

VeCycle: Recycling VM Checkpoints for Faster Migrations

Thomas Knauth
Technische Universität Dresden
thomas.knauth@tu-dresden.de

Christof Fetzer
Technische Universität Dresden
christof.fetzer@tu-dresden.de

ABSTRACT

Virtual machine migration is a useful and widely used workload management technique. However, the overhead of moving gigabytes of data across machines, racks, or even data centers limits its applicability. According to a recent study by IBM [7], the number of distinct servers visited by a migrating VM is small; often just two. By storing a checkpoint on each server, a subsequent incoming migration of the same VM must transfer less data over the network.

Our analysis shows that for short migration intervals of 2 hours on average 50% to 70% of the checkpoint can be reused. For longer migration intervals of up to 24 hours still between 20% to 50% can be reused. In addition, we compared different methods to reduce the migration traffic. We find that content-based redundancy elimination consistently achieves better results than relying on dirty page tracking alone. Sometimes the difference is only a few percent, but can reach up to 50% and more. Our empirical measurements with a QEMU-based prototype confirm the reduction in migration traffic and time.

Categories and Subject Descriptors

C.2.4 [Computer Communication Networks]: Distributed Systems

General Terms

Design, Performance

Keywords

WAN migration, Virtualization, Cloud Computing

1. INTRODUCTION

Hardware virtualization in the form of virtual machines is ubiquitous and an integral part of modern day computing. One particularly useful aspect of virtualization is the ease with which VMs can be moved between physical servers,

i.e., VM migration. Migrations can happen for a variety of reasons such as load balancing, hot spot mitigation [27], and vacating servers for maintenance [22]. Typically, a migration is seen as a one-off event with little optimization potential across migrations, all the while the migration-related traffic poses a real problem to data center operators [22].

A recent large scale study by IBM Research revealed that VMs do not migrate randomly between hosts, but instead follow a pattern: most interesting is that the set of servers visited by a virtual machine (VM) is small. In 68% of the cases a VM visits just two servers [7]. While we can only speculate about the underlying cause, the potential optimization is clear. Because a VM only migrates between a small set of servers, it may be beneficial to store a checkpoint at each visited server. When migrating to a previously visited server, the old checkpoint can be reused to initialize the VM. This should decrease the migration traffic and the migration time. Reading from the local disk is potentially faster than over a potentially slow and congested network link.

Numerous papers have been written on how to improve the migration of VMs in one way or another. Related work [5, 28, 21, 30] exploits the redundancy found within a single VM or a cluster of VMs. After all, identical pages must only be transferred once from the source to the destination. Deduplication is not independent of how quickly the VM's memory changes over time. In contrast, the practical benefit of reusing local checkpoints greatly depend on the rate at which the VM's memory changes. As we will demonstrate, the similarity between the VM and a recent checkpoint decreases over time. However, even after 24 hours, 20% to 40% of the pages are still unchanged. For shorter time frames, e.g., 2 hours, the similarity is even more significant; upwards of 60%.

To decrease the migration traffic and time, we propose that each migration source locally stores a checkpoint of the outgoing VM. As the probability is high that the VM will return to the host at some point in the future, the incoming migration can be bootstrapped with the old checkpoint. We argue that local storage is cheap and abundant and VM checkpoints may already be kept for other reasons such as, for example, fault tolerance [10]. Our measurements confirm that the migration time and traffic is reduced by a percentage equivalent to the similarity between the VM's current state and its old checkpoint. In a scenario where virtual desktops are routinely migrated between user workstations and a central consolidation server, we see a reduction of the migration traffic by up to 75%.

| Name | OS | Trace ID | RAM size |
|----------|-------|--------------|----------|
| Server A | Linux | 00065BEE5AA7 | 1 GiB |
| Server B | Linux | 00188B30D847 | 4 GiB |
| Server C | Linux | 001E4F36E2FB | 8 GiB |
| Laptop A | OSX | 001B6333F86A | 2 GiB |
| Laptop B | OSX | 001B6333F90A | 2 GiB |
| Laptop C | OSX | 001B6334DE9F | 2 GiB |
| Laptop D | OSX | 001B6338238A | 2 GiB |

Table 1: Summary of the 6 systems whose memory traces we evaluated for the first part of our study. Trace IDs are listed for easy reference within the original data repository [28].

2. PROBLEM AND MOTIVATION

2.1 Terminology and Notation

Formally, a machine with m bytes of memory and a page size s consists of $n = m/s$ pages. For example, a machine with 2^{32} bytes of RAM and a typical page size s of 2^{12} bytes, has $2^{32}/2^{12}$ pages. Given a list of pages, p_0 through p_{n-1} , we can compute a *fingerprint* F of the machine’s memory by computing a list of hashes, $h(p_0)$ through $h(p_{n-1})$, i.e., one hash per page. The number of *unique hashes* in a typical fingerprint is less than the total number of pages, either because of true hash collisions, which should be rare, or because the content of the pages is actually identical, a much more likely scenario. Pages with identical content are, for example, due to shared libraries. In addition, freshly (re)booted machines have a large number of pages containing only zeros, further reducing the number of unique hashes. We use U to denote the set of unique hashes of a fingerprint.

2.2 Motivation

The fact that pages with identical content exist within a single machine, physical or virtual, and between multiple virtual machines on a single host, has previously been observed [5, 28, 21, 30, 12]. Merging multiple logically distinct but content-wise identical pages reduces the overall physical memory usage. However, in multi-tenant virtualized environments page sharing across tenants can compromise the security [23] and may thus be disabled in practice.

In the context of virtual machine migration, the redundancy inherent in identical pages is used to reduce the migration-related network traffic. If a page already exists at the destination, e.g., because a previously sent page has the exact same content, it need not be sent again. This optimization to reduce the amount of data sent over the network has previously been observed and exploited [29].

However, to the best of our knowledge, previous work only exploited the data redundancy if it occurred within the context of a single migration operation, be it a single VM or a group of VMs. Our work goes beyond detecting duplicate pages within a single migration: we propose to reuse old memory checkpoints to reduce the migration traffic and time. Whenever a VM migrates to a host it has visited before, the old checkpoint contains a previous version of the VM’s memory. Our goal is to reuse the old checkpoint when constructing the memory contents during an incoming migration. Assuming there is an overlap between the old checkpoint and the VM’s current state, reusing the old checkpoint reduces the migration-related network traffic. Our goal is to

also reduce the overall migration time, by reading from fast local storage instead of the potentially much slower network.

Two aspects are key for the scheme to work successfully: first, VMs must visit the same servers. This actually is the case as a recent cloud study by IBM reports. Birke et al. observed that instances typically move between a very limited set of hosts [7]. Often even just two hosts are involved in a ping-pong-style migration pattern. One possibility could be hard- and software updates to the host that require all VMs to be temporarily moved. Another alternative could be dynamic workload consolidation [26]: all low-activity VMs are consolidated on a single server and migrated to another machine as soon as they become active. Once the VM becomes quiet again, it moves back to the consolidation server.

2.3 How Memory Changes over Time

Besides the observations that VMs migrate only among a small set of hosts, the second important question is how quickly the memory content of a machine changes over time. After all, if the VM’s memory changes completely between two migrations, nothing can be gained from storing and reusing an old checkpoint.

To determine how the memory content of a typical machine changes over time, we used publicly available memory traces¹ originally collected for the Memory Buddies project [28]. The goal of the Memory Buddies project was to identify virtual machines suitable for co-location. Good co-location partners have a high fraction of identical pages amenable for merging. The authors were more interested in the similarity between machines, whereas we focus on the evolution of a single machine’s memory contents. Table 1 lists the machines we analyzed for the first part of our study.

For the Memory Buddy study the authors collected memory traces for several Linux servers and laptops over a period of 7 days. Each traced machine creates one memory fingerprint every 30 minutes. Over the course of one week, this ideally results in $7 * 24 * 60/30 = 336$ fingerprints for each traced machine. Due to server reboots and network connectivity problems a handful of fingerprints for the servers are missing. For the laptops, only between 151 and 205 fingerprints exist, of the theoretically possible 336. We hypothesize that this is a result of the different usage patterns. While servers run 24/7, laptops are active only when the user needs them to. Overall, the smaller number of fingerprints for the laptops, still reveals the same underlying trends.

For each machine we have a list of fingerprints F_0 through F_m . To evaluate the memory’s evolution, we enumerate all possible fingerprint pairs and calculate their similarity. We define the similarity between two fingerprints F_a and F_b as the fraction of shared unique hashes. Basing the similarity on the count of unique hashes is more to the point when estimating the benefit of old checkpoints. Identical pages within a VM could also be exploited with other content-based redundancy techniques [29]. For example, the similarity of U_a with U_b is defined as $\frac{|U_a \cap U_b|}{|U_a|}$.

Based on the Memory Buddy traces, we determine the similarity of each fingerprint pair. Assuming 336 fingerprints, one fingerprint every 30 minutes for the duration of one week, this gives $n * (n + 1)/2 = 336 * 337/2 = 56616$ possible fingerprint pairs. We sort the fingerprint pairs into

¹<http://skuld.cs.umass.edu/traces/cpumem/memtraces/>

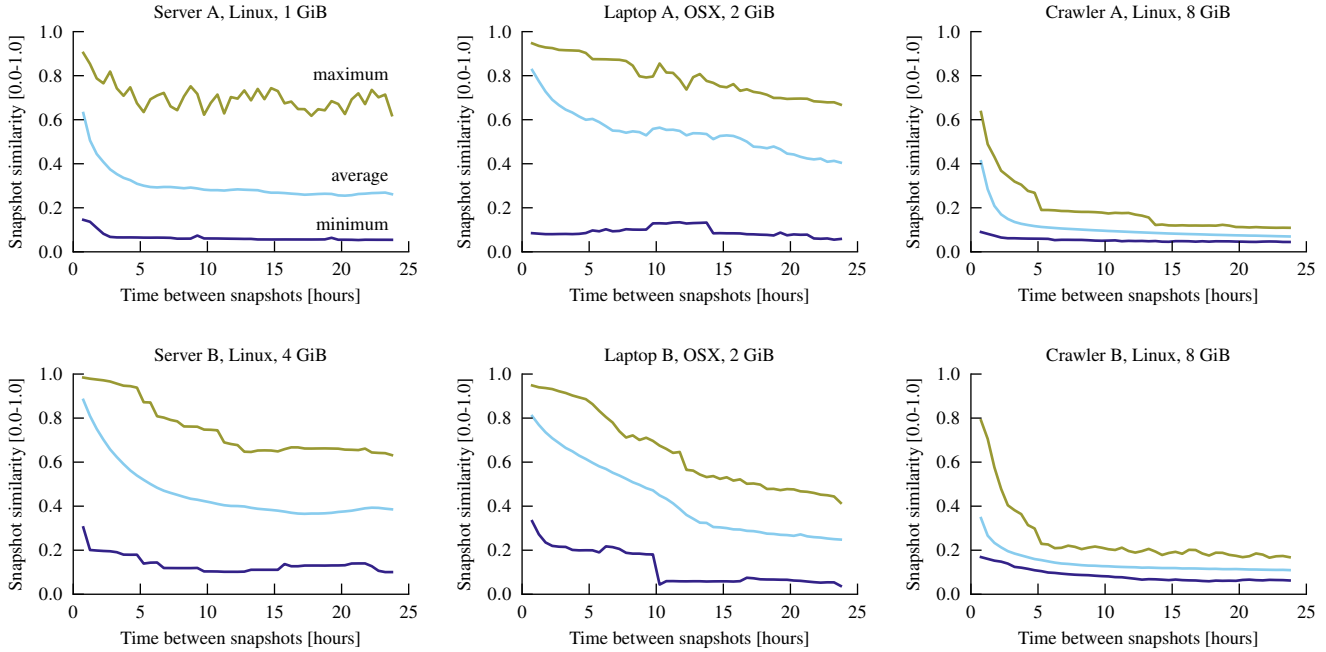


Figure 1: Time series of memory similarity for 2 servers (first column), 2 laptops (middle column) and 2 web crawlers (right column). Each figure shows the maximum, average and minimum similarity. Even though the similarity decreases over time, the average similarity after 24 hours is still between 10% to 40%.

bins according to their time delta. The first bin contains all pairs with a time delta (x-axis) in the range between [15, 45) minutes, the second bin [45, 75), and so on. For each bin we show the minimum, maximum, and average similarity up to a maximum time delta of 24 hours.

Figure 1 shows how the memory content for six machines changes over time. Each column shows data for a different kind of system. From left to right this includes two physical servers running a web/e-mail workload, two laptops running a desktop workload, and two virtual machines running a distributed web crawler. The general trend is as expected: the memory content changes over time. As a result, the similarity between fingerprints decreases as the time between two fingerprints increases. The worst case similarity, for all systems, quickly drops to below 20%, often even less than 10%, for example, in the case of Server C (cf. Figure 2). On average, the similarity after 24 hours is between 40% (Server B) and 20% (Server C). The difference between minimum, average, maximum similarity likely stems from different activity levels of the server. Periods of low activity result in a high similarity, whereas periods of high activity decrease the similarity.

In addition to the server and laptop traces from the Memory Buddy project, we also collected our own fingerprints. We traced 3 VMs running a modified version of the Apache Nutch web crawler [18]. Each VM had 4 cores and 8 GiB of memory. We collected 192 fingerprints for each VM: one fingerprint every 30 minutes over 4 days. For the web crawlers, the average similarity is only 40% after one hour and drops below 20% after 5 hours. This illustrates that the expected benefit depends on the actual workload within the VM. An active VM with no idle intervals will only gain a small benefit from a local checkpoint.

A real-world cloud likely consists of a mix of VMs of all activity levels; from constantly idle to constantly busy. Even though a checkpoint may not reduce the traffic for every VM and migration, we expect the average migration will still see a worthwhile reduction in migration traffic and time.

2.4 Expected Payoff

Based on how the memory content changes, we must also consider the specific scenario in which virtual machine migration is deployed to estimate the usefulness of reusing old checkpoints. In the study that originally sparked our investigation [7], the average time between migrations is 7 days. This is a long time, but even after one week there is some exploitable redundancy as Figure 2 shows. While we do not know the migration cause in Birke et al.’s study, the time between migrations is much smaller, on the order of a few hours, in other scenarios. For example, in the case of virtual desktop infrastructures [6, 20, 11] the time between migrations aligns with a typical workday, i.e., 8 and 16 hours. When the virtual desktop runs on the consolidation server, i.e., when the user is not interactively using the virtual desktop over night, we expect only minimal changes to the memory. Other use cases, such as follow-the-sun computing [25] and dynamic workload consolidation [26], also have inter-migration times of a few hours rather than multiple days.

3. DESIGN AND IMPLEMENTATION

After illustrating the potential benefits, we describe the design and implementation of our prototype. Our prototype is based on the open-source machine emulator QEMU/KVM because this is the virtualization technology we are most familiar with. In principle, the idea and concept can be

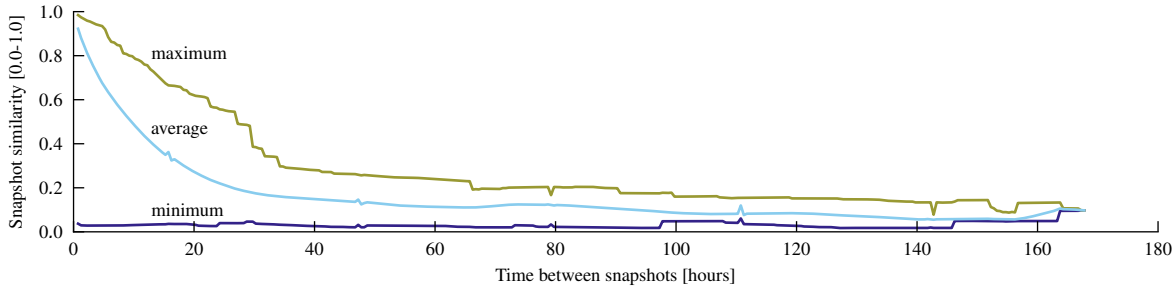


Figure 2: Server C’s snapshot similarity over the entire 7 day trace period. Even after one week about 20% of the memory content is unchanged. Depending on the use case, we expect the time between migrations to be much shorter, i.e., on the order of a few hours.

applied to other established virtualization technologies, such as Xen, HyperV, VMware, and VirtualBox.

3.1 Live Migration Algorithm Recap

Before detailing the changes we made to the migration logic, we recap the stages of a standard pre-copy live migration. Our description focuses on the memory portion of the VM’s state, as our optimizations only target the VM’s in-memory state. If migrating the on-disk state is necessary, i.e., because the source and destination do not share their storage, established techniques can be applied [16, 29].

A typical VM live migration progresses in a small finite number of rounds. During the first round, the VM’s entire memory, from the first to the last page, is transferred from the source to the destination. At the end of the first round, the destination has a more or less up to date picture save for the updates made at the source during the first round.

During a live migration the VM continues to run – answering request, serving queries – and performs memory updates as a result. In subsequent rounds, only the *dirty* pages are copied. This saves significant bandwidth, as unmodified pages need not be copied again. Ideally, the set of changed pages becomes smaller and smaller until, in a final round, the VM is paused, the remaining pages copied, and the execution resumed at the destination host.

In VeCycle, we only adapt the first iteration. Typically, the first round indiscriminately transfers all pages from the source to the destination. However, in VeCycle the migration source only sends pages if they do not already exist at the destination. We perform the necessary checksum computations and comparisons only during the first copy round. We consider it unlikely that a page updated between copy rounds matches a page already present at the destination.

3.2 Changes to the Migration Source

At the migration source, instead of sending every page indiscriminately, we only send the full page if it does not already exist at the destination. To this end, we compute the page’s checksum, currently MD5, and test if the checksum is within the set of checksums of pages existing at the destination. If the destination has the page already, we only send the checksum. Otherwise, we send both the page and the checksum. Sending the checksum along with the full page saves the receiver from re-computing the checksum for the received page.

This raises the question of how the sender knows the set of pages existing at the destination? In the case of migrating a VM back and forth between the same two hosts it is easy: when migrating from host *A* to host *B*, *B* keeps track of the incoming pages and their checksums. At the end of the incoming migration, server *B* knows the set of pages existing at server *A*. On an outgoing migration from *B* to *A*, *B* consults the set of checksums seen during the incoming migration. If the checksum of any outgoing page matches a checksum seen during the incoming migration, the page already exists at *A*. Again, on a match only the checksum is sent to the destination.

With migration patterns other than ping-pong, the destination host sends all the checksums of existing pages to the source in an initial setup phase. This would be an additional step before the actual migration. The size of the message is related to the VM’s size and the checksums used. For example, a 4 GiB VM has 2^{20} pages. Assuming each page has a unique MD5 checksum, the destination would send $2^{20} * 2^4$ bytes = 2^{24} bytes = 16 MiB of MD5 checksums to the source. Even in the worst case, i.e., a minimum similarity of a few percent, the additional traffic to exchange the checksums is compensated by the reduced migration traffic.

An alternative to sending the checksums in bulk before the actual migrations is to have the source send the checksum for each individual page to the destination. The destination would reply whether it knows the checksum or not. Although we have not evaluated this alternative scheme, we expect the high frequency exchange of small messages to slow down the migration performance. Hence, we send the checksums in-bulk before the actual migration begins.

3.3 Changes to the Migration Destination

When the destination VM starts up and prepares for an incoming migration, it initializes its main memory based on the contents of a checkpoint file. The path to the checkpoint file is given as a parameter to the VM. Before the VM is ready to receive the incoming migration, it sequentially reads the checkpoint file and copies the content into the memory representing the VM’s RAM. Sequential file access ensures optimal use of the disk’s available I/O bandwidth.

While reading through the checkpoint file, the VM calculates one checksum per 4 KiB block and records the checksum together with the file’s offset for later use. We currently keep the checksums and their offsets in a sorted list, such

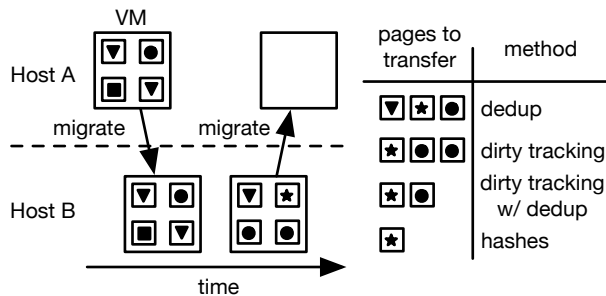


Figure 3: Different methods exist to reduce the network traffic during a migration and each method identifies a distinct set of pages to transfer. In the common case, deduplication transfers the most pages, followed by dirty page tracking. Checksum-based redundancy elimination typically performs better than dirty page tracking. It is also possible to combine the basic techniques.

that we can use binary search to quickly find the offset for a given checksum.

After the initialization, the destination sends the hashes of locally available pages to the source. Then the destination starts receiving messages from the migration source. In essence, each message is a page number plus either a checksum or the full page. If the message contains a checksum, the destination VM calculates the checksum for the page in its local memory. Because the local memory was initialized from an old checkpoint, chances are that the page frame already has the correct content. If the remote checksum does not match the local checksum, the remote checksum is used to lookup the page’s offset in the local checkpoint. The basic algorithm is given as pseudo-code in Listing 1.

3.4 Checksums

It is possible to determine updated pages by means other than checksums. For example, Akiyama et al. use dirty page tracking to reduce the migration traffic [3]. In principle, each technique identifies a unique set of pages to transfer, as illustrated in Figure 3. As our evaluation will show, content-based redundancy elimination yields the highest traffic reduction, i.e., outperforms stand-alone dirty page tracking.

Our prototype uses MD5 checksums to determine if two pages are identical. Widely used tools, such as `rsync`, use MD5 to this day to optimize the file transfer between remote hosts, even though (prefix) collision attacks on MD5 have been known for years. If MD5 is deemed a risk to security and correctness, the checksum can be replaced by more advanced algorithms, for example, SHA1 or SHA256.

Investigation of a checksum algorithm other than MD5 may become necessary if the checksum calculation itself becomes a bottleneck. Typically, the migration speed is determined by the available network bandwidth. Exclusive access to a gigabit Ethernet link allows the sender to transfer data at a rate of 120 MiB/s. Our benchmark machines can calculate MD5 checksums at a rate of 350 MiB/s on a single core, roughly 3 times faster than the bandwidth provided by gigabit Ethernet. For VMs with a low similarity, where a large fraction of the VM’s state must be sent over the network, the additional MD5 checksum computa-

```
read(checkpoint, mem, memory_size);
...
while (recv_msg(&addr, &checksum));
    if (MD5(&mem[addr], PAGE_SIZE) != checksum) {
        addr_old = lookup(checksum);
        lseek(checkpoint, addr_old, SEEK_SET);
        read(checkpoint, &mem[addr], PAGE_SIZE);
    }
}
```

Listing 1: Merging of the received pages/checksums and the existing checkpoint at the destination. This version allows out-of-order processing of pages, but requires to compute MD5 checksums. All reusable non-stationary pages are read from disk.

tion does not pose a problem over conventional gigabit links. For higher speed 10 or 40 gigabit links, the migration time will be dominated by the checksum rate. A cheaper checksum, hardware-acceleration, or multi-threaded execution are available options to increase the checksum rate. As our focus is on conventional and cost-effective gigabit Ethernet, we leave the investigation of higher transfer and checksum rates for future work.

With a high degree of similarity and a conventional gigabit link, the checksum rate still determines the achievable lower bound on the migration time. Even though VeCycle may speed up the migration time compared to a standard migration, the checkpoint-assisted migration will take at least as long as it takes to compute the checksums for the VM’s memory.

4. EVALUATION

The evaluation is split into four parts, answering the following questions: (1) How does VeCycle compare against sender-side deduplication? (2) How does VeCycle compare against dirty page tracking? (3) Does VeCycle reduce the migration time? (4) What are VeCycle’s expected payoffs in practice?

4.1 Setup

Our benchmark setup consists of three machines: two VM hosts and a third machine exporting shared storage. The two VM hosts have slightly different processor and memory configurations. Machine A is equipped with a 6-core AMD Phenom II processor and 12 GiB RAM. Machine B only has a 4-core Phenom II processor and 8 GiB RAM. Both machines have a 2 TB spinning disk (Samsung HD204UI) as well as a 128 GB solid state disk (Intel SSDSC2CT12). The disks are attached via Serial ATA (SATA) revision 2. During our benchmarks, the local disks only store the VM’s memory checkpoint, while the VM’s persistent storage is accessed over the network via NFS. The machines have a PCIe gigabit Ethernet card which is connected to a gigabit switch. The benchmark machines ran a recent version of Ubuntu (12.04) with a 3.2 Linux kernel, as did the VMs.

4.2 How Effective is Deduplication?

One question we asked ourselves was just how much more redundancy could be exploited in addition to existing techniques like deduplication. For example, CloudNet [29] dedu-

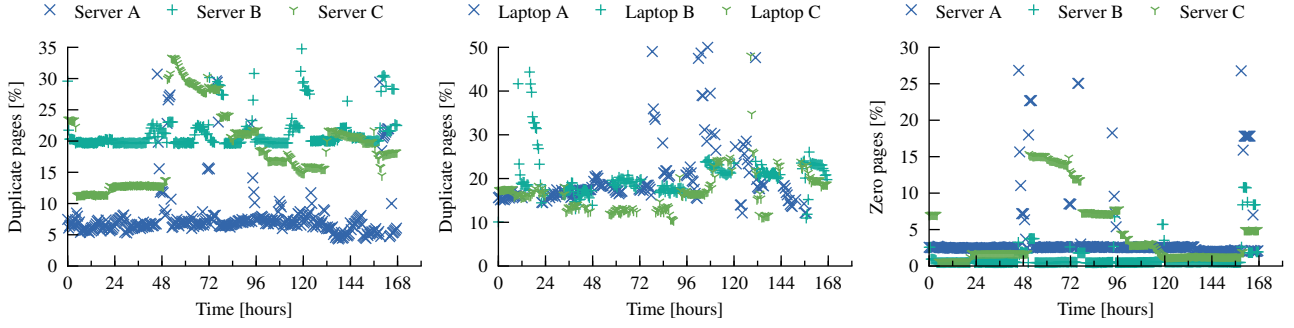


Figure 4: The percentage of duplicate pages varies between 5% and 20% for the servers and around 10% to 20% for the laptops. A high percentage of duplicate pages indicates redundancy exploitable by means other than recycling old checkpoints. Interestingly, empty pages, i.e., those only containing zeros, only make up a small part of the duplicate pages.

plicates pages at the migration source. The sender calculates one hash per outgoing page or part of a page. If the hash matches that of a previously sent page, and the pages are indeed identical, an index into the cache is sent to the destination instead of the full page. Note that it is possible for CloudNet to use hashes instead of cryptographically secure checksums to probe for potentially identical pages. Because the original page and its potential match both reside at the sender, they can be compared for actual equality without incurring any network traffic.

VeCycle has to rely on strong checksums, because the two pages are located on different physical hosts. A byte-for-byte comparison for actual equality would require to send the page over the network, which is exactly what VeCycle is designed to avoid.

Figure 4 shows a time series of the percentage of duplicate pages for 6 Memory Buddy machines [28]. In the context of memory sharing, i.e., merging of identical pages to save physical memory, this has previously been called *self-similarity* [5]. The fraction of duplicate pages is defined as $1 - (\text{unique hashes}) / (\text{total pages})$. We see that there is some difference in the percentage of duplicate pages between the servers, while the laptops all have a rather homogeneous fraction of duplicate pages. Server A has a very stable and low duplicate page count of slightly more than 5%. Server C has about 20% of duplicate pages, but at the same time, has even fewer zero pages among them than server A. To show that not all duplicate pages are due to zero pages, we also show the percentage of zero pages for the three servers in Figure 4’s rightmost plot. Although there are some spikes in the percentage of zero pages, it is stable and low at less than 5% for all three servers most of the time.

We find that the exploitable redundancy due to duplicate pages is only about 20% for most of the time. While periods with a higher percentage of duplicate pages exist, for example, during hours 48 to 72 for server C, this is uncommon. We conclude that stand alone deduplication is less effective than checkpoint-assisted migration. The latter often achieves similarity scores of 20% up to 70% (cf. Figure 1).

4.3 How Effective is Dirty Page Tracking?

Sender-side deduplication is not the only way to reduce the migration traffic. Even though pages with identical content are only sent once, they might still be part of an old

checkpoint at the destination.

An alternative to content-based redundancy elimination was explored by Akiyama et al.: their Miyakodori system uses dirty page tracking to determine updated pages [3]. Each page has a generation counter that is incremented if the page is written to after a migration. After an outgoing migration, the migration source stores a local checkpoint and the corresponding vector of generation counters. On an incoming migration, the checkpoint’s generation vector is compared with the VM’s current generation vector. Pages with matching generation counters need not be transferred.

In contrast to VeCycle, Miyakodori does not compute any checksums and instead relies on dirty page tracking to identify updates. However, Miyakodori may overestimate the pages to transfer. When pages move around in physical memory, as illustrated in Figure 5, they look like they have been updated, when in fact their content has not changed. In general, the set of pages identified by each method is distinct.

We analyzed the available traces to determine how effective each method is to reduce the migration traffic. The three basic methods are: (i) sender-side deduplication, (ii) dirty page tracking, and (iii) content-based redundancy elimination. It is also possible to combine the techniques. For example, dirty page tracking and content-based redundancy elimination can both be combined with deduplication. Note that adding dirty page tracking to deduplication or content-based redundancy elimination does not reduce the number of transferred pages. Dirty page tracking only reduces the computational overhead, as checksums need only be computed for dirtied pages.

We analyzed the available data sets with respect to the effectiveness of each method and possible combinations thereof. Unfortunately, neither the Memory Buddy traces nor our own contain actual dirty tracking information. Instead, given two fingerprints we say a page is dirty if its content changed between the two fingerprints. We were particularly interested in the difference between dirty page tracking, as implemented in Miyakodori, and content-based redundancy elimination, as implemented by VeCycle.

We constructed all possible fingerprint pairs for each of the machines in the Memory Buddies [28] study. For every pair, we calculated how many pages each technique would transfer. Figure 5 shows the average reduction for each

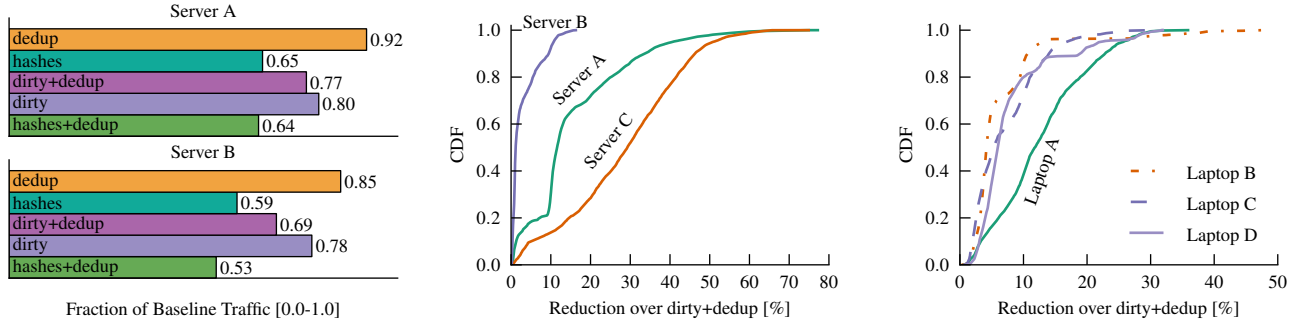


Figure 5: A comparison of different traffic reduction techniques and their combinations. Content-based redundancy elimination with deduplication often reduces the migration traffic by an additional 10% to 50% (and more) compared to dirty page tracking plus deduplication.

method for server A (left), one cumulative distribution function (CDF) plot for the servers (center), and one CDF plot for the laptops (right).

We find that deduplication transfers fewer pages than a straightforward full migration, but the other techniques reduce the traffic even more effectively. Dirty page tracking performs better than deduplication alone, but also benefits from deduplication (the bar for *dirty+dedup* is lower than for *dirty*). Content-based redundancy elimination (labeled *hashes*) transfers even fewer pages than dirty page tracking (with or without deduplication). At least in this particular example, combining content-based redundancy elimination with deduplication brings little, if any, benefit.

The CDF plots show how frequently and by which margin content-based redundancy elimination performs better than dirty page tracking combined with deduplication. We see, for example, that for Server A the difference between the two methods is minimal in 60% of the cases. For Server B, on the other hand, in 90% of the cases content-based redundancy elimination reduces the traffic by 10% and more when compared to dirty page tracking. For the laptops, content-based redundancy elimination outperforms dirty page tracking by at least 5% in half of the cases.

However, dirty page tracking determines on which subset of pages to calculate checksums, because clean pages can definitively be reused. This is especially attractive because dirty page tracking is done in hardware on modern Intel processors.

4.4 Best Case Scenario

We start the empirical evaluation of our prototype by measuring the migration time and traffic for an idle VM. This is the best case scenario, because we expect there to be maximum similarity between the VM’s current state and its most recent snapshot. Because the VM is idle, memory updates are rare. The VM runs Ubuntu 12.04 with some background daemons, but no other applications. We migrate the VM back and forth between the two benchmark hosts. After the migration, the source writes a checkpoint of the VM to its local disk. On the subsequent in-migration, the host uses the checkpoint to initialize the VM’s state.

We are interested in the total migration time and the migration-related network traffic. Both should be minimal because a large percentage of pages can be reused from the old checkpoint. We repeat the measurements for VMs with

different memory sizes up to a maximum of 6 GiB. Before the first migration, the VM executes a program which allocates 95% of the total memory and writes random data to it. Eliminating zero pages fits with our earlier analysis, i.e., Figure 4 showed the percentage of zero pages to be only a few percent at most. The operating systems aggressively use available memory to cache file system data; we expect all available memory to be used in the steady state.

We performed measurements for migrating the VM within a local area network and an emulated wide area network. For the wide area network we used the properties specified in the CloudNet paper [29]: a maximum bandwidth of 465 Mbps and an average latency of 27 milliseconds. Though local and wide area networks differ in a number of characteristics, most relevant to our evaluation is the WAN’s reduced effective bandwidth and its impact on how quickly the migration finishes. The available migration bandwidth may also be limited in a local area network, as the migration traffic competes with other network users. We used *netem* to emulate the wide area network in our Linux benchmark environment.

Figure 6 shows the average migration time and traffic for various VM sizes. We compare VeCycle with the standard migration implementation of QEMU/KVM version 2.0. The migration time captures how long it takes from initiating the migration at the source until the VM runs at the destination. We explicitly do not capture the setup phase at the destination or the time to write the checkpoint at the source. This is reasonable because the workload within the VM is primarily affected during the actual copy phase. As pages must be marked read-only to track the VM’s write set, there is some performance impact while the copy is in progress. Similarly, the background network traffic can negatively impact the VM’s performance. Hence, we discount the migration setup at the destination as well as writing the checkpoint at the source from the migration time.

As expected, the migration time grows linearly with the VM’s size; both for VeCycle and the QEMU/KVM’s default migration routine. The time to migrate a VM for QEMU/KVM is limited by the available network bandwidth, 1 gigabit in our setup. Copying one gigabyte takes about 10 seconds over a gigabit link. The migration of a VM with 1 GiB of memory takes around 10 seconds. Large VMs with 6 GiB of memory take around 60 seconds to migrate. VeCycle benefits from the fact that calculating and comparing

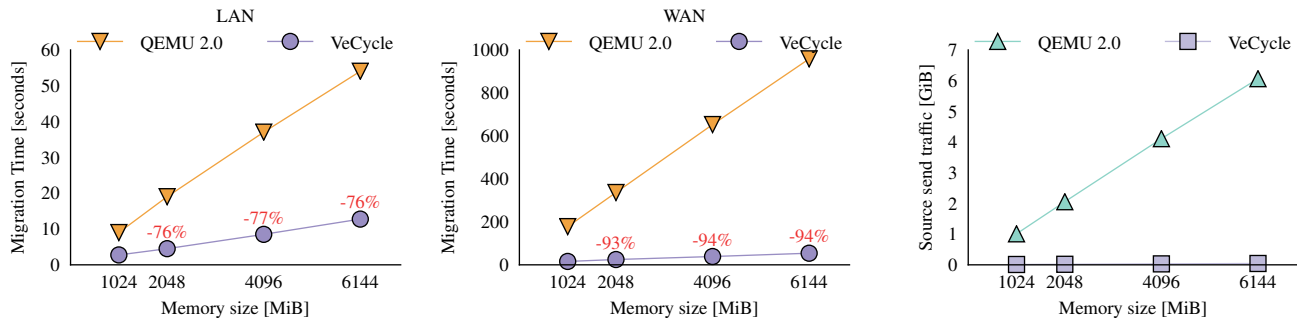


Figure 6: As the VM’s memory size increases, migrating over a local area network (left) and an emulated wide area network (center) takes progressively longer. The outgoing network traffic at the migration source also increases proportionally (right). This is a best case scenario with extremely high similarity of almost 100%, for example, an idle VM.

checksums can be done at a higher rate than transferring data over the network. It takes only 3 seconds to migrate small VMs (1 GiB) and 13 seconds for large VMs (6 GiB). VeCycle is 3 times faster on small VMs and around 4 times faster on large VMs than the default migration implementation.

Across an emulated wide area network the benefit of reducing the data volume becomes even more apparent. A single VM with 1 GiB of memory takes 177 seconds on average to migrate. This comes out to an average transfer rate of just over 6 Mbps; significantly lower than within the local area network. With VeCycle it only takes 16 seconds, because the data volume is decreased by two orders of magnitude: from 1 GB down to 15 MB. The reduction in migration time is even more impressive for large instances. For VMs with 6 GiB of memory, it takes 16 minutes to copy the state to the destination, compared to less than one minute for VeCycle.

The measurements presented in this section used a spinning hard disk on the host to store the checkpoint. We repeated the same set of experiments with a solid state disk, but the migration times did not change. Storing checkpoints on rotating disks is more cost-effective as spinning disks still have a lower cost per gigabyte than solid state disks.

4.5 Varying Update Rates

An idle VM allows us to observe the maximum benefit of our checkpoint-assisted migration. If the time between migrations is short, improvements close to what we described in the previous section are possible. To see how VeCycle’s payoffs change as the similarity between the VM and its previous checkpoint decreases, we performed a second set of benchmarks where we had full control over the percentage and distribution of memory updates.

For our controlled environment, we allocate a ramdisk within the VM taking up 90% of the VM’s memory. In our measurements we used a VM configured with 4 GiB of RAM. We create a single large file within the ramdisk, filling it sequentially with random data. The way the Linux kernel allocates memory for the ramdisk, the file is laid out sequentially in the VM’s physical memory. Subsequently, we can update select blocks of this single large file to create specific update patterns within the VM’s physical memory.

We expect the results of our controlled environment to

be representative of real-world workloads. After all, the expected benefit of VeCycle depends mainly on how much the VM has diverged from the checkpoint. While real-world workloads may have different update patterns, VeCycle’s performance gains are much more sensitive to the overall similarity. We report results for LAN and WAN migration as before. Storing the checkpoint on SSD instead of HDD had no impact on the results.

For Figure 7 we randomly updated 25%, 50%, 75%, and 100% of the ramdisk within the VM. The results align perfectly with our expectations. As the percentage of memory updates increases, i.e., the similarity decreases, the migration time for VeCycle increases proportionally and approaches the migration time of the baseline QEMU implementation. The baseline migration time for QEMU is independent of any memory updates *between* migrations, hence the baseline curve is flat.

For WAN migrations the correlation between memory updates and migration time is the same. Only the absolute times to migrate are much higher than within a local area network. The smaller effective throughput of the WAN link determines the total migration time. The data volume sent over the network corresponds to the size of updated memory in the case of VeCycle. QEMU always sends the full memory content of 4 GiB in this particular scenario.

4.6 Virtual Desktop Example

Finally, we demonstrate the applicability and effectiveness of VeCycle in a real-world use case. Virtual desktop infrastructures (VDI) have previously been investigated in the context of energy-aware computing [6, 11]. The user’s desktop workstation is virtualized and migrated to a central consolidation server during idle periods, e.g., over night. Consolidating the virtualized desktops saves energy because the physical workstation can be powered off. The virtual desktop continues to run on the consolidation server, which preserves the always-on semantics users have come to expect.

We collected memory traces from a desktop computer belonging to one of the author’s. One memory fingerprint is recorded every 30 minutes for 19 days (5 Nov 2014 to 23 Nov 2014), resulting in 912 fingerprints in total. The desktop is a Linux machine (Ubuntu 10.04) with 6 GiB of RAM used for web, e-mail, and day-to-day research work.

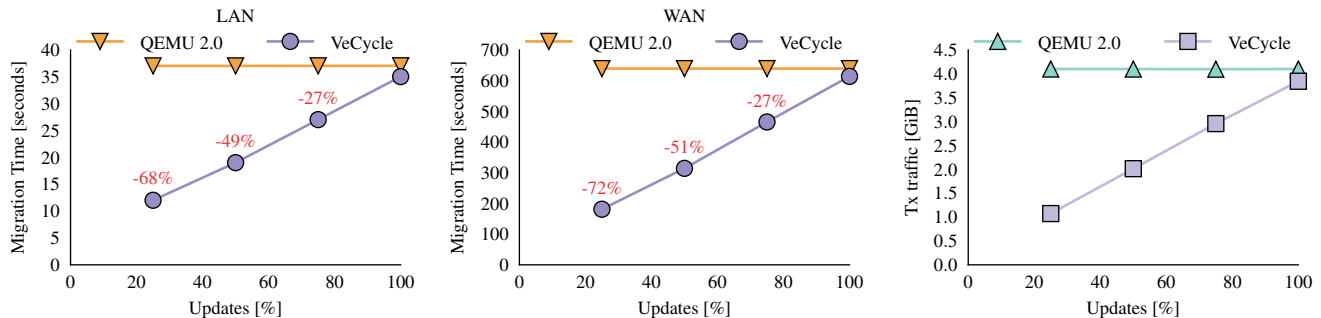


Figure 7: The plots show migration time (left and middle) and traffic volume (right). Lower is better in both cases. As more and more of the memory changes between migrations, the benefit of reusing an old checkpoint decreases accordingly.

After the data collection stage, we analyzed the fingerprints to analytically derive the benefit of deploying VeCycle in an energy-aware VDI consolidation scenario. We assume conservatively that two migrations happen every weekday. In a real system, the decision to migrate would be based on the user’s presence and the frequency of interactions with the desktop. We did not record information relevant to make informed migration decisions, hence we assume a fixed migration schedule. The VM migrates from the consolidation server to the workstation when the user arrives at the office (9 am). A second migration happens in the late afternoon (5 pm) when the user leaves the office. There are no migrations over the weekend. This leaves us with 13 weekdays during the trace period and a total of 26 migrations.

By calculating the overlap between fingerprints we determine the reduction in migration traffic due to VeCycle. The degree to which VeCycle reduces the migration traffic is visible in Figure 8. For our single desktop computer, 26 full migrations would result in roughly 159 GB of network traffic. On-the-fly deduplication of pages at the sender reduces the traffic to 138 GB, 86% of the baseline. VeCycle is able to reduce the total traffic to 40 GB, just 25% of the baseline, and 29% when compared to on-the-fly deduplication. Overall, this is a sizable reduction of the migration-related traffic and clearly illustrates the benefit of VeCycle. Note that VeCycle cannot reuse any checkpoint on the very first migration, which is why the first migration causes the most traffic. We assume that VeCycle still uses deduplication to reduce the migration traffic, i.e., only about 90% of the total number of pages are transferred.

5. RELATED WORK

There exists a large body of work in the area of virtual machine migration in general and live migration in local and wide area networks in particular. The basic technique of migrating a virtual machine, then called “capsule”, is more than a decade old [20] and is itself predated by process migration [17]. Sapuntzakis et al. pioneered the techniques still used today to optimize the migration of VMs. Our work has the same goals, i.e., optimizing the migration, but applies it to a broader range of uses cases. Instead of only focusing on desktop environments that follow their users wherever they go, we show that arbitrary VMs running in a data center can benefit from the same optimization techniques.

Since the original idea of migrating entire execution con-

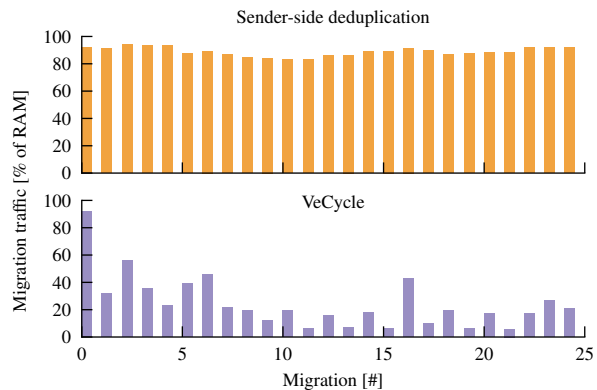


Figure 8: Sender-side deduplication at the migration source reduces the network traffic by 10% to 20%. VeCycle reduces the aggregated migration traffic by 75% compared to a full migration. VeCycle also transfers 9% fewer pages than dirty tracking in combination with deduplication.

texts, the next step was to do so with minimal service interruption [9]. Over time, the basic live migration technique was extended, for example, by switching to a post-copy instead of a pre-copy approach [13] or using record/replay in lieu of explicitly copying the state [14]. Compressing the migration data also helps to reduce the data volume [24]. All the insights from these works are still valid and can be combined with VeCycle. Still, each migration is viewed as a singular and independent event.

Miyakodori [3], which also reduces the migration traffic by reusing VM checkpoints, is closest to our work. In addition, we show that content-based redundancy elimination achieves better results than only tracking dirty pages. We also provide additional data points that exploitable redundancy actually exists in real-world workloads.

Migrating across wide area networks is particularly challenging because of the limited bandwidth and high latency links. Many works exist to optimize the migration across WANs in one way or another [19, 29, 4, 16, 8]. Again, each migration is considered in isolation. Extending earlier work on migrating within local area networks, optimizations are

now applied to entire VM clusters. Instead of deduplicating only the pages of a single VM, duplicates in all memory pages of the entire VM cluster are eliminated. The reuse of state already present at the destination is only done for persistent storage. We show that it is beneficial to apply the same idea also to the VM’s memory.

Data on the properties and characteristics of large VM populations is notoriously rare. A recent study from IBM revealed interesting insights with respect to migration patterns in real cloud deployments [7]. The key take away is that migrating VMs only move between a small set of servers, often just two. Although we do not know the underlying reason leading to the patterns, it is still noteworthy and sparked the current study. Other studies have also highlighted the usefulness of VM migrations as a workload management tool within the modern data center. However, the migration-related network traffic is a real problem and affects the frequency with which migrations can be performed [22, 26]. Our work directly addresses this pain point by reducing the migration-related traffic.

Related work also exists in the area of deduplication for in-memory and on-disk data. For example, the Memory Buddies project [28] uses the percentage of sharable pages as a guide for good co-location candidates. Instead, we analyzed the traces with respect to how quickly the overall memory content changes. Previous studies on page sharing [5, 21] never looked at the memory change rate, but only at how the set of duplicate pages evolves over time.

In general, there is a large body of work on how to employ VM migration to solve problems such as hot spot mitigation [27], increase the percentage of shareable pages [21, 12], decrease the contention on shared resources [2], and consolidate virtual desktops to save energy [11, 6]. Jettison [6] only moves the working set of an idle virtualized desktop to the consolidation server. Conversely, only pages dirtied on the server are copied back to the desktop. VeCycle generalizes this concept by persistently storing checkpoints at each visited server. We also apply the concept to a broader range of services beyond idle desktop VMs. These and future use cases will all benefit from cheaper migrations.

Lastly, one might question the whole concept of hardware virtualization and migration of heavyweight virtual machines. Approaches such as unikernels [15] and custom cloud operating systems, e.g., ZeroVM [1], demonstrate a more lightweight approach to application development and deployment “in the cloud”. A typical unikernel application is less than one megabyte in size, compared to hundreds of megabytes for a single VM. While the reduced space/memory footprint is a good selling point, people also like to reuse legacy applications and code. Unikernels and similar approaches put a high burden on the developer because they have to (re)write their application in a possibly unfamiliar language, e.g., Erlang and work with a possibly unfamiliar runtime. VMs allow the average programmer to develop and deploy in a familiar environment.

6. CONCLUSION

Our study was motivated by the observation that virtual machines within a data center only migrate between a small set of servers [7]. Based on this observation, we used publicly available and our own memory traces to look at how the memory content of a virtual/physical machines changes over time. We found that within a few hours a significant part

of the memory is unchanged. Even over longer time frames of 24 hours and more, up to 40% of the memory is still reusable. Hence, there is an opportunity to decrease the traffic and time when migrating virtual machines.

We then described our prototype implementation and discussed various design decisions. On an outgoing migration, the source writes a checkpoint of the VM to its local disk. A subsequent incoming migration of the same VM reuses the local checkpoint to bootstrap the VM. Only pages not already part of the checkpoint are sent across the network. Empirical measurements within a local area and an emulated wide-area network show that VeCycle reduces the network traffic and speeds up the migration as intended. We also found that content-based redundancy elimination sends fewer pages than alternative techniques, in particular dirty page tracking.

Finally, we analyzed the benefit of VeCycle within a virtual desktop infrastructure where virtualized desktops are periodically migrated between the user’s PC and a consolidation host. Based on real-world traces, we found that VeCycle effectively reduces the migration traffic by 75% compared to a full system migration and by 71% when compared to sender-side deduplication.

The QEMU code is available at <https://bitbucket.org/tknauth/vecycle-qemu> and we are working to have it integrated in mainline. The fingerprint traces are located at <http://wwwpub.zih.tu-dresden.de/~vecycle/>. Other artifacts related to this work are available at <https://bitbucket.org/tknauth/vecycle>.

7. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback, as well as Björn Döbel and Pascal Felber for their input on early drafts of this paper. In addition, Bohdan Trach helped with the QEMU prototype, Do Le Quoc set up the crawler experiments, and Cloud&Heat Technologies GmbH provided access to their infrastructure to collect the crawler traces.

This research was funded as part of the ParaDIME project supported by the European Commission under the Seventh Framework Program (FP7) with grant agreement number 318693. This research was also funded as part of the Sereca project under the H2020 Program with grant agreement number 645011.

References

- [1] URL <http://zerovm.org/>.
- [2] J. Ahn, C. Kim, J. Han, Y.-r. Choi, and J. Huh. Dynamic Virtual Machine Scheduling in Clouds For Architectural Shared Resources. In *Workshop on Hot Topics in Cloud Computing*. USENIX, 2012.
- [3] S. Akiyama, T. Hirofuchi, R. Takano, and S. Honiden. Miyakodori: A Memory Reusing Mechanism for Dynamic VM Consolidation. In *2012 IEEE 5th International Conference on Cloud Computing*, pages 606–613. IEEE, 2012.
- [4] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Rippeanu. VMFlock: Virtual Machine Co-Migration for the Cloud. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, pages 159–170. ACM, 2011.

- [5] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman. An Empirical Study of Memory Sharing in Virtual Machines. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 273–284, Boston, MA, 2012. USENIX. ISBN 978-931971-93-5. URL <https://www.usenix.org/conference/atc12/technical-sessions/presentation/barker>.
- [6] N. Bila, E. de Lara, K. Joshi, H. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan. Jettison: Efficient Idle Desktop Consolidation with Partial VM Migration. In *EuroSys*, 2012.
- [7] R. Birke, A. Podzimek, L. Y. Chen, and E. Smirni. State-of-the-practice in Data Center Virtualization: Toward a Better Understanding of VM Usage. In *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12. IEEE, 2013.
- [8] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *International Conference on Virtual Execution Environments*, pages 169–179. ACM, 2007.
- [9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.
- [10] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, 2008.
- [11] T. Das, P. Padala, V. Padmanabhan, R. Ramjee, and K. Shin. LiteGreen: Saving Energy in Networked Desktops Using Virtualization. In *Proceedings of the 2010 USENIX Annual Technical Conference*. USENIX Association, 2010.
- [12] D. Gupta, S. Lee, M. Vrabie, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. *Communications of the ACM*, 53(10): 85–93, 2010.
- [13] M. Hines and K. Gopalan. Post-copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning. In *International Conference on Virtual Execution Environments*, pages 51–60. ACM, 2009.
- [14] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live Migration of Virtual Machine Based on Full System Trace and Replay. In *International Symposium on High Performance Distributed Computing*, pages 101–110. ACM, 2009.
- [15] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 461–472. ACM, 2013.
- [16] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, and S. Setty. XvMotion: Unified Virtual Machine Migration over Long Distance. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 97–108, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL <http://blogs.usenix.org/conference/atc14/technical-sessions/presentation/mashtizadeh>.
- [17] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *SIGOPS Operating Systems Review*, 36(SI):361–376, Dec. 2002. ISSN 0163-5980. doi: 10.1145/844128.844162. URL <http://doi.acm.org/10.1145/844128.844162>.
- [18] D. L. Quoc, C. Fetzer, P. Fellber, É. Rivière, V. Schiavoni, and P. Sutra. Unicrawl: A practical geographically distributed web crawler. In *8th IEEE International Conference on Cloud Computing (CLOUD’15)*. IEEE Computer Society, July 2015.
- [19] P. Riteau, C. Morin, and T. Priol. Shrinker: Improving Live Migration of Virtual Clusters over WANs with Distributed Data Deduplication and Content-Based Addressing. In *Euro-Par 2011 Parallel Processing*, pages 431–442. Springer, 2011.
- [20] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. *ACM SIGOPS Operating Systems Review*, 36(SI):377–390, 2002. ISSN 0163-5980.
- [21] M. Sindelar, R. Sitaraman, and P. Shenoy. Sharing-aware Algorithms for Virtual Machine Colocation. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, pages 367–378, New York, NY, USA, 2011.
- [22] V. Soundararajan and J. M. Anderson. The impact of management operations on the virtualized datacenter. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA ’10, pages 326–337, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1816003. URL <http://doi.acm.org/10.1145/1815961.1816003>.
- [23] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory Deduplication as a Threat to the Guest OS. In *Proceedings of the Fourth European Workshop on System Security*. ACM, 2011.
- [24] P. Svård, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. In *International Conference on Virtual Execution Environments*, pages 111–120. ACM, 2011.
- [25] J. Van der Merwe, K. Ramakrishnan, M. Fairchild, A. Flavel, J. Houle, H. A. Lagar-Cavilla, and J. Mulligan. Towards a ubiquitous cloud computing infrastruc-

ture. In *Local and Metropolitan Area Networks (LAN-MAN)*, 2010 17th IEEE Workshop on, pages 1–6. IEEE, 2010.

- [26] A. Verma, J. Bagrodia, and V. Jaiswal. Virtual Machine Consolidation in the Wild. In *Proceedings of the 15th International Middleware Conference*, pages 313–324. ACM, 2014.
- [27] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-Box and Gray-Box Strategies for Virtual Machine Migration. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007.
- [28] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. In *International Conference on Virtual Execution Environments*, pages 31–40. ACM, 2009.
- [29] T. Wood, K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *International Conference on Virtual Execution Environments*. ACM, 2011.
- [30] X. Zhang, Z. Huo, J. Ma, and D. Meng. Exploiting Data Deduplication to Accelerate Live Virtual Machine Migration. In *IEEE International Conference on Cluster Computing*, pages 88–96. IEEE, 2010.