Master System and Network Engineering

University of Amsterdam

Large Installation Administration

Cluster Samepage Merging

Mike Berkelaar mike.berkelaar@os3.nl

Cedric Van Bockhaven cedric.vanbockhaven@os3.nl

March, 2014

Abstract

Kernel Samepage Merging (KSM) is a technique used to determine duplicate memory blocks. This is especially of interest on hypervisors where multiple virtual machines may be able to share a large number of memory segments.

The current use of KSM limits itself to improving the memory use on a per host base. In large environments it may be interesting to find a more optimal deployment of virtual machines, consolidating virtual machines with duplicate memory segments onto the same host. This research describes a number of algorithms and techniques to (re)deploy virtual-machines in order to maximize the memory density with deduplication opportunities.

Contents

1 Introduction

A number of mechanisms are actively being used to increase density of virtual machine (VM) deployments, like resource overprovisioning, memory deduplication and dynamic memory allocation [1]. This research is inspired by memory deduplication, especially as found implemented in KSM [2] and being used with the KVM hypervisor.

KSM can efficiently deduplicate memory of virtual machines, but it is limited in its scope to the local hypervisor. As virtual machine deployments in multi-server environments may be suboptimal for deduplication schemes this research was performed to look into ways of leveraging these local mechanisms to optimize the deployment for an entire environment. By migrating virtual machines that share large amounts of duplicate memory pages to the same hypervisor it is believed that an increased amount of memory pages can be deduplicated.

This research focuses on comparing the raw memory pages of VMs and explores a number approaches to finding an optimal deployment scenario.

1.1 Related work

The challenges of memory deduplication itself have been documented in a number of research papers:

Kernel Samepage Merging (KSM) is a memory deduplication mechanism implemented in the Linux Kernel [2]. Although applicable to all sorts of processes, its main use is to deduplicate virtual machines hosted by the KVM hypervisor. KSM has been inspirational to this research with the method of keeping a stable- and unstable binary search tree and calculating the hashes of memory pages in general.

KSM++ is a research project that aims to improve on the previously discussed KSM mechanism [3]. KSM++ claims to be a more aggressive mechanism of deduplicating memory by looking at I/O-based hints to exploit short-lived deduplication oppurtunities.

The Memory Buddies [4] paper explains a comparable research that fingerprints virtual machines to create lists of hash values. Their proposed solution contains a number of phases that opportunistically identifies machines that are candidates for consolidation. Besides finding alternative deployment scenarios there are also scenarios described that require attention in

this process, like the mitigation of memory hotspots and intermediate relocations to free up space.

Furthermore, Li et al. propose solutions for the placement of VMs that focus on energy efficiency in the EnaCloud paper [5].

In this research project we explore a universal approach to the hashing of live virtual machines and a number of centralized and distributed mechanisms to determine an improved or even optimal placement of VMs.

2 Problem statement

The question this research project focuses on is *if the principle behind KSM can be used on a cluster of multiple VM hosts to determine which guests to place together for maximal memory deduplication.*

A few items that will be further looked at during this research:

- How can memory pages efficiently be compared?
- How could this scale in large environments?

2.1 Approach

In order to improve the deduplication ratios, it is required to place virtual machines that could potentially share favorable amounts of memory together on the same hypervisor, so mechanisms like KSM can deduplicate the memory. To do this, we propose a number of models that compare the hashes of in-use memory pages of the virtual machines in an environment. Both a centralized comparison and a distributed optimization approach are explored.

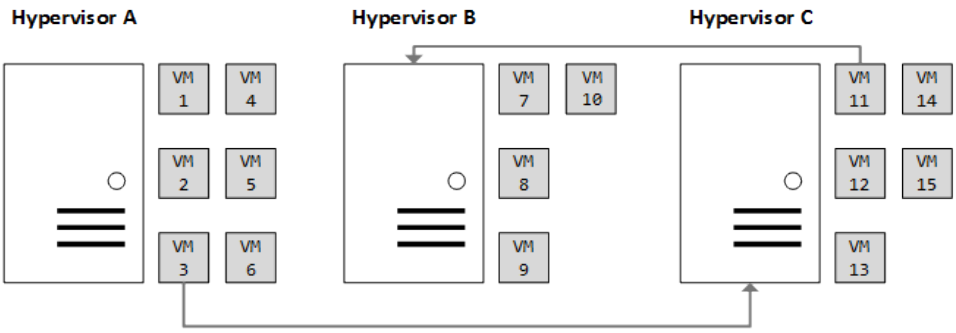

Figure 1: Before and after VM reconfiguration

3 Scanning

3.1 Memory hashing

The memory contents of virtual machines can generally be accessed by the hypervisor. As the memory size of virtual machines is usually an order of multiple gigabytes it is not feasible to store an entire memory dump. Instead, calculating the hashes of the memory contents creates a dataset that is easy to store and process. Memory contents are generally ordered in fixed size pages of 4096 bytes on most architectures [6].

KSM creates such a list of hashes by running the jhash2 algorithm over the memory pages, which is a fast hash function adapted from the *Lookup3* algorithm by Bob Jenkins [7]. The hashes are subsequently stored in a red-black tree [1]. There is an unstable and stable red-black tree: when a memory page is being tracked in the unstable tree and remains unchanged after a future pass, it is moved to the stable tree. Only pages in the stable tree are effectively being deduplicated.

Since KSM is readily available on linux systems, its internal data structures that contain the hashes could be read and used in our proposed mechanisms. This could be done by reading out the kernel memory of the `root_stable_tree` symbol which is defined in mm/ksm.c. All memory addresses of global kernel symbols can be found under /proc/kallsyms. A separate custom-made kernel module would have to be created and loaded in order to be able to access these memory contents.

CSM does not read KSM's internal memory at this point. Instead, memory is being hashed by a standalone script to have more control over when and how the hashing takes place, while it also doesn't require interaction with the kernel. This way, our mechanism doesn't rely on KSM and could be used in combination with other memory deduplication solutions.

While jhash2 is being used in the linux kernel, there are newer hashing algorithms available that may have better performance. In order to hash the vast amount of memory pages required by our mechanism, there has to be an emphasis on speed. xxHash [8] is a non-cryptographic hashing algorithm that claims to be fast while still producing few collisions. Based on the SMHasher test set [8] [9], xxHash outperforms other algorithms with a speed of 5.4GB/s compared to 1.2GB/s for the Lookup3 algorithm.

[1] A red-black tree is a self-balancing binary search tree.

For KVM, the following course of action can be followed to index and hash the virtual machines and their memory regions:

1. Identify all running VM processes. Pid files as found in `/var/run/libvirt/qemu`.
2. Get the logical RAM regions corresponding to the VM's RAM from `/proc/PID/smaps`.
3. Read the VM's RAM page by page, hash the page with xxhash, and add it to a dataset.

Other virtualization technologies like Xen and VMware allow access to the guest's memory in other ways, but the specification of these methods falls out of the scope of this research. The choice of an appropriate dataset is documented in chapter 3.2.

3.2 Pre-processing

The calculated hashes of the memory-contents of all virtual machines result in a large dataset. This data has to be structured so that it can be efficiently processed later on. The type of data that has to be structured has the characteristic of already being a hash value, which means that the creation of a hash table is likely to be less costly than when still having to calculate a hash for every key in the data-set [10]. Also the size of the hash table is known in advance as the length of the data-set, being the memory-allocation of the source VM, is a given.
Hashtables are performant in the area of doing look-ups at $O(1)$ on average. Even though this may not seem like a bottleneck, there are a number of optimalizations that can be done to decrease the volume of the hash tables. One can for example filter out a number of hashes like those of empty (Null) memory pages, as these are generally (sparse) memory pages handled by overprovisioning mechanisms like memory-balloons [11]. Other pages that can be filtered beforehand are the pages that are known to be present on only a single virtual machine, making it impossible for this page to be deduplicated.

Another way of limiting the hash table to only the likely-duplicate pages is by only processing the memory pages that are considered stable over a longer period of time. By doing successive scans with a short interval we can easily recognize pages that changed, rendering them too unstable for memory deduplication. This approach is similar to the unstable- and

stable-trees of KSM [2].

For the proposed models, it's important to think about the effect of the null hash. This is the hash that is produced when only null bytes occur in a page, and it usually occurs a lot more than other hashes. As these memory blocks are sparse, they can better be left out of the calculations.

4 Optimization models

A number of different approaches were thought of to optimize the virtual machine deployment to have as much memory overlap as possible. Both centralized and distributed approaches have been researched.

In the following models, a difference between shared and sharing pages is being made (see Appendix A). Both are terms that are adopted from KSM.

4.1 Centralized

The algorithms discussed in this chapter are executed on a central processing node. All physical machines gather the hash tables of the running virtual machines and offer them to the processing node. By processing them centrally, the load on the physical hosts is limited to only creating the hash tables of the memory dumps.

The placement of VMs can be seen as a bin packing problem: we start from a situation in which VMs have to be placed onto physical machines (the bins), while maximizing the memory density. However, the problem of bin packing is NP-hard [12].

First, the problem is approached from a theoretical point of view with an exact mathematical model, after which more practical algorithms and models will be introduced.

For the implementations of these algorithms, see Appendix B.

4.1.1 Maximizing the sharing pages

A model is proposed to optimize VM placement by maximizing the total number of sharing pages across all physical machines (1). This is an exact model that should deliver an optimal solution.

Since this model doesn't scale well, a model with simpler constraints is documented in section 4.1.2. A greedy algorithm is proposed in section 4.1.3.

Data:

P	: set of physical machines	V	: set of virtual machines
p	: a specific physical machine	v	: a specific virtual machine
p_{mem}	: memory of p (KiB)	v_{mem}	: memory of v (KiB)
H	: set of all hashes	H_v	: set of hashes on v
h	: a specific hash	h^v_{num}	: times the hash occurs on v
M	: big integer (Big M-method)		

Decision variables:

L^p_v : if a virtual machine v will be placed on physical machine p (boolean)
S^p_h : if the hash h is being shared on the physical machine p (boolean)
S^p : the total sharing amount on the physical machine p (int+)
F^p_h : number of times a hash h occurs on a physical machine p (int+)
T^p_h : the sharing amount of a hash h on physical machine p (int+)

Objective:

$$max \sum_{p \in P} T^p \tag{1}$$

Subject to:

$$T^p = \sum_{h \in H} T^p_h \qquad \forall p \in P \tag{2}$$

$$\sum_{p \in P} L^p_v = 1 \qquad \forall v \in V \tag{3}$$

$$\left(\sum_{v \in V} L^p_v \cdot v_{mem} \right) - 4T^p \leq p_{mem} \qquad \forall p \in P \tag{4}$$

$$F^p_h = \sum_{v \in V} \sum_{\substack{i \in H_v \\ h=i}} L^p_v \cdot h^v_{num} \qquad \forall p \in P, \forall h \in H \tag{5}$$

$$\begin{aligned} F^p_h &\geq 2S^p_h && \forall p \in P, \forall h \in H \\ F^p_h &\leq 1 + M \cdot S^p_h && \forall p \in P, \forall h \in H \\ F^p_h - 1 &\leq T^p_h && \forall p \in P, \forall h \in H \\ T^p_h &\leq F^p_h - S^p_h && \forall p \in P, \forall h \in H \\ T^p_h &\leq M \cdot S^p_h && \forall p \in P, \forall h \in H \end{aligned} \tag{6}$$

The number of sharing hashes on a physical machine S^p is fixed by taking the sum of the sharing amount of each hash on that physical machine S_h^p (2).

Constraint (3) expresses that a virtual machine has to be placed on exactly one physical machine.

Constraint (4) models that the total memory used by the virtual machines that are grouped together on a physical machine, minus the shared memory, may not exceed the total memory of the physical machine p_{mem}.

Constraint (5) can be explained as follows: for each physical machine is calculated how many virtual machines that are hosted on this physical machine, contain a specific hash. The number of times the hash occurs on the VM v is subsequently aggregated.

The group of constraints (6) tries to implement the following rules:

$$S_h^p = \begin{cases} 0 & \text{if } F \leq 1 \\ 1 & \text{if } F \geq 2 \end{cases}$$

$$T_h^p = \begin{cases} 0 & \text{if } S_h^p = 0 \\ F_h^p - 1 & \text{if } S_h^p = 1 \end{cases}$$

In KVM terms: when a page is being shared (S_h^p), then the sharing amount for that page is T_h^p, which is the frequency minus one ($F_h^p - 1$).

Note that the decision variables implicitly need extra constraints to limit the domain (boolean/int/int+).

4.1.2 Maximizing the shared pages

In an attempt to speed up execution time, the amount of unique shared pages per physical machine can be used instead of the total sharing pages. As a consequence, this model doesn't incorporate the number of times a hash occurs on a virtual machine. It either occurs or not, rather than having multiple occurrences or none.

Effectively, this causes that the sharing and shared pages are equal in this model: $T_h^p = S_h^p$.

The modified model for this optimization problem can be obtained by removing constraint (5) and (6) and replacing them with:

$$S_h^p = \left(2 \leq \sum_{v \in V} \sum_{\substack{i \in H_v \\ h=i}} L_v^p\right) \qquad \forall p \in P, \forall h \in H$$

This is a simplified constraint, in which the nested equation evaluates to a boolean value. Effectively, this constraint can be modeled by using the Big M-method:

$$\begin{aligned} F_h^p &= \sum_{v \in V} \sum_{\substack{i \in H_v \\ h=i}} L_v^p \quad \forall p \in P, \forall h \in H \\ F_h^p &\geq 2S_h^p \quad \forall p \in P, \forall h \in H \\ F_h^p &\leq 1 + M \cdot S_h^p \quad \forall p \in P, \forall h \in H \end{aligned}$$

Because there's only a notion of unique shared pages, there may be more deduplicated pages than is actually being optimized for. Constraint (4) may be modified to include a coefficient for S^p that accounts for these deduplicated pages.

$$\left(\sum_{v \in V} L_v^p \cdot v_{mem}\right) - q \cdot 4S^p \leq p_{mem} \qquad \forall p \in P$$

During tests, it was found that the ratio between unique shared pages and total sharing pages was $\frac{1}{2.97}$ for each virtual machine, tested for 78 combinations of 13 virtual machines running both Windows and a variety of Linux. In this case, coeffient q could hold a value of 3.

4.1.3 Greedy fat-first

One of the practical approaches explored iterates over all VM intersections to determine which pair offers the most overlap as a starting point. This pair is allocated on the biggest physical machine that is available. For this physical machine a new hash table is created that contains all hash-values of the VMs that are deployed there. The allocated set of VMs is then extended in a greedy fashion by iterating over the allocatable VMs and extending it with the one that shows the most overlapping pages. This process is looped till there is no VM left to allocate or the physical machine is provisioned to its maximum.

As the extending loop checks all pages of a potential VM with the hash table of the physical machine we reduce the amount of look-ups to that of the number of pages of the virtual machine. Since the amount of allocatable VMs drops every time a VM is allocated to a physical machine, the complexity also drops throughout the process as less VMs have to be iterated over.

The following loops define the steps taken in the greedy fat-first approach:

- Create iterable pairs of all VM combinations
- Loop till all VMs are deployed
  - Get the biggest host and the VM-pair with the most overlap
  - Loop till the physical machine is fully provisioned
    - Extend the VMs on this host with the most overlapping VM

The greedy approach is fairly simple and cheap to execute as the amount of look-ups is limited and decreasing throughout the process, though it may not result in the most optimal placement but rather an approximation.

The algorithm can be constructed given the following variables:

p	: Physical machine
v	: Virtual machine
e	: Designated pair of VMs
E	: All pair combinations of $Allocatable$
$Deployed$	: Set of deployed VMs p (KiB)
New	: Deployment scenario for machine H
$Allocatable$	: Set of all migratable VMs

Algorithm 1 Greedy fat-first

```
E = Iterpairs(Allocatable)
while Deployed != Allocatable do
    p = maxmem(hosts)
    e = maxoverlap(E) ∉ Deployed
    Deployed.add(e)
    while Deployed.mem < p.mem do
        for i ∈ Allocatable AND i ∉ Deployed do
            if getoverlap(i, Deployed) ≥ v then
                v = i
            end if
        end for
        Deployed.add(v)
    end while
end while
```

4.1.4 Maximizing the overlapping pages between VM pairs

The previously explained model maximizes the number of shared and sharing pages across all virtual machines. The newly proposed model looks instead at the number of overlapping pages between VM pairs. There is no longer notion of which hashes are present on a virtual or physical machine, only how many overlaps there are.

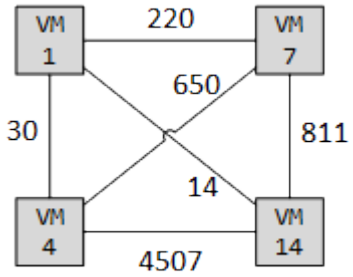

Figure 2: Full-mesh of VM pairings, with weighted edges for the overlap.

As shown in figure 2, the set of VMs (4,7,11) share a high amount of pages, which would cause them to be placed together using this optimization model.

Data:

P	: set of physical machines	V	: set of virtual machines
p	: a specific physical machine	v	: a specific virtual machine
p_{mem}	: memory of p (KiB)	v_{mem}	: memory of v (KiB)
H	: set of all hashes	H_v	: set of hashes on v
h	: a specific hash		
E	: set of all pairs of virtual machines		
e	: a pair of virtual machines		
$e_{1,2}$	: the individual VM in the pair		
O_e	: number of overlapping hashes between a VM pair		

Decision variables:

L_v^p : if a virtual machine v will be placed on physical machine p (boolean)
B_e^p : if the physical machine p hosts both machines in a VM pair e (boolean)
S^p : how many hashes are overlapping on the physical machine p (int+)

Objective:

$$max \sum_{p \in P} S^p \tag{7}$$

Subject to:

$$\sum_{p \in P} L_v^p = 1 \qquad \forall v \in V \tag{8}$$

$$\left(\sum_{v \in V} L_v^p \cdot v_{mem} \right) - 4S^p \leq p_{mem} \qquad \forall p \in P \tag{9}$$

$$\sum_{e \in E} B_e^p * O_e = S^p \qquad \forall p \in P \tag{10}$$

$$B_e^p = (L_{e_1}^p + L_{e_2}^p = 2) \qquad \forall p \in P, \forall e \in E \tag{11}$$

The total number of overlapping pages is maximized. This total is calculated as illustrated by the first part of the equation (12), while the second part (13) shows how this was calculated for the previous model.

$$|H_1 \cap H_2| + |H_2 \cap H_3| + |H_1 \cap H_3| \tag{12}$$

$$\geq$$

$$|(H_1 \cap H_2) \cup (H_2 \cap H_3) \cup (H_1 \cap H_3)| \tag{13}$$

The proposed model requires that overlaps between VM-pairs are calculated beforehand. This causes higher pre-processing times, but it pays off largely in solving time as it reduces the generated number of constraints.

The cost of calculating overlaps between a VM-pair is $O(min(|H_1|, |H_2|))$, where H_1 and H_2 are sets of hashes. One of the sets has to be looped completely, but lookups in the other set can be done in $O(1)$. The complexity of calculating the combinations of all VM-pairs is roughly given by $O(\frac{1}{2} \cdot |V| \cdot |H_v| \cdot (|V| - 1)) \approx O(|V|^2 \cdot |Hv|)$.

The constraint for the VM location (8), memory limit (9), and the objective for maximizing the shared pages (7) are unchanged from the first model.

The number of shared pages on a physical machine (10) can be modeled as the total of overlapping pages between all VM pairs that are hosted on a physical machine.

The last constraint (11) expresses that when a VM pair e occurs on a physical machine p, both e_1 and e_2 have to be placed on machine p. If needed, this simplified constraint can be modeled by using a logical AND as follows:

$$\begin{aligned} L^p_{e_1} &\geq B^p_e && \forall p \in P, \forall e \in E \\ L^p_{e_2} &\geq B^p_e && \forall p \in P, \forall e \in E \\ L^p_{e_1} + L^p_{e_2} &\leq B^p_e + 1 && \forall p \in P, \forall e \in E \end{aligned}$$

This is a rather fast model in comparison to the earlier mathematical models (comparable in speed to the greedy algorithm), although it may only deliver an approximation of the desired optimal configuration.

4.2 Distributed

Instead of relying on a central node to direct the maximization process we envision a distributed approach where every physical host can optimize the placement of the local virtual machines. By spreading the hash tables between the physical machines the load can be shared.

4.2.1 Distributed one-by-one approach

This method describes a theoretical approach to letting an environment converge to an optimized deployment of virtual machines in a distributed fashion. The following steps are taken by every node:

1. Each physical machine calculates its local memory density. The local density can be seen as the number of locally sharing pages, that factors in the total available memory. The average of the local density for all physical machines is what we call the global memory density.

2. A physical machine now picks one of its local virtual machines, either randomly or through means of a heuristic that selects a machine that is likely placed better elsewhere.

3. Other physical machines are queried to see if the selected VM can better be placed there. This can be determined by comparing the global memory density before and after a supposed acceptance of said VM.

4. A more sophisticated model could propose a VM trade, where physical machines can exchange VMs for another. This is particularly helpful in cases where a transfer can't happen because of memory constraints, although a better configuration is possible.

5. The migration of the VM happens and memory density should have increased. The process can now be repeated.

This approach is particularly useful in VM deployments that are already almost optimally placed. The algorithm could actively run and relocate VMs to try and keep the configuration in an optimal state. It could also be of benefit to deploy new VMs where they would profit the most of memory deduplication.

4.3 Optimizations

The previously discussed algorithms can be further improved upon, especially when dealing with larger environments. In these cases it may be an option to split the environment into multiple smaller groups, decreasing the complexity of the processing. These groups could be made on the physical level, grouping together a number of hosts, or at the virtual machine

level, grouping virtual machines based on higher level intelligence like the operating system or the type of service they offer.

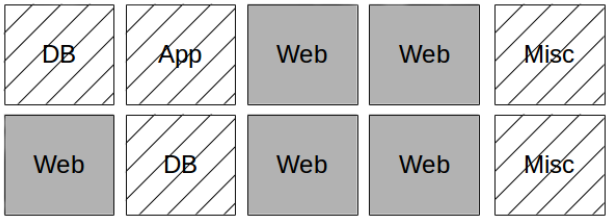

Figure 3: Grouping based on higher level intelligence

We suspect that running the maximization algorithms on these smaller groups could balance the cost to profit ratio opposed to maximizing the entire environment at once. However, further research is required to prove this.

As an alternative data structure, further speed improvements may be made by using a bit vector: every page can be represented as a bit in the vector. When the bit is set, the page occurs at least once on the VM. Since the current hashing algorithm (xxHash) produces a 32-bit hash, a Bloom filter could be used to save memory space. The use of Bloom filters as data structure for page hashes is further documented by Wood et al. [4]. The efficient approximation of Bloom filter intersections has been investigated by Swamidass and Baldi [13], which could be used to calculate estimated overlap between VM pairs.

As last remark, the proposed models currently don't consider the original VM configuration. It would be interesting to connect a cost to the actual relocation of VMs.

5 Conclusion

During this research we tried to apply the principle of finding duplicate memory pages with memory hashes cluster-wide, saving the most amount of memory through optimal deployment of virtual machines. We looked into a universal approach that extracts the memory and hashes it with the very fast xxHash algorithm, discussed in section 3.2. This data is filtered and structured in hash tables on the physical machines, after which it can be exchanged compactly over the network. A centralized or distributed algorithm then takes care of the optimization process (section 4).

We conclude that finding an optimal configuration is not scalable using an exact approach, as explained in section 4.1. Therefore, we proposed a number of algorithms that optimize the deployment of virtual machines by approximating the optimal solution.

During the time frame of this project we weren't able to test the algorithms for large clusters, leaving us without clear results to compare the mechanisms or draw conclusions on the optimality of the proposed algorithms (how the approximation compares to the optimal solution).

We suspect that the complexity in very large environments may still become problematic as some of the calculations are exponential in complexity. A technique we suggest to partially resolve this problem is to split the environment into deduplication groups, formed by either a number of physical hosts or closely related virtual machines (section 4.3).
A number of other ideas to extend on the subject of memory deduplication and VM placement are described in the future research section 6.

6 Future research

The optimization of VM placement on physical machines can hold additional advantages that are not only limited to memory deduplication. For instance, the spreading of resource intensive virtual-machines could improve the performance of the entire environment. More logic in the previously discussed algorithms could also make sure that other resources, like CPU- and I/O usage, are not overprovisioned too much, preventing a negative impact on performance after a VM-migration.
The CPU cache also holds frequently accessed memory locations. When VMs with overlapping memory pages that are frequently used are put on the same host machine, this will likely result in more CPU cache hits, ultimately reducing processor cycles.

As proposed in section 4.3, it would be interesting to see how higher level intelligence could be used to create deduplication groups, which could improve the cost to profit ratio.

The error margin of the proposed approximation algorithms remains largely untested. More research is needed to determine how the algorithms compare to each other.

7 References

[1] B. Posey. Virtualization: Optimizing Hyper-V memory usage. http://technet.microsoft.com/en-us/magazine/hh750394.aspx, 2012.

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the linux symposium*, 2009.

[3] K. Miller and F. Franz and T. Groeninger and M. Rittinghaus and M. Hillenbrand and F. Bellosa. KSM++ Using I/O-based hints to make memory-deduplcation scanners more efficient. http://www.dcs.gla.ac.uk/conferences/resolve12/papers/session3_paper2.pdf, 2012.

[4] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D 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*. ACM, 2009.

[5] Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. Enacloud: An energy-saving application live placement approach for cloud computing environments. *2013 IEEE Sixth International Conference on Cloud Computing*, 0, 2009. doi: http://doi.ieeecomputersociety.org/10.1109/CLOUD.2009.72.

[6] Wikipedia. Page (computer memory) — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Page_(computer_memory)&oldid=600068828, 2014.

[7] Bob Jenkins. lookup3.c. http://burtleburtle.net/bob/c/lookup3.c.

[8] xxHash: Extremely fast non-cryptographic hash algorithm. https://code.google.com/p/xxhash/.

[9] SMHasher: Test your hash functions. https://code.google.com/p/smhasher/wiki/SMHasher.

[10] Standard types Python, Mapping Types - Dict. http://docs.python.org/2/library/stdtypes.html#typesmapping.

[11] D. Magenheimer, Oracle. Memory Overcommit... without the commitment. https://oss.oracle.com/projects/tmem/dist/documentation/papers/overcommit.pdf, 2008.

[12] Edward G Coffman Jr, Michael R Garey, and David S Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996.

[13] S Joshua Swamidass and Pierre Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling*, 47(3), 2007.

Appendix

A Definitions

Shared pages: the amount of unique pages that are being shared, regardless of the amount of times a certain page is being deduplicated.

Sharing pages: the shared pages, including how many times each page is being deduplicated.

B Code repository

Implementations for the proposed centralized optimization models have been released under the Apache License 2.0, and can be found on the code repository on GitHub: http://github.com/c3c/CSM.

Method	**File on code repository**
Maximizing the sharing pages	`max_sharing.mod` (CPLEX model)
Maximizing the shared pages	`max_shared.mod` (CPLEX model)
Greedy fat-first	`greedy.py` (Python implementation)
Maximizing the overlapping pages between VM pairs	`pairoverlap.py` (Python implementation interfacing Gurobi)