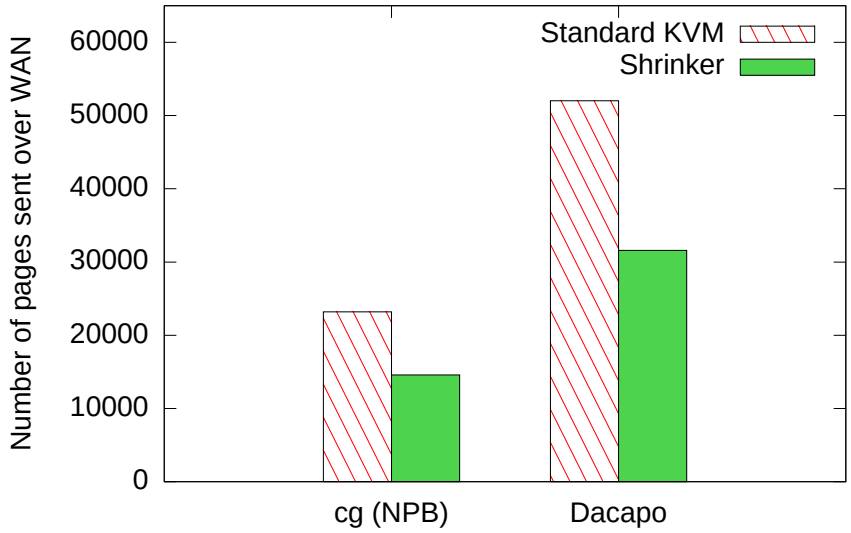


Figure 6: Average number of pages sent over the wide-area network for each migrated VM

4.2 Impact of Periodic Memory Indexing

To evaluate the performance impact of periodic memory indexing, we run the two benchmark suites with and without indexing running. We use an indexing period of 5 seconds. A memory page is registered in the DHT when it has been idle for 10 seconds (the page was not modified for two consecutive index runs).

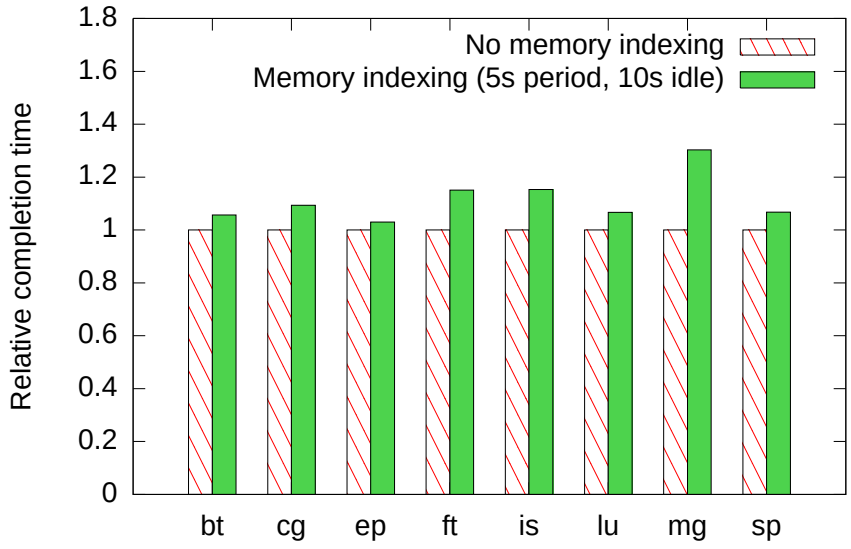


Figure 7: Relative performance of NAS Parallel Benchmarks with periodic memory indexing

Figure 7 presents the results from the NAS Parallel Benchmarks. With this suite, performance impact averages at 11.5% and ranges from 3% to 30%, depending on the application. Figure 8 presents the results from the Dacapo benchmarks. In this case, the overhead averages at 5.1% and ranges from 1% to 13%.

Periodic memory indexing adds a low to medium performance overhead, depending on the page dirtying rate of the application. However, since both the indexing period and the idleness threshold can be modified, the performance overhead could be reduced by tuning the indexing behavior according to the type of workload running in the virtual machine. Moreover, we think that our memory indexing implementation can be further optimized to offer better performance.

5 Related Work

To the best of our knowledge, Sapuntzakis et al. [26] were the first to introduce usage of content-based addressing to improve virtual machine migration efficiency. However, their work targeted offline migration (the VM is paused before being transferred), and only took advantage of data residing on the target node. Moreover, their case study at the time was migration of a paused VM over a DSL link during a work/home commute, and not the live migration of virtual

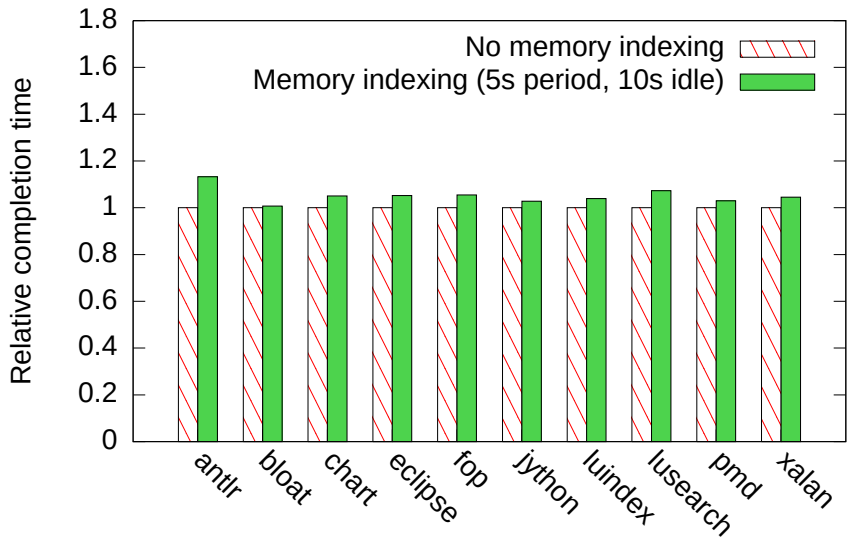


Figure 8: Relative performance of Dacapo benchmarks with periodic memory indexing

machines between cloud computing data centers. This kind of mechanism was also proposed by Tolia et al. [29]. Since then, few works exploited the concept of content-based addressing for virtual machine migration. Projects have looked at content-based addressing for storing VM images or snapshots [21, 25, 17], often in centralized repositories. They exploit data commonality in VM images to decrease storage consumption.

Live migration of virtual machines over WAN has been studied in the recent years. Bradford et al. [5] propose to use pre-copy and write throttling of local persistent state combined with dynamic DNS and tunneling to solve the shared storage and network transparency issues. Luo et al. [19] introduce a three-phase migration algorithm to transfer virtual storage. Hirofuchi et al. [13] combine on-demand fetching and background copying to relocate storage to the destination site after the VM has been migrated. Harney et al. [11] propose using Mobile IPv6 for network transparency of wide-area live migration.

Several studies have shown the benefits of using memory sharing between VMs collocated on the same host. This technique was originally proposed in Disco [6]. However, it required modification of the guest OS. VMware [30] implements content-based page sharing by hashing the pages, like we do in Shrinker. However, since VMs are hosted on the same physical machine, they can use a weaker hashing function and do a full comparison of pages in case of collision. This is of course not possible in our context since the pages are not on the same machine. Difference Engine [9] extends this mechanism by adding compression and patching of memory pages to significantly increase sharing possibilities compared to VMware. Satori [20] uses modifications of the guest OS to efficiently discover even short-lived sharing opportunities. Memory Buddies [31] uses fingerprinting to discover VMs with high sharing potential and to colocate them on the same host. It would be interesting to study how Shrinker can be combined

with mechanisms from Memory Buddies to select optimal nodes as targets of a wide-area live migration (e.g. by selecting as target a node hosting VMs with many memory pages in common with the migrated VM).

Hines et al. [12] propose using *post-copy* instead of *pre-copy* for live migration. The CPU state is first transferred on the target machine and is resumed. Memory pages are brought from the original node on faults, in addition to background copying. Shrinker could interface with this kind of mechanism: when a page is faulted on the destination machine, it would first be looked up on the local network before contacting the original node. Liu et al. [18] advocate the use of checkpointing/recovery and trace/replay instead of pre-copy. They show that it allows to reduce downtime, total migration time and total data transferred. However, the first phase of their algorithm transfers a whole checkpoint of VM memory. Shrinker would allow to optimize this phase, which is the most expensive regarding total data transferred. Jin et al. [14] propose using adaptive compression of migrated data: different compression algorithms are chosen depending on characteristics of memory pages. Shrinker could use these algorithms to compress pages sent to the destination node by the original node or peers in the target site. Hacking et al. [10] introduced two techniques to speed up live migration of virtual machines running large enterprise applications. The first is the addition of a warmup phase to the live migration process. Memory content is sent in the background to the destination node, but without committing to a migration. When live migration is performed, it can leverage the data already present on the destination node. This allows to reduce total live migration time and downtime during live migration. The second technique is delta compression based memory transfer. When memory pages are dirtied, the hypervisor sends a delta computed from the old and the new version instead of sending a full copy of the new page. This allows to reduce the bandwidth used by live migration when a large amount of memory is being dirtied. SnowFlock [16] is a system allowing to quickly instantiate VM forks on new hosts machines, using demand-paging and multicast distribution of data. It allows to very quickly create clones of VMs in a local network.

6 Conclusion and Future Work

In this paper, we present the design and implementation of Shrinker, a system which improves bandwidth efficiency of wide-area live virtual machine migration. It identifies memory pages already present on the remote site in order to transfer them using the local network instead of the wide-area network. Shrinker uses content-based addressing in a distributed hash table combined with periodic VM memory indexing. It is implemented as a modification of the KVM hypervisor. Our performance evaluation shows that it is able to reduce wide-area bandwidth usage by 30 to 40% and migration time by 20%.

In future work, we plan to add support for disk migration over wide-area networks using the same type of mechanism. However, since accessing disks is much slower than memory, we may require techniques to index only a subset of a disk image on each node. We will also study how our design can be adapted to volunteer computing systems without access to local networks with a large number of running VMs.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM.

[5] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179, New York, NY, USA, 2007. ACM.

[6] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[7] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *NSDI'05: Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[8] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 7–12, Berkeley, CA, USA, 2003. USENIX Association.

[9] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 309–322, 2008.

[10] Stuart Hacking and Benoît Hudzia. Improving the live migration process of large enterprise applications. In *VTDC '09: Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, pages 51–58, New York, NY, USA, 2009. ACM.

[11] Eric Harney, Sebastien Goasguen, Jim Martin, Mike Murphy, and Mike Westall. The Efficacy of Live Virtual Machine Migrations Over the Internet. In *VTDC '07: Proceedings of the 3rd international workshop on Virtualization technology in distributed computing*, pages 1–7, 2007.

[12] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60, New York, NY, USA, 2009. ACM.

[13] Takahiro Hirofuchi, Hirotaka Ogawa, Hidemoto Nakada, Satoshi Itoh, and Satoshi Sekiguchi. A Live Storage Migration Mechanism over WAN for Relocatable Virtual Machine Services over Clouds. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2009)*, pages 460–465, 2009.

[14] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive memory compression. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing (Cluster 2009)*, 2009.

[15] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Linux Symposium*, volume 1, pages 225–230, June 2007.

[16] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12, New York, NY, USA, 2009. ACM.

[17] Anthony Liguori and Eric Van Hensbergen. Experiences with content addressable storage and virtual disks. In *Proceedings of the First Workshop on I/O Virtualization (WIOV '08)*, 2008.

[18] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 101–110, New York, NY, USA, 2009. ACM.

[19] Yingwei Luo, Binbin Zhang, Xiaolin Wang, Zhenlin Wang, Yifeng Sun, and Haogang Chen. Live and incremental whole-system migration of virtual machines using block-bitmap. In *2008 IEEE International Conference on Cluster Computing*, pages 99–106, 2008.

[20] G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 USENIX Annual Technical Conference*, 2009.

[21] Partho Nath, Michael A. Kozuch, David R. O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 2006. USENIX Association.

[22] National Institute of Standards and Technology. Secure Hash Standard, April 1995.

[23] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 391–394, Berkeley, CA, USA, 2005. USENIX Association.

[24] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[25] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 143–156, Berkeley, CA, USA, 2008. USENIX Association.

[26] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 377–390, New York, NY, USA, 2002. ACM.

[27] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[28] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 2–12, New York, NY, USA, 1985. ACM.

[29] Niraj Tolia, Thomas Bressoud, Michael Kozuch, and Mahadev Satyanarayanan. Using Content Addressing to Transfer Virtual Machine State. Technical report, Intel Corporation, 2002.

[30] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 181–194, New York, NY, USA, 2002. ACM.

[31] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark Corner. Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*, pages 31–40, 2009.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399