Algorithms for Automated Live Migration of Virtual Machines

Mattias Forsman, Andreas Glad, Lars Lundberg, Dragos Ilie*

Blekinge Institute of Technology
Karlskrona, Sweden

Abstract

We present two strategies to balance the load in a system with multiple virtual machines (VMs) through automated live migration. When the *push* strategy is used, overloaded hosts try to migrate workload to less loaded nodes. On the other hand, when the *pull* strategy is employed, the light-loaded hosts take the initiative to offload overloaded nodes.

The performance of the proposed strategies was evaluated through simulations. We have discovered that the strategies complement each other, in the sense that each strategy comes out as "best" under different types of workload. For example, the *pull* strategy is able to quickly re-distribute the load of the system when the load is in the range low-to-medium, while the *push* strategy is faster when the load is medium-to-high.

Our evaluation shows that when adding or removing a large number of virtual machines in the system, the "best" strategy can re-balance the system in 4–15 minutes.

Keywords: live migration, virtualization, load balancing

1. Introduction

The informal interpretation of virtualization is that of a mechanism for running concurrently several operating system (OS) instances on a single computer node. We call these nodes physical machines (PMs). A typical virtualization architecture is shown in Figure 1. The *hypervisor* is the core component of a virtualization platform. The main responsibility of the hypervisor is to delegate computer hardware to virtual-machine monitors (VMMs)[1]. Each VMM is responsible for providing hardware abstraction for exactly one running virtual machine (VM). The VM typically hosts a *guest OS*. By running multiple VMs simultaneously on a PM, the hardware can be used more efficiently.

Several components of a system, such as CPU, memory, network and storage, can be virtualized. The main focus here is on CPU virtualization.

Many hypervisors are able to migrate VMs from one host to another. There are three different approaches to migrate a VM. When *cold migration* is used, the guest OS is shut down, the VM is moved to another host and then the guest OS is restarted there. *Hot migration* suspends the guest OS instead of shutting it down. The guest OS is resumed after the VM is moved to the destination host. The benefit of hot migration is that applications running inside the guest OS are not restarted from scratch. Some platforms offer a feature called *live migration*. This feature allows a VM to be moved from one host to another while the guest OS is running. Live migration reduces the downtime dramatically for applications executing inside the VM. It is highly valuable if the migrations can be performed automatically, without

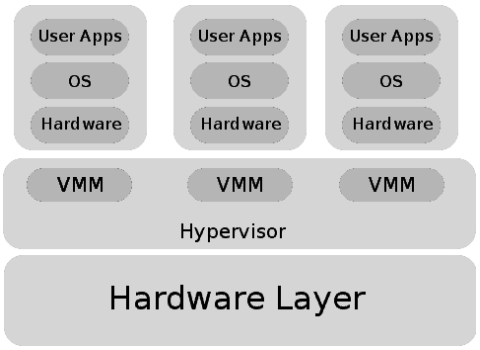


Figure 1: Virtualization architecture

*Corresponding author. Tel.: +46 455 38 58 71.
Email addresses: mattias.forsman@gmail.com (Mattias Forsman), andreas.glad86@gmail.com (Andreas Glad), lars.lundberg@bth.se (Lars Lundberg), dragos.ilie@bth.se (Dragos Ilie)

[1]The terms hypervisor and VMM are treated as equivalent in most literature. However, in [1], the terms refer to two separate entities, as described here.

the involvement of a human operator. We call this type of unattended operation *automated live migration*.

In this paper we investigate two distributed algorithms for automated live migration. The goal of the algorithms is to achieve a load-balanced system through node cooperation. The performance of these algorithms is evaluated by simulations.

The remainder of the paper is organized as follows. Section 2 provides a brief introduction to virtualization and live migration techniques. Related work is presented in Section 3. Details about the migration strategies are available in Section 4. Section 5 provide a walk-through for the *pull* and *push* algorithms. The simulation testbed as well as simulation scenarios are described in Section 6. Our simulation results are presented in Section 7. Finally, we share our conclusions in Section 8.

2. Live Migration

In live migration, the state of the guest OS on the source host must be replicated on the destination host. This requires migrating processor state, memory content, local storage and network state [2]. We focus on migration of CPU state and memory content.

The tricky part of live memory migration is that VM memory pages can be modified on the source host before the transfer to the destination host is complete. The dominant technique to handle this problem is called *pre-copy* and is the approach used by Xen [3, 2], KVM [4], and VMware [5]. The memory pages of a virtual machine are copied in iterations. In the first round all pages are copied while in the following rounds only modified (dirty) pages are moved. The modified pages are tracked via a *dirty bitmap* maintained by the hypervisor. The *stop-and-copy* phase begins after pre-copy. Its purpose is to halt the VM on the source host and transfer the remaining dirty pages. At the end of this phase the VM can be resumed on the destination host.

VM migration has important applications in dynamic resource management for cloud-based systems and large data centers. In these environments a group of VMs can begin competing for resources provided by a single PM. A *hot spot* occurs when the performance of VMs degrades because the PM is unable to respond to the resource demand. The opposite situation, when resources on a PM are underutilized, is termed *cold spot*. This happens when the running VMs consume only a tiny fraction of the resources provided by the hosting PM. Both hot spots and cold spots can be handled by moving around one or more VMs. In *hot spot mitigation* the selected VMs are moved to a less loaded PM. *Server consolidation* is the strategy to handle cold spots by regrouping VMs from lightly-loaded hosts to a smaller subset of PMs, thus freeing up the remaining PMs for resource-hungry VMs [6].

Load-balanced systems are desirable for several reasons, such as to avoid large discrepancies between the level of service afforded to various VMs from the same service class and to keep an even ambient temperature to reduce cooling cost [7, 8].

Live migration enables hot spot mitigation and server consolidation with minimal disturbance to applications running inside the migrating VMs.

3. Related Work

Several research efforts have improved live migration with respect to the volume of data transfered between hosts. For example, Jin *et al.* [9] reduced the migration time by compressing the memory pages that are sent over the network during Xen's *pre-copy* and *stop-and-copy* stages. Their solution exploits inherent redundancy in memory areas (*e. g.*, large blocks of memory consisting of zero bytes or identical memory pages belonging to multiple guest OSes running on the same host) to achieve high compression ratios. The cost function used for our migration strategies, described in Section 4.1, is proportional to the memory size of the VM. Therefore our migration strategies are biased against migrating large VMs. However, a VM with a large amount of memory is likely to contain more similar pages than a small VM. If the compression technique from [9] could be integrated with our cost function, the probability of selecting large VMs for migration can increase.

In [10], the authors present a technique that minimizes the data volume sent over the WAN during live migration by avoiding to send redundant memory pages. When a redundant page is encountered only a hash is sent to destination host. The host uses the received hash to lookup and replicate a previously received memory page. The effect is similar to that of compressing the pages, as described above. However, the advantage in this case is that there is typically lower overhead in computing hashes than in compressing data.

In yet another approach to reduce migrated data, the authors of [11] presents a live migration technique that combines checkpoints and traces for non-deterministic events. The results presented by the authors indicate that this approach greatly reduces migration overhead while still achieving better average performance for downtime and total migration time. The advantage of this approach is that the volume of trace data (*i. e.*, the

events that change CPU state and memory contents) is smaller than the volume of data transferred in the typical pre-copy approach described in Section 2. However, this approach is less successful in multi-core/multi-processor environments.

Some work aims at reducing the power consumption in data centers. In [12], the authors present a framework that reduces the power consumption by dynamically re-mapping VMs to PMs at runtime. Their framework is a centralized solution, unlike the distributed approach presented here. Interestingly, their re-mapping algorithm uses an auction model in similar vein to our migration strategies presented in Section 4.1.

In [13], the authors propose an energy efficient live migration system. One of their goals is to find underutilized PMs that can be powered down. If the PM that is to be powered down currently hosts VMs, these virtual machines are forced to migrate. This is also a centralized solution. Their scheme seems targeted towards load balancing and server consolidation, whereas our approach deals with load balancing and hotspot mitigation.

In [14], the authors propose a framework to enable live migration between different kinds of hypervisors. The authors used an actual implementation of their framework to performed live migrations between Xen [3] and KVM [4]. The framework uses the *pre-copy* technique for live migration. This means that the cost model presented in Section 4.2, which is also based on *pre-copy*, can be used to implement migration strategies between different kind of hypervisors.

In [15], the author describes a couple of load-sharing schemes. Among these are the *sender* and the *receiver* schemes. In the sender scheme, the overloaded host takes the initiative to share its load. In the receiver scheme, the underutilized hosts takes the initiative and tries to find work. As shown below, our *push* strategy resembles the *sender* scheme while the *pull* resembles the *receiver* scheme.

The authors of [16] use Shannon's concept of information entropy to detect load imbalance between nodes. Another paper utilizing Shannon's information entropy to detect imbalance in a cluster storage system is [17]. Our model for load distribution in a system is based on these two papers.

In [18], the authors characterize which parameters affect a migration the most. They establish that available bandwidth and page dirty rate are factors with a great impact on total migration time and downtime. These parameters are used by our cost model for live migration.

Mishra *et al.* [6] provide an overview of resource management issues in a cloud environment that are solved through VM migrations. We include this article here because it provides a clear and concise description of tradeoffs for various migration techniques, which was helpful to us in designing our migration strategies.

Few attempts have been made to dynamically map VMs to PMs and use the mapping to perform migrations automatically. One article related to this is [19], where a framework for automated live migration is presented. The authors measured both the PMs' and the VMs' resource utilization to decide when a migration is necessary. The authors use CPU utilization (*i. e.*, the fraction of time the CPU is busy) as a metric to determine the occurrence of hotspots. We believe that the CPU load metric described in Section 4 is better at describing the utilization of a system because it reflects the number of processes competing to use the CPU.

4. Automated load-balancing

We present a solution for system load-balancing through automated live migration. Our solution handles the following sub-problems:

i) detect hot spots (*i. e.*, overloaded PMs),
ii) identify PMs that can take over workload from the hot spot, without creating a new hot spots,
iii) avoid cold spots,
iv) minimize migration costs.

Most of these sub-problems can be quite difficult to solve optimally, as illustrated in many of the references mentioned in Section 3(*e. g.*, [12, 17, 19]). We used a pragmatic approach and made the following simplifications:

- a VM is the smallest workload unit that can be transferred between hosts,
- no more than one VM is transferred in a single iteration of our algorithm,
- PM's CPU load is the only hot spot indicator used.

A hot spot, in our approach, is defined as a host that has a CPU load that exceeds a specific utilization threshold λ.

Our notion of load is similar to the load metric used by the Linux OS, which denotes the number of processes competing for CPU at a given time. For a host with n cores or n processors, a load value l ($0 \leq l < n$) means that the running processes require less than the available CPU time. When the load value exceeds n,

some processes are unable to get a time slice from the scheduler and CPU run queues start to build up. Consecutive load values can exhibit a high degree of variability, which makes it difficult to discern load trends over longer time windows. Linux handles this problem by applying an exponentially weighted moving average (EWMA) function to the load data to smooth out large variations and obtain *load average* values [20].

We have developed two main migration strategies to handle hot spots. Under the *push* strategy, overloaded PMs attempt to migrate workload to less loaded PMs. When the *pull* strategy is used, underutilized PMs request workload from heavier loaded PMs. Each of these strategies is implemented through a communication protocol that allows PMs to exchange information about their CPU load and workload that can be shared.

A PM chooses the destination host and which VM to migrate based on the following selection factors: average post-migration CPU load on the destination PM, cost of the migration, and expected load distribution in the system after the migration. In addition, checks are in place to ensure that the destination PM has enough free memory to accomodate the migrated VM [21].

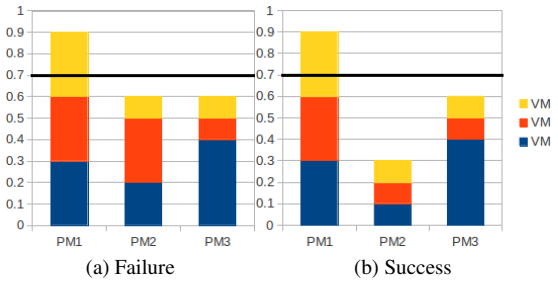
Selection factors and migration strategies are described in detail in the following subsections.

4.1. Migration strategies

Our migration strategies can be modeled as a special type of multi auction selection process. The process is based on the first-price sealed-bid auction type [22], but with several items being auctioned and only one of them being sold.

In the case of the *push* strategy, an overloaded PM plays the role of the seller. The VMs hosted on this PM constitute the auctioned items. The PM multicasts information about these items to the potential buyers (*i. e.*, the other PMs in the system). The information shared with the buyers contains the characteristics of each VM, such as current load and memory usage. The buyers may bid on one, several, or all auctioned VMs. A bid consists of the PM's available resources as well as of the PM's past resource usage. The seller selects the best candidate from the bids and performs the migration. Only one out of all bids will get selected and a single VM is migrated. When the *pull* strategy is used, the seller is an underutilized PM that advertises free resources. The other PMs in the system are potential resource buyers that bid with the characteristics of their VMs as described above. The seller decides which bid it accepts and initiates the VM migration.

The first-price sealed-bid auction model is suitable to use with the the *push* and *pull* strategies because the seller does not know in advance how the auctioned item is "valued" by the buyers and also because each bidding PM is unaware of the bids placed by the other PMs [23]. In this case, the "value" is the resources that sellers offer in the *pull* strategy and the VMs they are willing to release in the *pull* strategy.



(a) Failure (b) Success

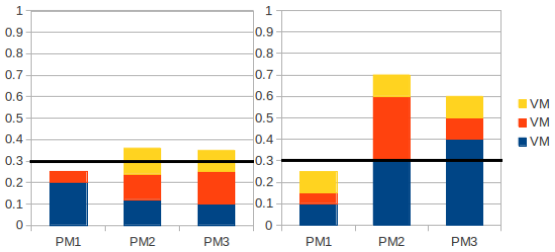
Figure 2: Auction outcomes under the *push* strategy for overloaded host PM1

In the *push* strategy, an auction will fail if none of the buyers has sufficient resources to hold any of the seller's VMs. In the scenario illustrated in Figure 2a, the horizontal line located at 0.7 on the y-axis indicates the load threshold λ. A PM is considered overloaded if its load exceeds the threshold value, as is the case of PM1. However, a migration from PM1 to PM2, or from PM1 to PM3, will overload the destination host instead. Therefore, the auction will fail.

A successful *push* scenario is shown in Figure 2b. Here, PM1 is still overloaded and needs to migrate one of its VMs. By migrating a VM to PM2, the load on both PM1 and PM2 will decrease below the threshold and the system will attain a more balanced state.

Figure 3 depicts the *pull* strategy. In this case, when the load on a PM drops below the threshold value, the host considers itself underutilized and initiates an auction to sell its free resources. The goal of the action is to move workload from a potential buyer to the seller, but without causing the buyer's load to drop below the threshold. In Figure 3a we can see that the underutilized host PM1 is unable to acquire any VMs from PM1 or PM2. A success scenario is illustrated in Figure 3b where PM1 can acquire a VM from either PM2 or PM3, with the exception of the blue VM on PM3.

Fixed threshold values can lead to situations where the system is unable to react to obvious load imbalance. For example, imagine that we have a *push*-strategy system in the state shown in Figure 4. There, PM1 and PM2 are slightly below the high-load threshold while PM3 and PM4 experience light load. Clearly, workload from PM1 and PM2 could be transferred to PM3 and PM4,



(a) Failure (b) Success

Figure 3: Auction outcomes under the *pull* strategy for underutilized host PM1

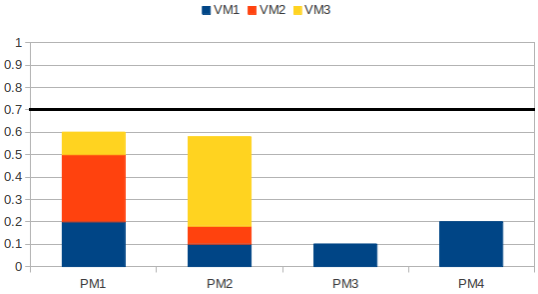


Figure 4: Imbalanced system due to fixed threshold value

but because the thresholds are not reached the system will continue in an unbalanced state.

A fixed threshold value can also be damaging for the *pull* strategy. Consider the case where the load of a host is just above the low-load threshold (*e. g.*, PM2 or PM3 in Figure 3a). If other PMs in the system become overloaded the system will remain in an unbalanced state because the host, although lightly loaded, will not offer its resources for auction. In this case, not only are resources poorly utilized, but system services provided to customers may fall below acceptable levels resulting in violations of service level agreements [18, 24].

An adaptive threshold value is an obvious solution to the problems created by fixed thresholds. The threshold λ begins with an initial value and is gradually increased in the *pull* strategy, while being gradually decreased when the *push* strategy is used. Figure 5 depicts the adaptive threshold concept applied to a *push* scenario.

Whenever an auction fails, the threshold for the PM that started the auction is reset to its initial value. This approach is similar to how the TCP congestion control behaves when the congestion window is reset whenever a timeout happens [25]. Periodically, each PM increases (or decreases) their threshold value by a constant amount. Each period length is computed as 5 s plus a random factor uniformly distributed on the set

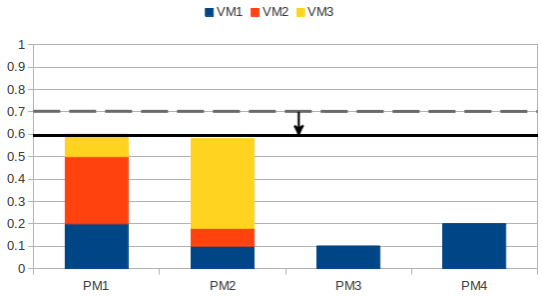


Figure 5: Balanced system due to active threshold value

$[-2, 2]$ s to avoid synchronization with other PMs.

We have also introduced a binary exponential *back-off* algorithm, similar to the one used for local area networks [26], to ensure conservative use of CPU and network resources when auction attempts fail repeatedly. The algorithm is always initiated by the seller. If the auction fails, the seller waits a random amount of time before performing another auction. The range from which the random time is drawn increases exponentially with each try, but is truncated at 120 s.

4.2. *Selection factors for VM migration*

A seller uses three factors when selecting which VMs to migrate:

- buyer's load after migration,
- expected distribution of load in the system,
- migration cost.

We use an EWMA algorithm to estimate the buyer's average load after migration. The EWMA is useful in smoothing out variations in the average load. Without it, short-term load variations would generate unnecessary migrations. Although the EWMA is unable to forecast trends, it is easy to compute and requires only the previous estimate and a new sample to produce a new value [27].

Each PM samples its CPU load periodically (*i. e.*, once per second) as well as the load of the VMs it hosts. If we denote the n-th load sample by l^n, then the estimated load for the $(n+1)$-th sample, $\hat{l}^{n+1}$, is

$$\hat{l}^{n+1} = \begin{cases} \alpha l^n + (1-\alpha)\hat{l}^n, & \text{if } n > 6 \\ \dfrac{\sum_{i=1}^{5} l^i}{n}, & \text{if } n \leq 5 \end{cases} \quad (1)$$

The second case in Equation 1 is used to bootstrap the algorithm by computing the initial estimates $\hat{l}^1, \ldots, \hat{l}^5$.

The actual EWMA algorithm, shown in the first case in Equation 1, computes the remaining estimates.

The parameter α decides the influence of the current load sample over the past samples in computing the estimate. We have set $\alpha = 0.2$ as suggested by [28]. This smoothes out high variability between consecutive samples, which is why we prefer the estimate over using the actual load values.

In the *push* strategy, bids from buyers include PM load estimates. Similarly, when the *pull* strategy is used, bids from buyers include load estimates for their VMs. The seller uses the estimates to compute the expected load of the buyer and the load distribution in the system if the bid is accepted.

We modeled the load distribution using the concept of *entropy*, similarly to how it was done in [16] and [17]. The entropy function has several properties [17] that are useful in capturing the notion of load distribution:

i) it takes on values in the range 0.0–1.0,
ii) it approaches 0.0 when a single element is present in the system,
iii) it approaches 1.0 for a perfectly load-balanced system,
iv) it increases when an element with zero load is added to the system.

Obviously, the goal is to maximize the entropy metric when selecting a VM for migration.

The model takes as input the load values of all PMs and outputs the entropy value. More formally, let $l_i \geq 0$, be the load of i-th PM in a system with n PMs. The normalized load $0 \leq p_i \leq 1$ with respect to the total load in the system is

$$p_i = \frac{l_i}{\sum_{i=1}^{n} l_i} \tag{2}$$

If we let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of the normalized load values p_i, the entropy of P can be defined

$$\mathrm{H}(P) = -\sum_{i=1}^{n} p_i \log p_i \tag{3}$$

It is useful if $\mathrm{H}(P)$ is also normalized because values in the range 0.0 – 1.0 are easy to interpret. As shown in Equation 4, we normalize $\mathrm{H}(P)$ with respect to the maximum entropy value $\mathrm{H}(P)_{max} = \log n$, which corresponds to the case where $p_i = 1/n$, for all i [16]. We refer to $\mathrm{E}(P)$ as the *normalized entropy* metric[2].

[2]It should be noted that in [17] the entropy metric $\mathrm{F}(P)$ is defined as the complement of our entropy function, that is $\mathrm{F}(P) = 1 - \mathrm{E}(P)$. With this definition, one strives to minimize $\mathrm{F}(P)$ to obtain a load-balanced system. Certain optimization problems can be solved easier when the goal is to minimize the objective function.

Table 1: Parameters for the cost model, adapted from [29]

V_{mem} (MB)	Total amount of VM memory
V_i (MB)	Size of VM memory to be migrated during round i, $V_0 = V_{mem}$
V_{mig} (MB)	Total amount of network traffic required to perform a migration
T_{mig} (s)	Total migration time when the VM is still operational
T_{down} (s)	Total downtime when the VM is not operational
V_{thd} (MB)	Threshold value for the remaining memory due to dirty pages to be transferred in the last round
R (MB/ s)	Available bandwidth that can be used for the migration, *i. e.*, the memory transmission rate
D (MB/ s)	Page dirty rate during migration
W_i (MB)	Writable working set, frequently changed memory pages in round i

$$\mathrm{E}(P) = \frac{\mathrm{H}(P)}{\mathrm{H}(P)_{max}} = \frac{\sum_{i=1}^{n} p_i \log p_i}{log\frac{1}{n}} \tag{4}$$

To estimate the migration cost of a VM, we use an algorithm proposed in [29]. The algorithm implements a performance model which is built on the assumption that live migration uses the *pre-copy* technique. This is a reasonable assumption since the *de-facto* standard hypervisors (*i. e.*, Xen, KVM and VMware) in fact use pre-copy for migration [3, 2, 4, 5].

All parameters used by the performance model are summarized in Table 1 and the actual algorithm is shown in Algorithm 1. The model provides three cost indicators: migration time T_{mig}, estimated service downtime T_{down}, and traffic volume due to migration V_{mig} (line 1, Algorithm 1). The migration time is the time required by the *pre-copy* phase of migration. During this time, the guest OS is still running and services running on top of it may suffer from performance penalties. Downtime is the time required by the *stop-and-copy* phase when the VM is suspended and guest OS services are not available.

The algorithm uses the V_{mem} input parameter, which is the memory size of the VM, to compute T_0, the duration of the first *pre-copy* round. The duration of a round i is decided by V_i, the volume of data to be transferred

Algorithm 1 Cost model algorithm, adapted from [29]

1: **Input:** V_{mem}, V_{thd}, D, R **Ouput:** V_{mig}, T_{mig}, T_{down}
2: let $V_0 \leftarrow V_{mem}$ /* data moved in the first round */
3: **for** $i = 0$ to max_rounds **do**
4: $T_i \leftarrow V_i/R$
5: $\gamma \leftarrow \omega T_i + \xi D + \psi$
6: $W_{i+1} \leftarrow \gamma T_i D$
7: $V_{i+1} \leftarrow T_i D - W_{i+1}$
8: **if** $V_{i+1} \leq V_{thd}$ **or** $V_{i+1} > V_i$ **then**
9: $V_{i+1} \leftarrow T_i D$ /* data moved in the last round */
10: $T_{i+1} \leftarrow V_{i+1}/R$
11: $T_{down} \leftarrow T_{i+1} + T_{resume}$
12: **break**
13: **end if**
14: **end for**
15: $V_{mig} \leftarrow \sum_{i=0}^{max\_rounds} V_i$
16: $T_{mig} \leftarrow \sum_{i=0}^{max\_rounds} T_i$

during that round, and R, the achievable transmission rate between the source host and the destination host. When $i = 0$, $T_0 = V_{mem}/R$ (line 1–4, Algorithm 1).

Computing the volume of data in subsequent rounds is a bit more complicated. Some memory pages that change very frequently are part of what is called the writable working set (WWS). Hypervisors should avoid copying these pages during *pre-copy* to conserve resources, and instead transfer them during the *stop-and-copy* phase [2]. Liu *et al.* [29] assume that the WWS in round i is proportional to V_i, the memory pages dirtied in the previous round

$$W_i = \gamma V_{i-1}, \tag{5}$$

where γ is related to the dirtying rate D and the round duration T_i (line 6, Algorithm 1). The authors described this relationship through a linear model

$$\gamma = \omega T_{i-1} + \xi D + \psi, \tag{6}$$

where $\omega = -0.0463$, $\xi = -0.001$, and $\psi = 0.3586$ (line 5, Algorithm 1). The coefficient values were estimated by training the model with the DaCapo benchmark [29].

Using the arguments above, the volume of data to be transferred in round i is (line 2 and line 7, Algorithm 1)

$$V_i = \begin{cases} V_{mem}, & \text{if } i = 0, \\ D \times T_{i-1} - W_i, & \text{otherwise,} \end{cases} \tag{7}$$

which, in turn, can be used to compute the duration of round i (line 10, Algorithm 1):

$$T_i = \frac{V_i}{R} = \frac{D \times T_{i-1} - W_i}{R} \tag{8}$$

The algorithm ends the *pre-copy* phase when one of the following conditions is met (line 3 and line 8, Algorithm 1):

- the remaining volume of dirty pages reaches or decreases below the threshold V_{thd},
- the remaining volume of dirty pages exceeds the volume from previous round,
- the maximum number of *pre-copy* rounds is reached.

At this point the algorithm can compute T_{mig}, the duration of the *pre-copy* phase, by summing together the duration of each round (line 16, Algorithm 1).

The duration of the *stop-and-copy* phase, T_{down} has two additive components: $T_n = V_{n-1}/R$, the time it takes to transfer the remaining amount of dirty pages in the final round n, and T_{resume}, the time it takes to resume the VM on the destination host (line 9–11, Algorithm 1). T_{resume} is assumed to be constant.

We define the total cost to migrate VM_i as

$$C_i = T_{mig} + T_{down} = T_{mig} + T_n + T_{resume}. \tag{9}$$

Here it is useful to summarize what has been described so far in this section. Buyers use Equation 1 to compute load estimates for their PMs and VMs. Seller use the estimates to discard bids that, if accepted, would cause the buyer to become overloaded or underutilized. Sellers also compute an expected entropy value for each VM in the remaining bids. The value serves as a metric for how well the load would be distributed if the corresponding VM is migrated. Additionally, sellers use Algorithm 1 together with Equation 9 to compute the migration cost for each VM in the remaining bids.

The aim of these three selection factors is to allow a seller to choose a VM for migration such that the load distribution in the system is improved while keeping the migration cost reasonable. We take a pragmatic approach in defining "reasonable" and discard all bids with a cost exceeding the average cost of all bids

$$C_{mean} = \frac{\sum_{i=1}^{n} C_i}{n}. \tag{10}$$

The remaining bids are sorted based on the entropy value and the candidate yielding the highest entropy is selected as the final candidate:

$$\text{VM}_{candidate} = \max(\text{E}(P)) \tag{11}$$

where $\text{E}(P)$ is defined in Equation 4.

Algorithm 2 Push strategy - Source node

```
1:  if Event then
2:    if Event = hotspot then
3:      for all PMs do
4:        Multicast hotspotMessage containing
          resource usage and list of hosted VMs
5:      end for
      /* Waiting for bids */
6:      while bids < PMs − 1
          and elapsedTime < maxTime do
7:        Receive bids
8:      end while
9:      for all bids do
10:       Calculate the utility for all VMCandidates
          and append to utilityArray
11:     end for
12:     for all utilites in utilityArray do
13:       Calculate migration cost
14:     end for
15:     Sort utilityArray based on utility
16:     Remove candidates that have a high cost
17:     Set winningBid to the highest value in
        utilityArray
18:     for all PMs do
19:       Multicast information about winningBid
20:     end for
21:     Listen for a confirmation message from
        the winning PM
22:     if none or negative response then
23:       Run backoff algorithm and exit
24:     else
25:       Run migration and exit
26:     end if
```

Algorithm 3 Push strategy (Continued) - Candidate node(s)

```
27:   else if Event = hotspotMessage then
28:     if resources are not locked then
29:       Lock resources
30:       for all VMs in hotspotMessage do
31:         if VM fits the available resources then
32:           Add VM to the array VMcandidates
33:         end if
34:       end for
35:       Reply with VMcandidates and resource
          usage to sender
36:     else
37:       Reply with an empty message
38:     end if

39:   else if Event = winningBidAnnouncement then
40:     if winner then
41:       Reply with acknowledgment
42:       Wait for the VM to be migrated
43:       Release resource lock
44:     else
45:       Release resource lock
46:     end if
47:   end if
48: end if
```

5. Algorithms

In this section we provide a thorough description of the algorithms for the *push* and *pull* strategies and conclude with an analysis of their computational complexity.

5.1. Push algorithm walk-through

When a PM becomes overloaded (line 2, Algorithm 2), it initiates an auction by multicasting its resource usage and a list of hosted VMs to the other PMs in the network (line 3–5, Algorithm 2). The overloaded host collects bids until all other PMs have replied or until the waiting time (*elapsedTime*) has exceeded its limit (*maxTime*) (line 6–8, Algorithm 2). Each bid contains a set of VMs (*VMcandidates*) from the auctioned list that would fit within the bidding PM's available resources (line 28–35, Algorithm 3). The overloaded node computes the load entropy (*utility*) and the cost due to each *VMCandidate* from the bids and stores these in the *utilityArray* (line 9–14, Algorithm 2). The best candidate from the *utilityArray* is sent as an announcement (*winningBid*) to all PMs (line 17–20, Algorithm 2). The winner must reply with an acknowledgment and wait for the migration to be initiated. All the other candidates will release their locks[3] (line 39–47, Algorithm 3) after receiving the winning announcement. If any unexpected errors occur, the backoff algorithm described in Section 4.1 will be triggered. Otherwise, the migration will be initiated (line 21–26, algorithm 2).

5.2. Pull algorithm walk-through

When a PM becomes underutilized (line 2, Algorithm 4), it initiates an auction by multicasting its resource usage to the other PMs in the network (line 3–5, Algorithm 4). The selling host collects bids until all other PMs have replied or until the waiting time

[3] A resource lock is a mechanism to prevent resources negotiated under an auction from being used in other concurrent auctions.

(*elapsedTime*) has exceeded its limit (*maxTime*) (line 6–8, Algorithm 4). Each bid lists all the VMs running on the bidding PM (line 34, Algorithm 5). The underutilized PM verifies each VM from the bids for its *suitability*. A suitable VM is one that fits within PM's available resources and does not make the bidding node underutilized if a migration would occur. The underutilized PM computes the load entropy (*utility*) for each VM from the bids as well as its migration cost and stores these in the *utilityArray* (line 9–16, Algorithm 4). As long as the *utilityArray* contains candidates, the PM will attempt to retrieve the VM with the highest utility from the bidding PM. If the attempt times out (*timeout*) or is denied by the buyer, the VM entry is removed from the *utilityArray* and the PM attempts to retrieve the second best candidate (line 19–27, Algorithm 4). If the attempt succeeds, the VM is migrated.

5.3. Algorithm complexity

If we assume that the maximum number of VMs on each PM will not exceed some (potentially large) constant value, the complexity of our five algorithms can be expressed in terms of n (the number of PMs). Modern datacenters contain a lot of PMs, and it is therefore important to consider the complexity of our algorithms, *i. e.*, how the execution times of the algorithms scale when the number of PMs increases.

In Algorithm 1 we do (in the worst case) *max_rounds* iterations, and since *max_rounds* was set to 27 in our case, the algorithm has constant complexity ($O(1)$). Algorithm 3 is executed by the hosts that are (potentially) prepared to receive a VM in the *push* strategy. This algorithm iterates over the list of VMs sent by the overloaded host, and since we assume that the maximum number of VMs on each PM will not exceed some constant value, Algorithm 3 has constant complexity ($O(1)$). Algorithm 5 contains no loops, and has also constant complexity ($O(1)$).

The complexity of Algorithm 2 is quadratic ($O\left(n^2\right)$), since the iterations in lines 9–11 potentially calculate the utility (normalized entropy) when moving each VM in the system, and the complexity for calculating the normalized entropy when moving one VM is linear in n (see Equation 4). The two loops in lines 3–5 and 6–8 both have linear complexity ($O(n)$). Since the utility array cannot exceed the number of VMs, the loop in lines 12–14 has linear complexity ($O(n)$). The sort operation in line 15 has complexity $O\left(n^2\right)$ or $O(n * \log n)$ depending on which sort algorithm we use. Finally, the loop in lines 18–20 has linear complexity ($O(n)$).

The complexity of Algorithm 4 is also quadratic ($O\left(n^2\right)$), since the iterations in lines 9–12 potentially calculate the utility (normalized entropy) when moving each VM in the system, and the complexity for calculating the normalized entropy when moving one VM is linear in n (see Equation 4). The two loops in lines 3–5 and 6–8 both have linear complexity ($O(n)$). Since the util-

Algorithm 4 Pull strategy - Source node

```
1:  if Event then
2:    if Event = Underutilized then
3:      for all PMs do
4:        Send stealMessage containing resource
          usage
5:      end for
      /* Waiting for bids */
6:      while bids < PMs − 1
          and elapsedTime < maxTime do
7:        Receive bids
8:      end while
9:      for all PMs do
10:       Calculate suitability for each VMs
11:       Calculate utility for the suitable VMs and
          append to utilityArray
12:     end for
13:     Remove candidates from utilityArray that
        would worsen the load distribution in the
        system
14:     for all utilites in utilityArray do
15:       Calculate migration cost
16:     end for
17:     Sort utilityArray based on utility
18:     Remove candidates that have a high cost
19:     while utilityArray ≠ empty do
20:       Select VM with highest utility from
          utilityArray
21:       Send stealAttemptMessage to PM hosting
          the VM
      /* Waiting for ack */
22:       Receive ack
23:       if elapsedTime > timeout
          or ack ≠ "OK" then
24:         Remove VM from utilityArray
25:         if utilityArray ≠ empty then
26:           Reset timeout
27:         end if
28:       else
29:         Wait for the VM to be migrated
30:       end if
31:     end while
32:     Run backoff algorithm
```

Algorithm 5 Pull strategy (Continued) - Candidate node(s)

```
33:   else if Event = stealMessage then
34:     Reply with information about PM and hosted VMs
35:   else if Event = stealAttemptMessage then
36:     if requested VM is locked then
37:       Set ack to "NOT OK"
38:       Send ack to steal initiator
39:       Exit
40:     end if
41:     Lock requested VM
42:     Set ack to "OK"
43:     Send ack to steal initiator
44:     Initiate migration of requested VM
45:     Remove VM lock when the migration is done
46:   end if
47: end if
```

ity array cannot exceed the number of VMs, the loop in lines 14–16 has linear complexity ($O(n)$). The sort operation in line 17 has complexity $O\left(n^2\right)$ or $O(n * \log n)$ depending on which sort algorithm we use. Finally, the loop in lines 19–31 has linear complexity ($O(n)$).

6. Test Setup

We evaluated the proposed algorithms using a simulation testbed[4] based on OMNeT++ v4.3, a powerful, open-source, discrete-event simulator [30].

Table 2 shows the main configuration settings for different scenarios that were simulated. A complete list of simulation settings can be found in Appendix A.

In all scenarios, we have used the normalized load values 0.3 and 0.7 as thresholds for low load (*i. e.*, underutilized PMs) and high load (*i. e.*, overloaded PMs), respectively. When the threshold is adaptive these values are used as initial values. We compute the load level of the system as the average load for all PMs in the simulated system. We have defined the load ranges 0.0–0.3, 0.3–0.5, 0.5–0.7 and 0.7–1.0 as *minimal*, *low*, *medium* and *high* system load level, respectively. The system load changes due to the simulated load in existing VMs and also due to VMs being added or removed from the system.

Each simulation run requires an initial load level. The simulations begin by creating VMs until the load of the system reaches the desired level. The VMs are then randomly distributed to the PMs.

We have experimented with three scenario types. In a *burst* scenario, at a certain time during the simulation, the load level is suddenly raised to the high threshold level of the system. The *drain* scenario is the opposite of the burst scenario. In this case, at a certain time during the simulation, the load is suddenly dropped to the low threshold level of the system. The purpose of the burst and drain scenarios is to allow us to observe how long it takes for the two different strategies, *push* and *pull*, to bring the system back to a state similar to the one before the sudden load change occurred. Bursts and drains can occur when IT strategies such as cloud bursting and "follow-the-sun" are in use. In cloud bursting, enterprises handle temporary periods of peak load by moving some of their virtual servers to rented computing resources in the cloud. In the "follow-the-sun" strategy, virtual servers and workstations are moved periodically between different geographical sites and time zones so that various teams can collaborate in operating projects and services around the clock (*i. e.*, one team pick from where the other has left off) [10].

The *normal* scenario represents a system where the system load will not be exposed to any sudden drain or burst, but only changes slightly over time.

The *physical machines* parameter defines the number of PMs used in a scenario. We have conducted simulations with 19, 49 and 99 PMs.

The *VMs added* parameter defines how many VMs will be added during the simulation. The VMs are added at random points in time as decided by an exponential distribution. The *VMs removed* parameter defines how many virtual machines will be removed during the simulation run. The VMs are removed at uniformly distributed points in time.

For each scenario, we have performed 30 simulation runs with different seeds for the random number generators. This provided us with data to compute averages and corresponding 95% confidence intervals.

The *push* and *pull* strategies were tested independently in every scenario.

The specification of each PM is chosen at the beginning of a simulation run from one of the configurations shown in the list below:

- 4 cores, 4 GB of RAM, 1 Gbps network bitrate
- 4 cores, 8 GB of RAM, 1 Gbps network bitrate
- 8 cores, 8 GB of RAM, 1 Gbps network bitrate
- 8 cores, 16 GB of RAM, 1 Gbps network bitrate

[4]The simulation source code developed during our study is available from `http://www.bth.se/com/dil.nsf/pages/alm`.

Table 2: Simulated scenarios

Scenario #	Type	Physical machines	VMs added	VMs removed	Adaptive threshold
1	Normal	19	0	0	No
2	Normal	19	0	0	Yes
3	Normal	49	0	0	No
4	Normal	49	0	0	Yes
5	Normal	99	0	0	No
6	Normal	99	0	0	Yes
7	Burst	19	0	0	Yes
8	Burst	49	0	0	Yes
9	Burst	99	0	0	Yes
10	Drain	19	0	0	Yes
11	Drain	49	0	0	Yes
12	Drain	99	0	0	Yes
13	Normal	19	60	60	Yes
14	Normal	49	150	150	Yes
15	Normal	99	300	300	Yes

- 16 cores, 16 GB of RAM, 1 Gbps network bitrate

The configurations are selected with a probability of $1/6$ except for the combination 4 cores and 4 GB of RAM, which is selected with the probability of $1/3$. These configurations represent typical hardware choices for low- to mid-priced blade servers.

A VM's specification is decided at the instant it is created in the simulation. The number of virtual cores are selected from a discrete uniform distribution consisting of values 2, 4 and 8. The VM memory size is also chosen from a discrete uniform distribution with values 256 MB, 512 MB, 768 MB or 1024 MB. Finally, the page dirty rate is also chosen from a discrete uniform distribution with values $5000, 10000, 15000, \ldots, 60000$ pages[5] per second. This allows for a total of 144 different VM configurations.

We have performed a basic evaluation of the simulator by verifying that the simulator output follows the expected outcome in simple scenarios. Such a scenario contained a small number of PMs and VMs so that the implemented algorithms could easily be traced. Furthermore, the initial load as well as later changes in the load of PMs and VMs were chosen such that triggered auctions and their outcome could be predicted by inspecting the algorithm descriptions from Section 5. More details are available in [21].

7. Results

We will use the term *profile* to refer to the combination of a migration strategy with an initial system load level. We have the following profiles: *pull*-low, *pull*-medium, *pull*-high, *push*-low, *push*-medium, and *push*-high. Also, we use the term *system performance* to denote the load entropy of the system.

7.1. Fixed and Adaptive Thresholds

Figure 6 shows results from Scenario 1 (see Table 2). In Figure 6a, we can see that most migration attempts take place for *push*-high and *pull*-low. We can also see, in Figure 6b, that these profiles show many more successful migrations than the other profiles. This is not surprising because in a highly loaded *push*-based system there will be many overloaded PMs trying to auction load away. Similarly, in a lightly loaded *pull*-based system there will be many underutilized PMs offering resources for auction. When the load in these systems moves towards the other end of the spectrum, the PMs' willingness to migrate work decreases because the corresponding load thresholds are not reached. A complementary view of this behaviour is provided by the system performance metric (*i. e.*, load entropy) shown in Figure 6c. There, we can observe that the *push*-low and *pull*-high profiles have less success in balancing the load in the system.

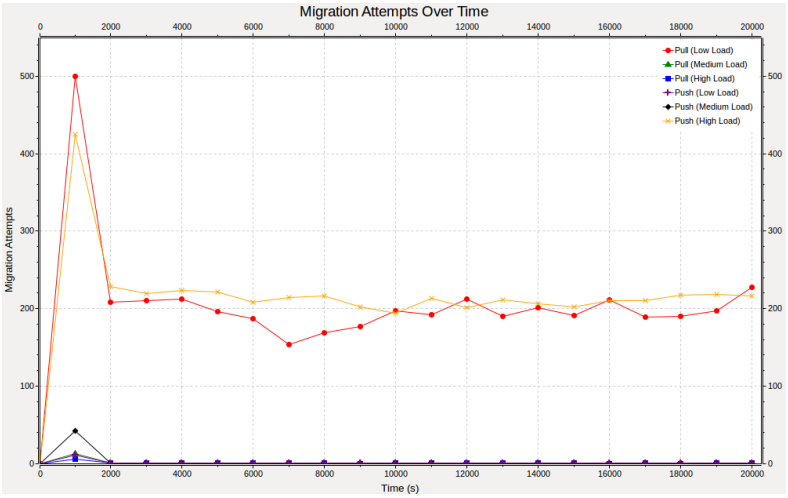
In Scenario 2 the adaptive threshold is activated. We can see in Figure 7a and Figure 7b that for the *pull*-low and *push*-high profiles the curves for the number of migration attempts and the number of successful migrations are almost identical to those in scenario 1 (*c.f.*, Figure 6a and Figure 6b). The adaptive threshold does not improve the performance of the system for these profiles. The reason is that in the *pull*-low profile no PM is willing to release workload because a new cold spot would be created. On the other hand, in the *push*-high profile no PM is interested in accepting workload because it will cause them to become overloaded.

However, for the remaining profiles we can observe a significant increase in the number of migration attempts and successful migrations, in particular from simulation time 2000 s and onwards. This is even more visible in Figure 7c where we can observe that the load entropy corresponding to these profiles is closer to 1.0 than in Scenario 1.

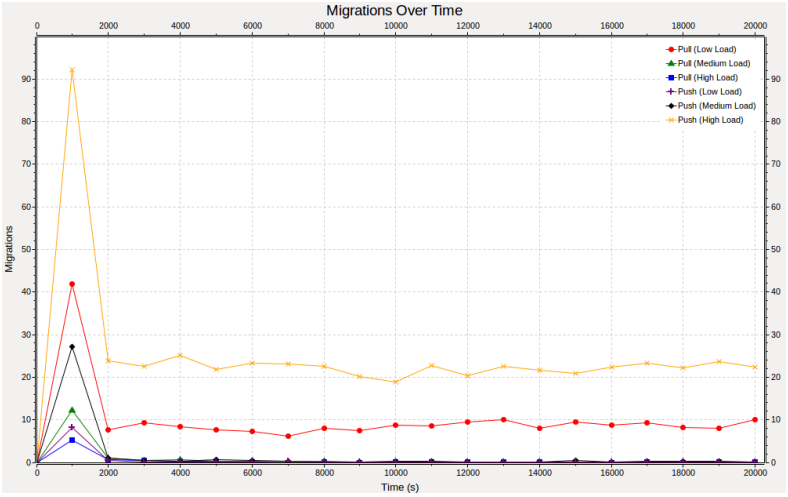
The results from scenarios 3–6 show similar behaviour in terms of effects due to fixed thresholds and adaptive thresholds.

Based on the results for the scenarios 1–6, we decided to enable the adaptive threshold for the rest of the experiments, as can be observed in Table 2.

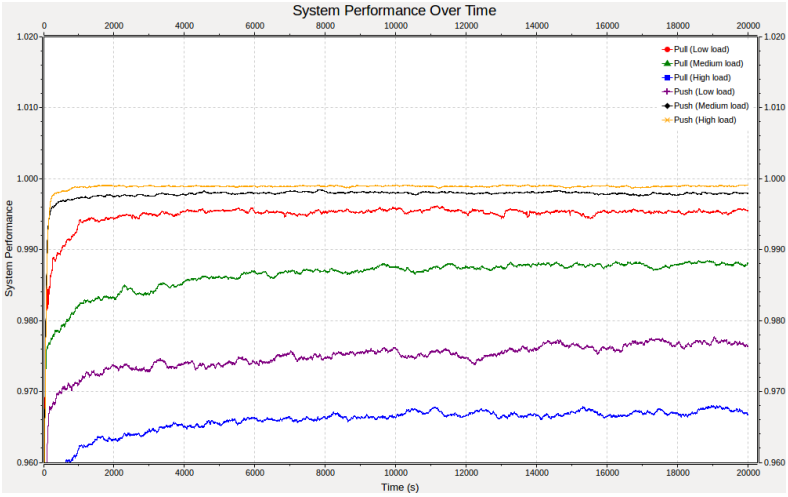
[5] We used 4 KB for the size of a memory page.



(a)

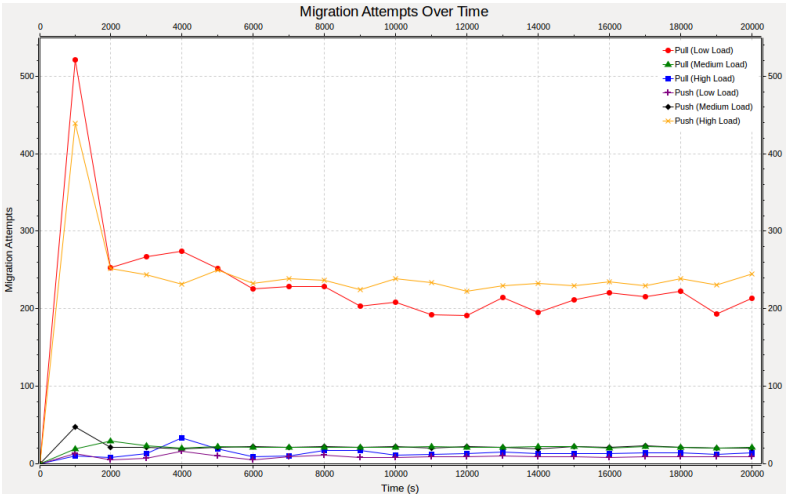


(b)

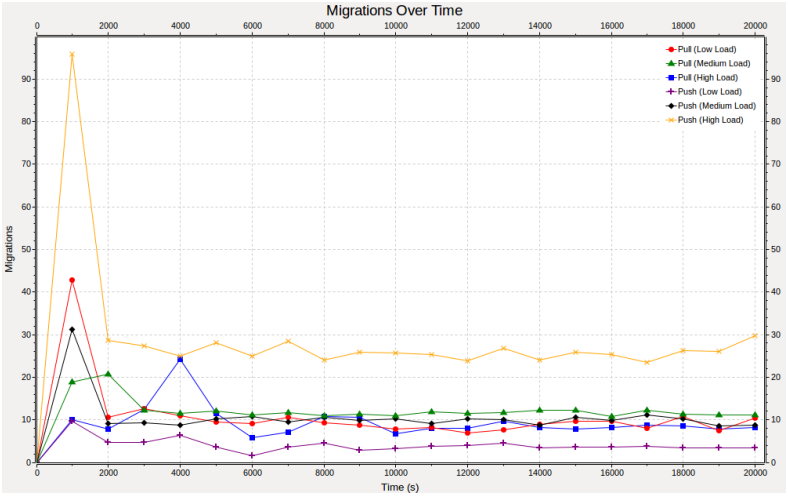


(c)

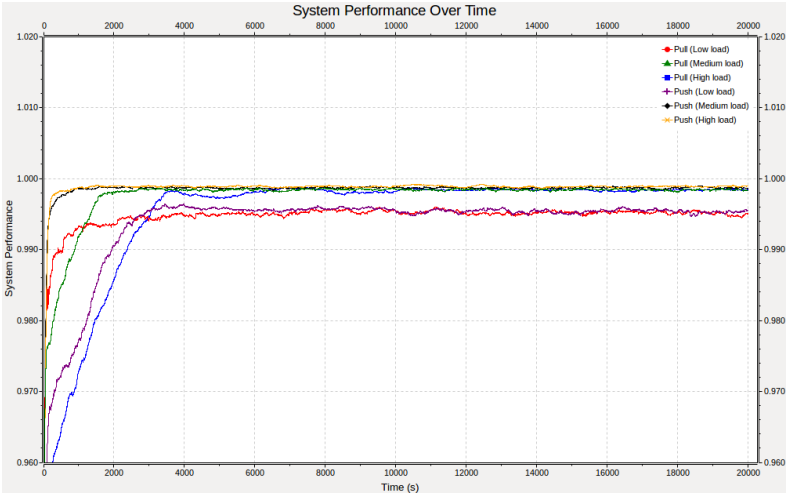
Figure 6: Scenario 1: *Normal*, 19 PMs, fixed threshold



(a)



(b)



(c)

Figure 7: Scenario 2: *Normal*, 19 PMs, adaptive threshold.

7.2. Burst and Drain Scenarios

This section presents results for the burst and drain scenarios 9 and 12. The results for the scenarios 7–8 and 10–11 show a similar pattern and are omitted in order to save space.

We simulate a burst by suddenly increasing the system load level from low to high, or from medium to high, depending on the profile. This is done by adding VMs to the system. We exclude the *pull*-high and *push*-high profiles from burst scenarios because their load is already set to a high level.

Similarly, we simulate a workload drain by suddenly decreasing the system load level from high to low, or from medium to low, depending on the profile. This is done by removing VMs from the system. In this case, we exclude the *pull*-low and *push*-low profiles from drain scenarios because their load is already low.

We have scheduled the burst to occur 10000 s after the simulation has started. In Figure 8a we can see that before the burst occurs most migration attempts are performed by the *pull*-low profile. This happens because in this profile many PMs are underutilized, which triggers them to initiate auctions to retrieve more work. As soon as the burst enters the system, the hosts become more loaded and are less willing to pull additional workload. Therefore, we see a drastic decrease in the number of migration attempts for the *pull*-low profile. By contrast, the *push*-based profiles are quite passive before the burst, but become more active afterwards. In Figure 8b we can see a similar behaviour for the number of successful migrations performed. If we compare the pre-burst region of Figure 8b with the same region in Figure 8a we notice that the ratio of successful migrations to attempted migrations is very low for the *pull*-low profile. Most hosts in this profile are lightly loaded and willing to acquire work. However, they are also unwilling to release work to other PMs.

Figure 8c shows how fast the system is able to distribute a burst of workload among the PMs. We can observe that the *push* profiles require about 15 minutes to balance the system load after a burst, whereas the *pull* profiles need 30–50 minutes for the same purpose. It is interesting to note that the *pull*-medium profile requires less time to bring the system back to a good state than the *pull* low profile does. This behaviour is caused by the adaptive threshold mechanism. The initial load in the system was already balanced before the burst occurs. As a result, for the *pull*-low profile there is very little workload that can be shuffled between the nodes. It means that auctions typically fail causing nodes to reset (*i. e.*, lower) the adaptive threshold to the initial value. Consequently, the threshold for most PMs stays around 0.3. In the *pull*-medium profile, there is more work to be moved between hosts, which means that the adaptive threshold can increase well beyond 0.3. Immediately after the burst occurs, most hosts in a *pull*-profile are not interested to start an auction because their load exceeds the threshold. The auctions begin first after the adaptive threshold is raised above the current load. This happens faster for PMs in the *pull*-medium profile because the adaptive threshold was already set to a higher value than in the case of the *pull*-low profile.

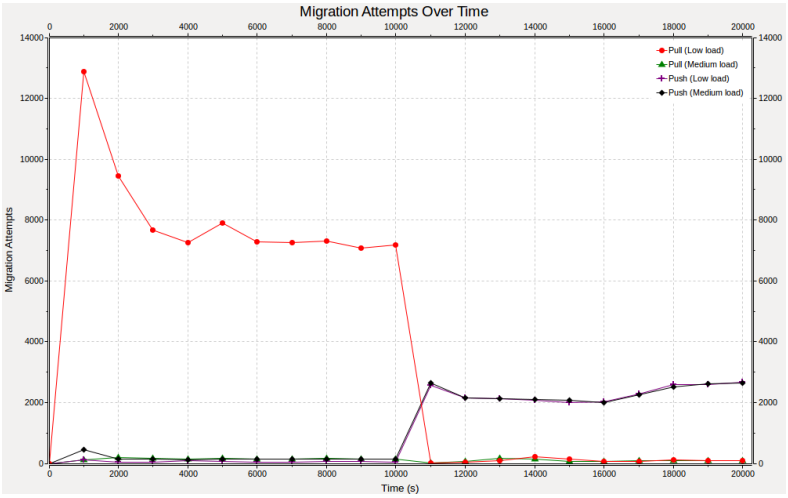
The results for the drain scenarios are almost a mirror image to those from the burst scenarios. For example, we can observe in Figure 9a and Figure 9b that the *push*-high profile in the initial phase of the simulation has both the largest number of migration attempts as well as successful migrations. After the drain occurs the level of activity in the *push*-profile drops steeply while increasing for the *pull*-profiles. Also, we see again a low ratio for migration attempts to successful migrations for the *pull* profiles. In this case, it occurs in the post-burst region of the graph. However, the cause is slightly different than in the burst scenario. Here, the adaptive thresholds of the PMs had adapted to a medium or high load level. After the drain occurs, most PMs return to a low level of load and begin auctioning their resources. But most auctions fail, because there are very few hosts that are willing to give up workload.

In Figure 9c we can observe that the *pull* profiles are able to redistribute the load after a drain in just 4 minutes compared to the 30–40 minutes required by the *push* profiles.

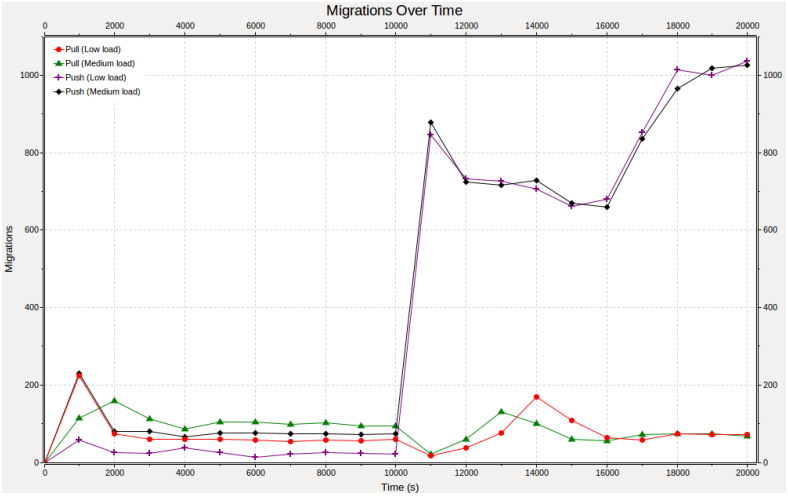
7.3. Normal Operation

Scenarios 13–15 reflect a more typical form of operation for our automated migration system, at least more normal than in the burst and drain scenarios, which were meant to examine extreme behaviour. In scenarios 13–15, workload, in the form of new VMs, arrives at the system at exponentially distributed time instants. The result is that the arriving VMs are more likely to appear in the beginning of the simulation and have a more rare occurrence towards the end. We use a uniform distribution for the time instants when a VMs are removed from the system. Figure 10 show the system load for the low, medium, and high profiles when VMs are dynamically added and removed from the system, as describe here. The slightly curved appearance of the plots resembles the load variations throughout the day, for example when going towards peak hour and then away from it.

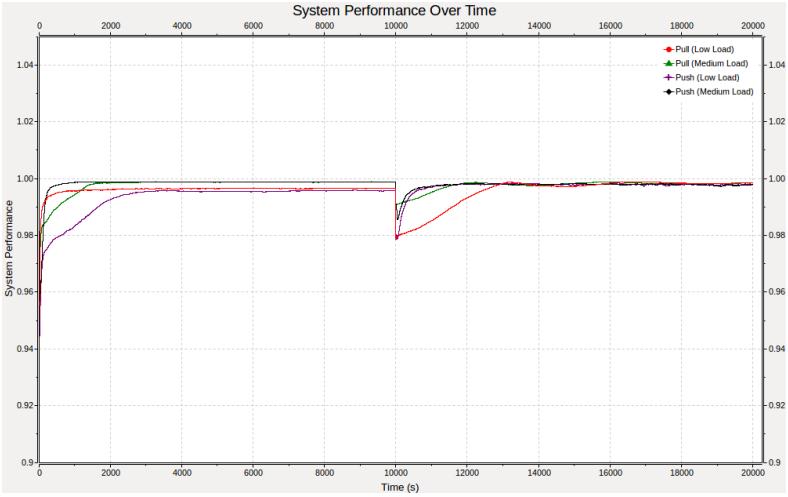
As in the previous cases, the results are quite similar between the three scenarios and therefore we choose to



(a)

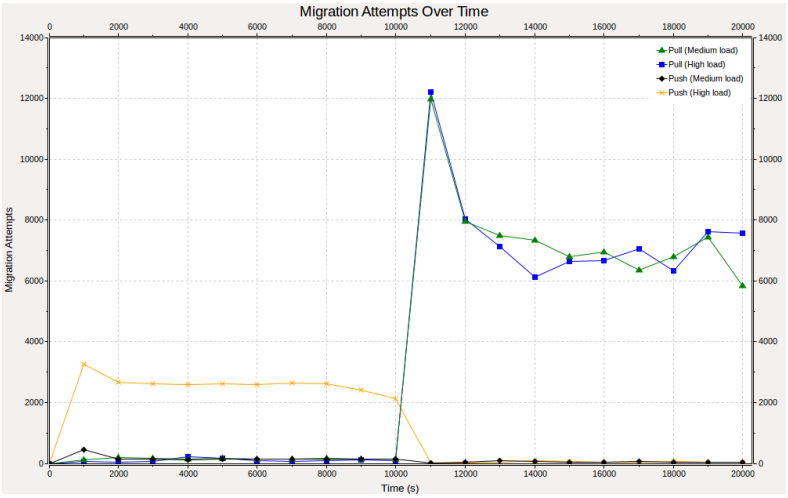


(b)

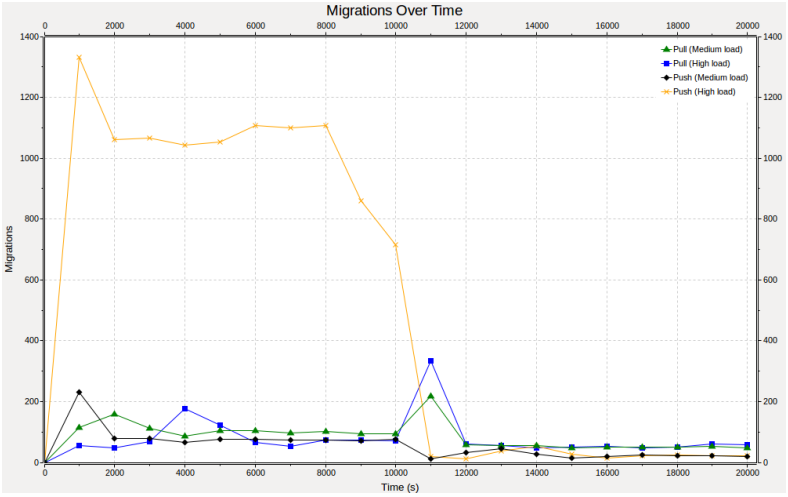


(c)

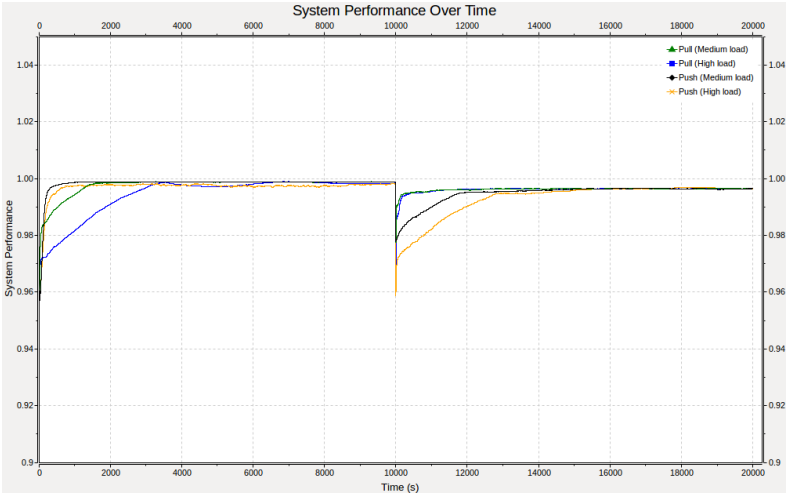
Figure 8: Scenario 9: *Burst*, 99 PMs



(a)



(b)



(c)

Figure 9: Scenario 12: *Drain*, 99 PMs

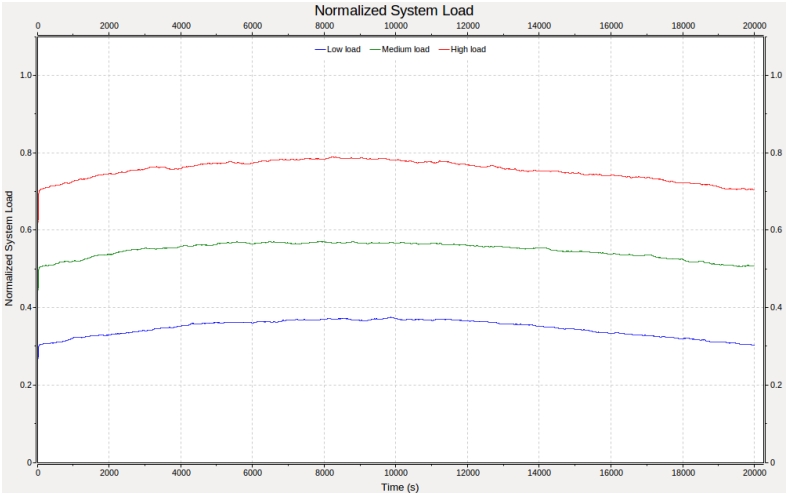


Figure 10: System load for profiles in scenario 13–15

focus on Scenario 15 only.

In Figure 11a we can see the number of migration attempts for each profile. The profiles *push*-high and *pull*-low perform a large number of migration attempts immediately after the simulation starts. The other profiles are less active because the adaptive thresholds need time to adjust themselves to the existing load.

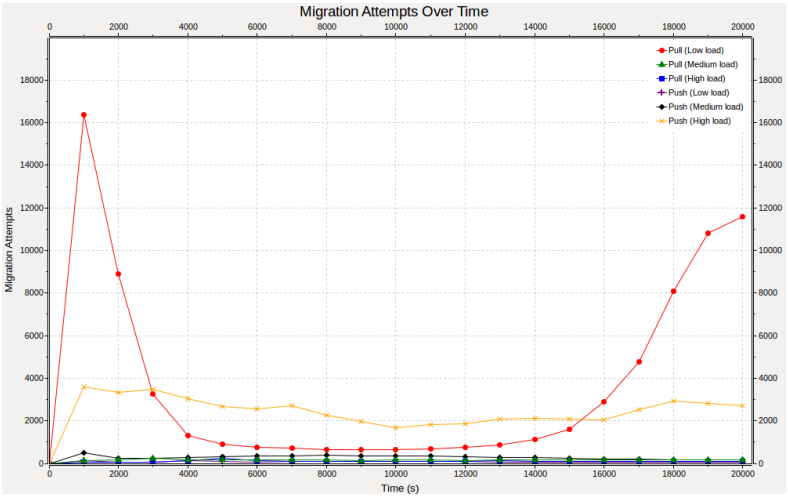
It is interesting to note that the *pull*-low profile show a peak at the beginning and at the end of the simulation with a trough in between. The initial peak was observed also in the previous scenarios and is therefore less surprising than the other two items. These are caused by the interaction between adaptive thresholds and the changing system load shown in Figure 10. After the initial load is distributed, the load of most PMs is slightly above the threshold. Thus, most PMs do not initiate any auctions. The little amount of migration attempts that takes place is due to VMs that are added and removed from the system and to adaptive thresholds being slowly raised. Because the system is lightly loaded and because the thresholds are set higher than the initial values, the attempts have a high rate of success, as can be seen in Figure 11b. Eventually, the system load decrease as shown in the bottom graph in Figure 10. At that point, many PMs see their load drop below the threshold and begin auctioning their resources. However, because the threshold of potential buyers is high, the auctions fail. This triggers seller to reset their adaptive thresholds to the initial value. In turn, this makes them less willing to release own VMs to other buyers.

The *push*-high profile shows a different behaviour. In this case, there is also an initial peak in Figure 11a where workload is distributed in the system. The system load level is gradually increased causing the load of individual hosts to raise near or above the threshold. Overloaded PMs begin auctions to offload VMs but the auctions fail because the potential buyers are not interested in acquiring additional work. The drop in successful migrations can be observed in Figure 11b. Eventually, the system load begins decreasing as shown by the top curve in Figure 10. At this point the number of successful migrations increases again.

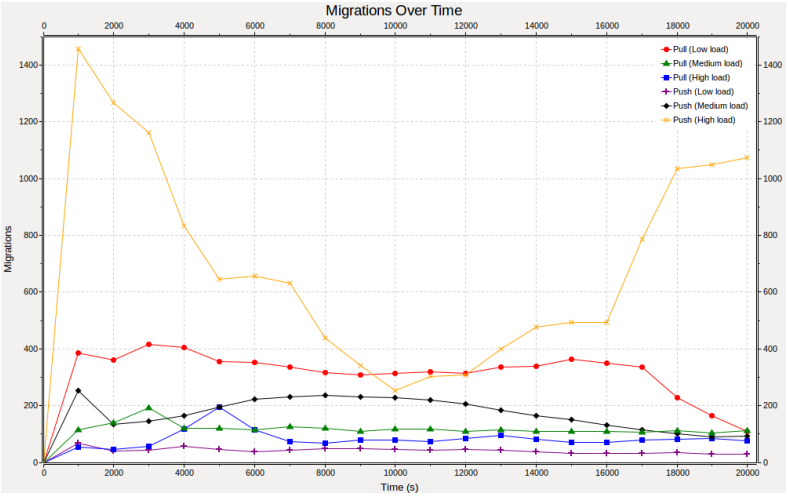
Figure 11c shows that once the initial load is distributed in the system, the slow change in the system load level has little effect on the system performance (*i. e.*, on the load entropy). The *push*-medium profile shows the best performance overall. This is not surprising because medium load level is an optimal condition to avoid both hot spots and cold spots. Also, we see a dip in the curve for the profiles *push*-low and *pull*-low, in the end of the simulation runs, roughly around 17000 s on the time axis. For the *push*-low profile this is caused by the combination of underutilized PMs and the time it takes for the adaptive threshold to reach a level where migrations can be performed. For the *pull*-low profile the cause is a decrease in the number of VMs while most of the PMs are underutilized. When this happens there will be fewer and fewer migrations because additional migrations do not improve the performance of the system.

8. Conclusions

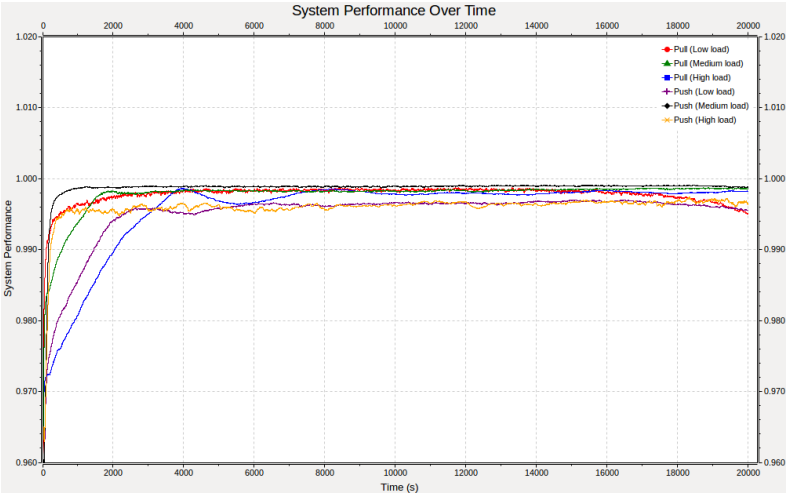
An important quality metric for the proposed strategies is the time required to re-balance the load when VMs are added to or removed from the system. From



(a)



(b)



(c)

Figure 11: Experiment 15: *Normal*, 99 PMs, normal operation

the simulation results we can conclude that the *pull* and *push* strategies require more time when the system load level is high and low, respectively. In these cases, the adaptive thresholds require time to adjust themselves to load level, which is the main reason for the slow response.

On the other hand, the *push* strategy will not perform well if the load level is very high. In this case, many auctions will be triggered but most of them will fail because the hosts are overloaded. This situation is very detrimental to the well-being of the system, because the large number of migration attempts will increase the load, and network capacity will be wasted.

If the system load level is close to the initial threshold value, the migration strategies will be able to re-balance load quickly. However, this will still generate bursts of high volume of network traffic because migrations are "packed" together. This may also increase the load of the nodes participating in auctions.

Our results show that the *push* and *pull* strategies complement each other. One could, therefore, consider running both in parallel, *i. e. ,* initiate auctions if the load is either under a low threshold or over a high threshold. Another alternative would be to let the system switch between using *push* or *pull* strategies based on the current system load level. Similar approaches have shown promising results in previous studies [15].

Appendix A. Simulation Settings

These are the configurable settings together with the values used for the simulations described in the article. Table A.3 shows the random generators connected to different random variables used by the simulations. The PM parameter are listed in Table A.4. The remaining parameters are displayed in Table A.5.

Table A.3: Random number generators.

Parameter	Description
rng-0	Used for the backoff algorithm.
rng-1	Sets the size of PM memory.
rng-2	Sets VM ids.
rng-3	Sets VM load pointer initialization.
rng-4	Sets a VMs memory limit.
rng-5	Sets the VM memory page dirty rate.
rng-6	Selects which host to send the initial VMs to.
rng-7	Sets the number of virtual cores on a VM.
rng-8	Sets the creation times for the dynamically created VMs.
rng-9	Used to select a PM and dynamically create a VM on it.
rng-10	Used to decide at what time to update the adaptive threshold.
erng-11	Used to decide how much the adaptive threshold will increase/decrease.
rng-12	Used to decide when to remove a VM from a PM.

References

[1] VMware, "Understanding full virtualization, paravirtualization, and hardware assist," VMware, Inc., White Paper, Nov. 2007.

[2] C. Clark, K. Fraser, S. Hand, and J. G. Hansen, "Live migration of virtual machines," in *Proceedings of NSDI*, Boston, MA, USA, May 2005.

[3] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, "Xen 3.0 and the art of virtualization," in *Linux Symposium*, Ottawa, Ontario, Canada, Jul. 2005.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, Ottawa, Canada, Jul. 2007.

[5] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of USENIX*, Anaheim, CA, USA, apr 2005.

[6] M. Mishra, A. Das, P. Kulkarni, and A. Sahoo, "Dynamic resource management using virtual machine mgrations," *IEEE Communications Magazine*, vol. 50, no. 9, pp. 34–40, Sep. 2012.

[7] J. Moore, J. Chase, P. Ranganathan, and R. Sharma, "Making scheduling "cool": Temperature-aware workload placement in data centers," in *Proceedings of USENIX*, Anaheim, CA, USA, Apr. 2005.

[8] O. Sarood, P. Miller, E. Totoni, and V. K. Laxmikant, ""Cool" load balancing for high performance computing data centers," *IEEE Transactions on Computers*, vol. 61, no. 12, pp. 1752–1764, Dec. 2012.

[9] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *Proceedings of IEEE CLUSTER*, New Orleans, LA, USA, Sep. 2009.

[10] T. Wood, P. Shenoy, K. K. Ramakrishnan, and J. Van der Merwe, "CloudNet: Dynamic pooling of cloud resources by live WAN migration of virtual machines," in *Proceedings of ACM VEE*, Newport Beach, California, USA, Mar. 2011.

[11] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in *Proceedings of ACM HPDC*, Munich, Germany, Jun. 2009.

[12] X. Liao, H. Jin, and H. Liu, "Towards a green cluster through dynamic remapping of virtual machines," *Future Generation Computer Systems*, vol. 28, no. 2, pp. 469–477, 2012.

Table A.4: Physical machine parameters.

Parameter	Value(s)	Description
maxMemory	$2^{0\|1\|2} * 4096$	Sets the maximum primary memory of a PM (4, 8, 16 GB).
linkSpeed	1024	The bandwidth capacity for each PM.
PageSize	4	The memory page size (KB)
PageDirtyRateMultiplier	5000	Steps for the page dirty rate.
PageDirtyRateMax	12	Maximum multiplier value, i. e. $5000 * 12$
Threshold	256	Used for the cost model to determine when the pre-copy algorithms terminates (KB).
TransmissionRateMultiplier	0.5	The amount of bandwidth that should be available for a migration, 0.5 = 50%.

Table A.5: General parameters.

Parameter	Default value(s)	Description
repeat	30	The number of times an experiment will be repeated with different seeds.
num-rng	13	The number of RNG's.
seed-set	runnumber	The seed for the RNG's is based on the runnumber (0-29).
sim-time-limit	20000	The number of seconds a simulation run will last.
backoffLimit	120	The maximum time allowed for the backoff algorithm (s).
EWMALambda	0.2	The factor of how much historical data points will affect the predicted value.
loadWindow	100	The number of samples stored that the EWMA will make its predictions on.
hotspotReplyTimeout	10	How many seconds a node will wait for a reply of the initial broadcast.
recordHotspotInterval	1000	How often the number of attempted migrations and successful migration will be sampled (s).
initialSystemLoad	low/medium/high	0.3, 0.5 and 0.7 system load at startup.
burstHigh	true/false	Decides whether a burst should occur.
burstLow	true/false	Decides whether a drain should occur.
burstHighTime	$\frac{sim-time-limit}{2}$	The time when a burst will occur.
burstLowTime	$\frac{sim-time-limit}{2}$	The time when a drain will occur.
DynamicVM	0	The number of VMs that will be dynamically created.
VMsToRemove	0	The number of VMs that will be dynamically removed.
variableThreshold	1	Decides whether the adaptive threshold should be used.
requiredSystemStateImprovement	0	How much better (%) the predicted system performance is required to be to perform a migration.
variableUpdateTime	300	The mean time when the adaptive threshold should be updated.
CPUThreshold	0.7	The upper CPU threshold.
minCPUThresholdLimit	0.3	The lower CPU threshold.
numNodes	19/49/99	The number of PMs that will be used.
loadLevel	5	The load values will be an average over 5 minutes.
pushOrPull	0/1	Decides whether the *Push* or *Pull* strategy is used.

[13] K. Sammy, R. Shengbing, and C. Wilson, "Energy efficient security preserving VM live migration in data centers for cloud computing," *International Journal of Computer Science Issues*, vol. 9, no. 2, pp. 33–39, Mar. 2012, ISSN: 694-0814.

[14] P. Liu, Z. Yang, X. Song, Y. Zhou, H. Chen, and B. Zang, "Heterogeneous live migration of virtual machines," in *Proceedings of IWVT*, Beijing, China, Jun. 2008.

[15] A. Svensson, "Dynamic alternation between load sharing algorithms," in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, Kaual, Hawaii, Jan. 1992.

[16] X. Qin, W. Zhang, W. Wang, J. Wei, X. Zhao, and T. Huang, "Towards a cost-aware data migration approach for key-value stores," in *Proceedings of IEEE CLUSTER*, Beijing, China, Sep. 2012, pp. 551–556.

[17] D. Kunkle and J. Schindler, "A load balancing framework for clustered storage systems," in *Proceedings of IEEE HiPC*, Bangalore, India, Dec. 2008.

[18] S. Akoush, R. Sohan, A. Rice, A. Moore, and A. Hopper, "Predicting the performance of virtual machine migration," in *Proceedings of IEEE/ACM MASCOTS*, Miami Beach, FL, USA, Aug. 2010.

[19] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proceedings of USENIX NSDI*, Cambridge, MA, USA, Apr. 2007.

[20] R. Walker, "Examining load average," *Linux Journal*, vol. 152, Dec. 2006. [Online]. Available: http://www.linuxjournal.com/article/9001

[21] M. Forsman and A. Glad, "Automated live migration of virtual machines," Master's thesis, Blekinge Institute of Technology (BTH), Karlskrona, Sweden, Sep. 2013, MCS-2013:14.

[22] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. Chichester, West Sussex, United Kingdom: John Wiley & Sons, 2009, ISBN: 978-0470519462.

[23] J. Kleinberg and D. Easley, *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge, UK: Cambridge University Press, 2010, ISBN: 978-0521195331.

[24] K. Schmidt, *High Availability and Disaster Recovery*. Berlin Heidelberg, Germany: Springer-Verlag, 2010, ISBN: 978-3642063794.

[25] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 6th ed. Harlow, UK: Pearson Education, 2012, ISBN:978-0273768968.

[26] *IEEE std 802.3 - Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Computer Society Std., Dec. 2008.

[27] C. C. Holt, "Forecasting seasonals and trends by exponentially weighted moving averages," *International Journal of Forecasting*, vol. 20, no. 1, pp. 5–10, Jan. 2004.

[28] J. S. Hunter, "The exponentially weighted moving average," *Journal of Quality Technology*, vol. 18, no. 4, pp. 203–210, 1986.

[29] H. Liu, C.-Z. Xu, H. Jin, J. Gong, and X. Liao, "Performance and energy modeling for live migration of virtual machines," *Cluster Computing*, vol. 16, no. 2, pp. 249–264, Jun. 2013.

[30] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proceedings of Simutools*, Marseille, France, Mar. 2008.