# Overdriver: Handling Memory Overload in an Oversubscribed Cloud

Dan Williams * Hakim Weatherspoon

Dept. of Computer Science
Cornell University, Ithaca, NY
{djwill, hweather}@cs.cornell.edu

Hani Jamjoom Yew-Huey Liu

IBM T. J. Watson Research Center, Hawthorne, NY
{jamjoom,yhliu}@us.ibm.com

## Abstract

With the intense competition between cloud providers, oversubscription is increasingly important to maintain profitability. Oversubscribing physical resources is not without consequences: it increases the likelihood of overload. Memory overload is particularly damaging. Contrary to traditional views, we analyze current data center logs and realistic Web workloads to show that overload is largely transient: up to 88.1% of overloads last for less than 2 minutes. Regarding overload as a continuum that includes both transient and sustained overloads of various durations points us to consider mitigation approaches also as a continuum, complete with tradeoffs with respect to application performance and data center overhead. In particular, heavyweight techniques, like VM migration, are better suited to sustained overloads, whereas lightweight approaches, like network memory, are better suited to transient overloads. We present *Overdriver*, a system that adaptively takes advantage of these tradeoffs, mitigating all overloads within 8% of well-provisioned performance. Furthermore, under reasonable oversubscription ratios, where transient overload constitutes the vast majority of overloads, Overdriver requires 15% of the excess space and generates a factor of four less network traffic than a migration-only approach.

***Categories and Subject Descriptors*** D4.2 Operating Systems [*Storage Management*]: Main Memory

***General Terms*** Performance, Design, Experimentation, Management

***Keywords*** Cloud Computing, Virtualization, Resource Oversubscription, VM Migration, Network Memory

## 1. Introduction

Cloud computing is becoming increasingly competitive, with a growing list of large companies including Amazon, Microsoft, Google, and IBM, all investing in massive data centers [14]. Physical resources in these data centers are leased on an as-needed basis in the form of virtual machines. With this trend, effective usage of data center resources is becoming increasingly important. A cloud provider using the classical model of overprovisioning each VM with enough physical resources to support relatively rare peak load conditions will have trouble competing with one that can provide similar service guarantees using less resources. This suggests an opportunity for cloud providers to oversubscribe data center resources, placing multiple VMs on the same physical machine, while betting that the aggregate VM resource demands at any one point in time will not exceed the capacity of the physical machine. Unfortunately, without complete knowledge of all future VM resource usage, one or more VMs will likely experience *overload*. As oversubscription becomes increasingly popular, overload will become increasingly prevalent. The ability to manage overload is therefore a critical component of a next-generation, competitive cloud service.

While overload can happen with respect to any resource on a physical machine, we focus on memory overload. The availability of physical memory contributes to limits on VM density and consolidation and as such, is an attractive resource for oversubscription. In addition, recent pricing data for different configurations in Amazon's Elastic Compute Cloud (EC2) indicate that memory is twice as expensive as EC2 Compute Units.[1] However, memory is typically not oversubscribed in practice as much as other resources, like CPU, because memory overload is particularly devastating to application performance. Memory overload can be characterized by one or more VMs swapping its memory pages out to disk, resulting in severely degraded performance. Whereas overload on the CPU or disk result in the hardware operating at full speed with contention introducing some performance loss, memory overload includes large overheads, sometimes to the point of thrashing, in which no progress can be made. Unless a next-generation, oversubscribed cloud can manage memory overload, memory will be a bottleneck that limits the VM density that can be achieved.

Since the ability to manage overload is and will become increasingly critical, and overload of memory is particularly dangerous, we ask the question: *Can performance degradation due to memory overload under real workloads be effectively managed, reduced, or eliminated?*

* This work was performed while the first author was an intern at the IBM T. J. Watson Research Center in Hawthorne, NY

[1] We use Amazon's pricing data as input parameters for a series of linear equations of the form $p_m \times m_i + p_c \times c_i + p_s \times s_i = price_i$, where $m_i$, $c_i$, $s_i$, and $price_i$ are pricing data for configuration $i$ for memory, EC2 Compute Units, storage, and hourly cost, respectively. Also, $p_m$, $p_c$, and $p_s$ are the unknown unit cost of memory, EC2 Compute Units, and storage, respectively. Approximate solutions for the above equations consistently show that memory is twice as expensive as EC2 Compute Units. Particularly, the average hourly unit cost for memory is 0.019 cents/GB. This is in contrast with an average hourly unit cost of 0.008 cents/EC2 Compute Unit and 0.0002 cents/GB of storage.



*VEE'11*, March 9–11, 2011, Newport Beach, California, USA.

In order to answer this question, it is essential to understand and characterize the types of memory overload a next-generation cloud provider should expect to mitigate. As we have mentioned, the cloud provider must address overload caused by oversubscription. Through analysis of data center logs from well-provisioned enterprise data centers, we conclude that there is ample opportunity for memory oversubscription to be employed: only 28% of machines experience any overload whatsoever, an average of 1.76 servers experience overload at the same time, and 71% of overloads last at most only long enough for one measurement period. Experimenting with higher degrees of oversubscription on a Web server under a realistic client load, we find, while the likelihood of overload can increase to 16% for a reasonably oversubscribed VM, the duration of overload varies. While overload occasionally consists of long, sustained periods of thrashing to the disk, this is not the common case: 88.1% of overloads are less than 2 minutes long. The fact that memory overload in an oversubscribed environment is a continuum, rather than entirely sustained or transient, suggests that different types of overload may be best addressed with a different mitigation strategy/technique.

Any overload mitigation strategy will have an effect on application performance and will introduce some overhead on the data center itself. Existing migration-based strategies [3, 15, 28, 34] address memory overload by reconfiguring the VM to physical machine mapping such that every VM has adequate memory and no VMs are overloaded. VM migration is a heavyweight process, best suited to handle predictable or sustained overloads. The overload continuum points to a class of transient overloads that are not covered by migration. Instead, we propose a new application of network memory [1, 2, 7, 8, 13, 19, 24] to manage overload, called *cooperative swap*. Cooperative swap sends swap pages from overloaded VMs to memory servers across the network. Unlike migration, cooperative swap is a lightweight process, best suited to handle unpredictable or transient overloads. Each technique carries different costs, and addresses a different section of the overload continuum, but neither technique can manage all types of overload.

We present *Overdriver*, a system that adaptively chooses between VM migration and cooperative swap to manage a full continuum of sustained and transient overloads. Overdriver uses a threshold-based mechanism that actively monitors the duration of overload in order to decide when to initiate VM migration. The thresholds are adjusted based on VM-specific probability overload profiles, which Overdriver learns dynamically. Overdriver's adaptation reduces potential application performance degradation, while ensuring the chance of unnecessary migration operations remains low.

For the mitigation techniques to work, excess space is required somewhere in the data center, whether it is as a target for migration or a page repository for cooperative swap. Overdriver aggregates the VM-specific probability overload profiles over a large number of VMs in order to provide insight into the amount and distribution of excess space in the data center. We have implemented Overdriver and evaluated it when compared to either technique on its own to show that Overdriver successfully takes advantage of the overload continuum, mitigating all overloads within 8% of well-provisioned performance. Furthermore, under reasonable oversubscription ratios, where transient overload constitutes the vast majority of overloads, Overdriver requires 15% of the excess space and generates a factor of four less network traffic than a migration-only approach.

To summarize, we make three main contributions:

- We observe the overload continuum: memory overloads encountered in a data center are, and will likely continue to include both transient and sustained bursts, although an overwhelming majority will be transient.
- We show there are tradeoffs between memory overload mitigation strategies that are impacted by the overload continuum, and propose a new application of network memory, called cooperative swap, to address transient overloads.
- We design, implement and evaluate Overdriver, a system that adapts to handle the entire memory overload continuum.

The rest of the paper is organized as follows. Section 2 examines the characteristics of overload to conclude that overload is and will likely continue to be overwhelmingly transient. Section 3 describes the tradeoffs between mitigation techniques under different types of memory overload. Section 4 describes the design and implementation of Overdriver and how it adaptively mitigates the damage of all types of memory overload. Section 5 quantifies Overdriver's effects on the application and the data center, comparing it to systems that do not adapt to the tradeoffs caused by the overload continuum. Finally, related work is presented in Section 6, and Section 7 concludes the paper.

## 2. The Overload Continuum

In this section we make three observations. First, we describe various causes of overload in an oversubscribed data center, with particular focus on overload due to oversubscription. Second, we justify the claim that an opportunity for memory oversubscription exists in today's data centers through analysis of data center logs from a large enterprise. Finally, we experimentally examine the characteristics of overload caused by oversubscription to conclude that overload is a continuum, with transient overloads being dominant.

### 2.1 Types of Overload

A VM is *overloaded* if the amount of physical memory allocated to the VM is insufficient to support the working set of the application component within the VM. In a cloud infrastructure that oversubscribes memory, overload can be caused by the cloud user or the cloud provider. The former occurs when the cloud user does not rent an instance configured with enough memory to handle its working set, or if a running application component has a memory leak. In this paper, we assume that the user should purchase a larger instance to eliminate this type of overload, which is in line with most (if not all) providers' policies.

We thus focus on mitigating overload caused by the cloud provider. We call the amount of memory that a cloud provider dedicates to a cloud user's VM the *memory allocation*. If the VM's memory allocation is less than requested, then we say the physical machine hosting the VM is *oversubscribed*. Oversubscribing memory while preserving performance is possible because application components running in VMs do not require a constant amount of memory, but experience application-specific fluctuations in memory needs (e.g. change in working set). In practice, memory oversubscription can be accomplished by taking machine memory away from one VM to give to another through memory ballooning [31], transparent page sharing [31], or other techniques [10]. If the aggregate memory demand of VMs sharing a physical machine exceed the amount of physical memory on the machine, it is the cloud provider's responsibility to manage overload such that a cloud user believes it has the amount of memory it requested.

### 2.2 Opportunities for Oversubscription

To justify the opportunity for memory oversubscription in today's data centers, we examine log data from a number of production enterprise data centers, which tend to be well-provisioned. The log data covers a number of performance metrics (including CPU, memory, and disk usage) for a large data center that hosts diverse applications, including Web, financial, accounting, CRM, etc. The collected performance data is typically used by the various data

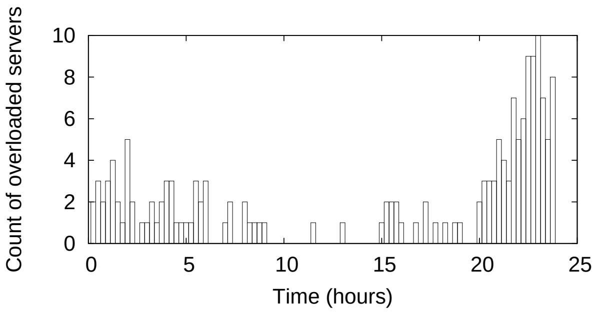



**Figure 1.** Count of simultaneously overloaded servers out of 100 randomly selected servers over a single representative day. Each point represents the number of overloaded servers during the corresponding 15 min. interval.

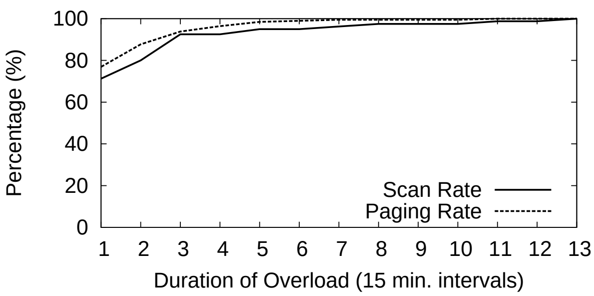



**Figure 2.** Memory overload distribution of 100 randomly selected servers over a single representative day.

centers to trend application resource usage to identify resource contention and assess the need for workload rebalancing.

There are generally two indicators used by the data center to identify if a server is having memory overload problems: page scan rate and paging rate. Paging rate is the primary indicator as it captures the operating system's success in finding free pages. In addition, a high rate of page scans provides an early indicator that memory utilization is becoming a bottleneck.

In well-provisioned data centers, overload is unpredictable, relatively rare, uncorrelated, and transient, indicating that an opportunity exists for memory oversubscription in today's data centers. To support this claim, we processed performance logs from 100 randomly selected servers. Each log is 24 hours long, while each point in the trace is the average paging rate over a fifteen-minute interval. This is the finest granularity of the log data; thus, sub-fifteen-minute information is not available to us without additional instrumentation of the servers. To capture transient overload bursts that may appear as very low paging rates when averaged over the entire fifteen minute interval, we define overload as an interval with a non-zero paging rate.

We analyzed the data in three different ways. First, we looked at the prevalence of overload (irrespective of its duration) across the 100 servers. We observed that overload is rare: only 28 of the servers experience some kind of memory overload. Second, we studied the frequency of simultaneous overload. Figure 1 shows a time series plot of the count of overloaded servers over the 24-hour measurement period. The figure shows that at most 10 servers were simultaneously overloaded. However, the average over the 24-hour period is 1.76 servers, suggesting that servers sharing physical machines are unlikely to experience correlated overload. Finally, we studied the duration of the overload. Figure 2 shows the distribution of the duration of memory overload (using both metrics—page rate and scan rate). By definition, the figure only looks at the servers that experienced overload in one or more 15-minute intervals. The figure shows that 71% were overloaded for one interval, 80% (71% + 9%) up to two intervals, 92.5% (71% + 9% + 12.5%) up to 3 intervals (15 min, 30 min, 45 min respectively).

### 2.3 Overload Due to Oversubscription

If a cloud provider indeed takes advantage of memory oversubscription, we must understand the characteristics of overload as oversubscription is increased. We would like to analyze real data center logs again, however, we do not have access to traces from a data center that currently employs memory oversubscription.

Instead, we introduce a realistic application and workload in an environment within which we can experiment with different levels of oversubscription and gather fine-grained data at both application and system level. We use the SPECweb2009[2] banking benchmark to run on a LAMP[3] Web sever to provide a realistic client load. SPECweb2009 models each client with an on-off period [33], classified by bursts of activity and long stretches of inactivity. Each client accesses each Web page with a given probability, determined from analyzing trace data from a bank in Texas spanning a period of 2 weeks including 13+ million requests [29]. SPECweb2009 is intended to test server performance with a fixed client load, so, by default, client load is stable: whenever one client exits, another enters the system. This makes the benchmark act like a closed loop system. Real systems rarely experience a static number of clients, so, in order to better approximate real workloads, we use a Poisson process for client arrivals and departures. We choose Poisson processes for the clients as a conservative model; real systems would likely have more unpredictable (and transient) spikes.

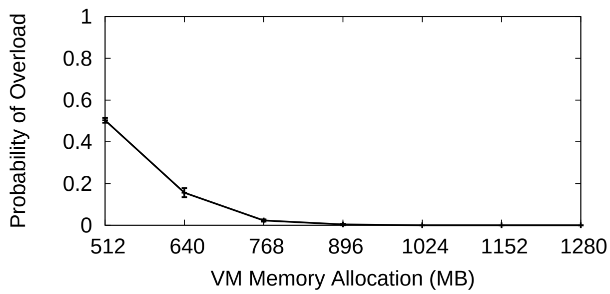
We next examine the effect of oversubscription on the duration of overload. To do so, we varied the VM's memory allocation to simulate oversubscription and ran the SPECweb2009 Web server with Poisson processes for client arrivals and departure set so that the arrival rate is 80% of the service rate. Each experiment lasted for 10 minutes. Our measurement granularity within an experiment was set at 10 seconds. To ensure high confidence, each point in the graph is the average of 75 experiments.

From this experiment, we construct a probability profile for the application VM under the different memory allocations. As expected, Figure 3(a) shows an increase in the probability of overload as memory becomes constrained. However, in addition to the frequency of overload, we are also interested in the prevalence of each overload duration. Figure 3(b) shows a Cumulative Distribution Function (CDF) of the duration of overload. We see that even at high memory oversubscription ratios, most overload is transient: 88.1% of overloads are less than 2 minutes long, and 30.6% of overloads are 10 seconds or less for an allocation of 512 MB. If the VM memory is increased to 640 MB, 96.9% of overloads are less than 2 minutes long, and 58.9% of overloads are 10 seconds.

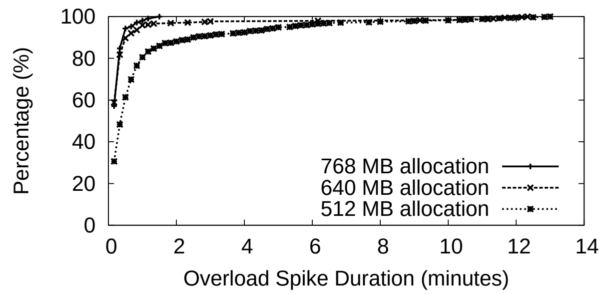
To conclude, we have confirmed that memory overload increases with memory oversubscription. In addition, overload is not solely sustained or transient, but covers a spectrum of durations. Finally, we have found that even at the high oversubscription levels that we expect to see in tomorrow's data centers, transient overloads dominate, which will be important to consider when designing Overdriver.

[2] http://www.spec.org/web2009/

[3] Linux, Apache, MySQL, PHP




(a) Likelihood of overload.




(b) CDF of overload duration

**Figure 3.** These two graphs form a memory overload probability profile for the web server component of the SPECweb2009 banking application under a variety of different oversubscription levels, including both the frequency and duration of overload.

## 3. Overload Mitigation

When considering an overload mitigation strategy, the cost of the strategy can be measured in two dimensions: the effect to the application that is experiencing overload and the effect to the data center caused by overhead intrinsic to the strategy.

Application effects refer to the performance of the application that is experiencing overload. Ideally, a mitigation strategy would sustain application response time, throughput, or other performance metrics throughout the overload, so that the cloud user is unaware that it even took place. Data center effects include the overhead or contention introduced by the overload and the mitigation strategy. Ideally, the resources used during the overload are no more than what would have been actively used if no oversubscription was in effect. In this section, we discuss the application and data center effects of two different mitigation techniques: VM migration and network memory.

### 3.1 Migration

Existing techniques to mitigate overload consist largely of VM migration techniques that address overload by reconfiguring the VM to physical machine mapping such that every VM has adequate memory and no VMs are overloaded [3, 15, 28, 34] (i.e. the VM memory allocation is increased, possibly up to the amount originally requested by the cloud user). VM migration is a relatively heavyweight solution: it incurs delays before it goes into effect, and has a high, fixed impact to the data center, regardless of the transience of the overload. For these reasons, migration strategies are usually designed to be proactive. Trending resource utilization, predicting overload, and placement strategies to minimize future overloads are key components of a migration strategy. Despite these challenges, VM migration strategies are popular because once migrations complete and hotspots are eliminated, all application components will have adequate memory to return to the performance they would have enjoyed without oversubscription.

While live migration boasts very low downtime, as low as 60ms for the migrating VM [5], in the best case, the time-until-completion of migration is dependent on the speed at which the entire memory footprint of the migrating VM can be sent over the network. In many settings, further migration delays are likely to arise from the complexity of migration decisions. In addition to computing VM placements for resource allocation, migration decisions may require analysis of new and old network patterns, hardware compatibility lists, licensing constraints, security policies and zoning issues in the data center. Even worse, a single application in a data center is typically made up of an elaborate VM deployment architecture, containing load balancers, worker replicas, and database backends, that may experience correlated load spikes. [27] A migration decision, in such case, has to consider the whole application ecosystem, rather than individual VMs. This complexity can be reason enough to require sign off by a human operator. Finally, in the worst case, the infrastructure required for efficient VM migration may not be available, including a shared storage infrastructure, such as a SAN, and a networking infrastructure that is migration aware.

The effect that a migration strategy has on the data center is mostly measured in terms of network impact. Typically, VM migration involves sending the entire memory footprint of the VM or more in the case of live migration. This cost is fixed, regardless of the characteristics of the overload that may be occurring[4]. A fixed cost is not necessarily a bad thing, especially when considering long, sustained overloads, in which the fixed cost acts as an upper bound to the data center overhead. Migration also requires a high, fixed amount of resources to be available at the target physical machine. The target must have enough resources to support a full VM, including CPU and enough memory to support the desired allocation for the migrating VM.

Coupled with the unbeatable performance of local memory available to a VM after a migration strategy completes, the fixed cost to the data center makes migration an attractive strategy to handle both predictable load increases, as well as sustained overloads.

### 3.2 Network Memory

Two important results from Section 2 highlight a gap in the solution space that existing migration-based solutions do not address. First, overload follows unpredictable patterns. This indicates that, unless oversubscription policies are extremely conservative, reactive strategies are necessary. Second, transient overload is, and will likely continue to be, the most common type of overload. As described above, migration, when used reactively, has high delays, and high network overhead due to relatively short lived transient overloads. There is an opportunity to consider reactive strategies that focus on transient overloads.

Network memory is known to perform much faster than disk [1, 2, 7, 8, 13, 19, 24], especially on fast data center networks, and has been applied to nearly every level of a system. We propose *cooperative swap*, an application of network memory as an overload mitigation solution in which VM swap pages are written to and read from page repositories across the network to supplement the

[4] Live VM migration will send more than the few hundred megabytes to tens of gigabytes of data comprising the VM's memory footprint because it must re-transmit the written working set in iterations. However, the network cost of live migration strategies is still relatively fixed because live migration strategies impose a limit on the number of iterations.

| | Tput (MB/s) | | Latency ($\mu$s) | |
|---|---|---|---|---|
| | Read | Write | Read | Write |
| Network Mem | 118 | 43.3 | $119 \pm 51$ | $25.45 \pm .04$ |
| Local Disk | 54.56 | 4.66 | $451 \pm 95$ | $24.85 \pm .05$ |

**Table 1.** Network memory vs local disk performance.

memory allocation of an overloaded VM. Cooperative swap is entirely reactive, and begins to mitigate overload when the very first swap page is written out by the overloaded VM[5]. Its impact on the network is dependent on the duration of overload. However, using cooperative swap does not match the performance of local memory over the long term.

Table 1 shows the relative throughput of disk I/O vs. network memory. These numbers were computed from running a disk dump between the system (`/dev/zero` or `/dev/null`) and a hard disk (`/dev/sda1`) versus a Linux network block device (`/dev/nbd0`) attached to a ramdisk across the network. The network connecting physical machines is 1 Gbps so can sustain a maximum rate of 125 MB/s. Both are block devices and so should be affected by Linux equally in terms of overhead[6]. Each result is the average of 20 runs of writing or reading 1 GB for the throughput test and one 4 KB page for the latency test. In our setup, network memory is significantly faster than disk. Using cooperative swap, short bursts of paging complete faster, allowing it to maintain application performance through transient bursts of overload. However, cooperative swap does not restore local memory and so cannot restore application performance to where a cloud user would expect if oversubscription was not taking place.

Cooperative swap affects the data center by spewing memory pages across the network. The amount of memory pages read or written from the network is dependent on the length of the overload. This means that cooperative swap is relatively cheap in terms of network overhead for transient overloads, but could generate unbounded amounts of network traffic for very sustained overloads. The amount of pages that must be available from the remote page repositories is also dependent on the duration of overload, however, this number is bounded by the size of the VM's originally requested memory size.

Reactivity, proportional network overhead to overload duration, and long-term performance issues make cooperative swap an attractive solution for unpredictable overloads and transient overloads, filling the gap in the solution space left by existing migration strategies.

## 4. Overdriver

As seen in Section 3, the overload continuum leads to tradeoffs between mitigation strategies. We design a system, called *Overdriver* to manage overload while being aware of these tradeoffs, adaptively deciding to use cooperative swap for transient overloads and migration for sustained overloads.

Figure 4 shows the high level components in Overdriver. Each physical machine is running a hypervisor and supporting some number of guest VMs. It also runs an *Overdriver agent* in its control domain (Domain 0), that monitors the memory overload behavior of the local guest VMs in terms of paging rate. Within the Overdriver agent, the *workload profiler* locally learns a memory overload probability profile for each VM similar to that in Figure 3, which it then uses to set adaptive thresholds on the length at which the overload is classified as sustained. Using these thresholds, the *overload controller* decides which mitigation strategy to employ for a given overload. A transient overload, defined as an overload whose duration has not yet passed the threshold, is mitigated by redirecting the overloaded VM's swap pages to cooperative swap page repositories, rather than to the local disk. If the transient overload becomes sustained, characterized when the duration of the overload exceeds the threshold, the overload controller initiates a migration operation, wherein one or more VMs on the physical machine, are migrated to perform an increase of the memory allocation of the overloaded VM.

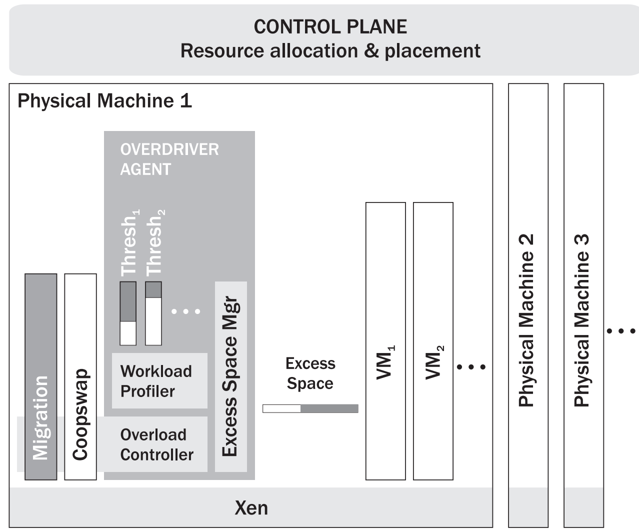



**Figure 4.** The memory resources in the data center are split into space for VMs and excess space to mitigate overload, of which there are two types: cooperative swap page repositories, and future migration targets. Resource allocation and placement of all resources is performed by the control plane.

For either mitigation strategy to be useful, there must be some *excess space* in the data center. The *excess space manager* on each physical machine is responsible for dedicating some of the local excess space to act as a target for migration, and some to act as a page repository for cooperative swap from overloaded VMs throughout the data center. The actual allocation and placement of VMs and excess space, including page repositories, throughout the data center is performed by the *control plane*, which is similar to the one in [34]. The control plane may run a proactive migration algorithm to avoid hotspots [34] or to consolidate VMs [15], however its design is out of the scope of this paper.

### 4.1 Deciding to Migrate

As described above, Overdriver uses a threshold on the duration of overload to determine when to classify an overload as sustained and employ migration. Choosing an appropriate threshold is difficult, and a good choice depends on the overload characteristics of the application VM. At one extreme, a very low threshold approaches a solely migration-based overload mitigation strategy, with good application performance, but high data center cost. On the other hand, setting the threshold to be too large approaches a network memory-based strategy, with lower application performance but also lower data center cost. Intuitively, a good choice for the threshold would be high enough that a vast majority of the overloads do not require migration while being low enough that performance does not suffer

[5] We do not use cooperative swap unless the VM has a reduced memory allocation due to oversubscription. Otherwise, overload is the responsibility of the cloud user.

[6] Care was taken to eliminate caching effects as much as possible for the latency tests, dropping the caches and writing 500 MB to the device between each latency test.

too much. A profile of the application VMs, including the probabilities of each duration of overload, can help determine a reasonable value for the threshold.

In reality, an application VM profile that describes the probability of overload having a particular duration is not available. However, rather than selecting a single threshold for all VMs, the overload manager starts with a fixed threshold, then relies on the workload profiler to learn the probabilities of various duration overloads to give a basis for reducing the threshold.

There are two challenges in learning a probability profile. First, in the case of sustained overload where migration is employed, the overloaded VM will be granted additional resources that will fundamentally change its overload behavior. In particular, the probability profile of the application VM will change, requiring the learning process to start anew. Second, the learned probability profile takes some time to converge. For example, we attempted to learn the probability profile for a VM allocated 640 MB from the SPECweb2009 experiments in Section 2. For this scenario, at least 25 sustained overloads must be witnessed before the learned profile becomes reasonable. Since the local profile is useless after a migration takes place, the 25 sustained overloads must be endured using only cooperative swap in order to learn a complete profile.

Despite these challenges, focusing only on the different durations of transient overloads, the workload profiler learns enough to reduce the threshold without resorting to excessive migration. The workload profiler maintains a list of buckets for each VM, corresponding to possible transient overload durations. As the paging rates of a VM are monitored, the profiler maintains a count for the number of times an overload of a specified duration is encountered in the appropriate bucket. Once the number of measurements exceeds a base amount, we begin to estimate a tighter bound on the migration threshold by computing the distance from the mean in which transient overload is unlikely to occur. For example, as a heuristic, we assume the distribution of transient overloads is normal, then compute $\mu + 3\sigma$ to be a new threshold. If the new threshold is lower than the original threshold, we adopt the tighter bound to reduce the time until migration is triggered for sustained overload.

### 4.2 Capacity Planning

Despite the limitations of the learned probability profiles, they can be sent to the control plane, where they can be aggregated over a large number of VMs over time. This can give insight into the quantity of excess space in the data center needed to handle overload and how to partition it between space for migration and space for cooperative swap. The control plane, in return, informs each excess space manager how to subdivide its resources.

To demonstrate how the probability profiles inform the subdivision of excess space, consider a VM running the webserver component of the SPECweb2009 application described in Section 2, when allocated only 640 MB of memory. According to its probability profile (Figure 3) this application VM has a 16% chance of experiencing overload where 96% of the overloads are less than 1 minute in duration. In order for migration to handle sustained overload, we assume that there must be enough excess capacity to have room to migrate a full VM. Similarly, in order for cooperative swap to handle transient overload, we conservatively assume that the sum of the overloaded VM's current memory allocation and the excess space that is required for cooperative swap is equal to the non-oversubscribed allocation, regardless of how short the burst is.

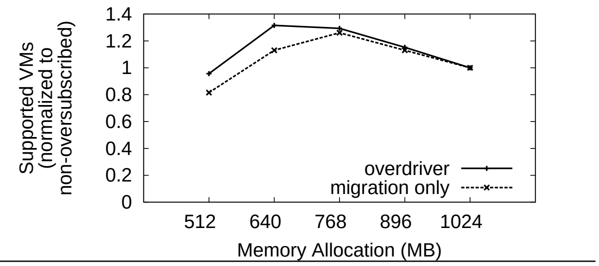



**Figure 5.** Overdriver allows more VMs to be supported by balancing the amount of excess space needed for sustained and transient overloads.

Assuming the overload characteristics of all VMs are independent[7], if $p$ is the probability of the most likely VM to have overload, we can compute a bound on the probability that at most $k$ VMs will experience simultaneous overload:

$$P\{\#\text{ overloaded VMs} \leq k\} = \sum_{i=0}^{k} \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i},$$

where $n$ is the number of VMs in the data center. For example, consider an oversubscribed data center supporting 150 VMs, all running the SPECweb2009 configuration described above. Even if each VM is allocated 640 MB, rather than the 1024 MB they would have requested, we would expect—with probability $0.97$—that no more than 3 VMs experience simultaneous sustained overload and no more than 31 VMs experience simultaneous transient overload. This, along with our assumptions about how much excess space is required to satisfy a single overload of either type, allows us to compute—with high probability—the amount of each type of excess space needed to handle all overloads in the entire data center.

Recognizing the overload continuum exists, and selecting the correct amount of each type of excess space can allow an increase in the number of VMs supported in the data center. Assuming that migration and cooperative swap can handle overload longer and shorter than one minute respectively while preserving application performance, we numerically compute the total number of VMs that can be run in the data center under different oversubscription levels. We fix the total amount of memory in the data center as a constant just under 100 GB. For each memory allocation, we input the likelihood of overload and the likelihood that the overload is transient or sustained from Figure 3. The analysis iteratively increases the number of VMs and calculates the amount of excess needed to handle the expected number of simultaneous overloads, in order to find the maximum number of VMs that can be safely supported.

Figure 5 shows the number of VMs that can be supported with high probability using Overdriver or a migration-only strategy. As memory becomes more oversubscribed, more VMs can be run, but more excess space must be maintained to endure overload. Since cooperative swap consumes space proportional to the overload duration, it allows more room for VMs than an approach that uses migration to address all types of overload, including transient overload. To make this point clear, Figure 6 shows the breakdown of memory in the data center, with Overdriver saving 12% more of the data center resources for VMs in the 640 MB case. If the physical machines are oversubscribed too much, corresponding to VM allocations of only 512 MB in this case, the excess space required

[7] VMs that make up a single application can experience correlated overload; however the enterprise data in Section 2 indicates that overload is not highly correlated across all VMs in a well provisioned data center.

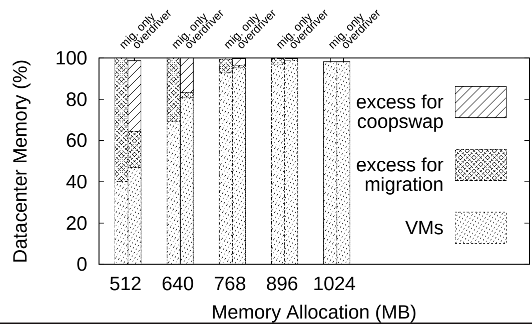



**Figure 6.** The breakdown of data center resources in terms of space for VMs, excess space for migrations, and excess space for cooperative swap.

to handle sustained overload through migration begins to dominate, reducing the available memory to support additional VMs.

### 4.3 Discussion

The maintenance of excess space is a key issue when designing a system to handle overload in the data center. Where the various types of excess space are placed throughout the data center, when and how excess space is reclaimed after an overload, and how to combat fragmentation within excess space, are all important questions to consider.

Each mitigation strategy imposes different constraints on how its portion of excess space can be placed throughout the data center. Excess space saved for cooperative swap in the form of page repositories has very few constraints on placement: there is no requirement that swap pages are stored on the same physical machine, nor do the page repositories need many other resources, like CPU. On the other hand, excess space saved for migration has more constraints: there must be enough resources to support a full VM co-located on a single physical machine, including memory and CPU. Another mitigation strategy that we have not discussed is memory ballooning, or modifying the memory allocation on the local machine. Excess space saved for ballooning has very limiting constraints, namely that it must reside on the same physical machine as the VM that is experiencing overload. Resource allocation and placement algorithms must adhere to each of these various constraints.

Overload occurs in a continuum of durations, but when it finally subsides, the excess space that was being used for the overload must be reclaimed. The reclaiming process can be proactive, in which excess space is pre-allocated before overload occurs, or reactive, in which resources are dynamically carved out of the data center on an as-needed basis. Regardless, policies for reclaiming resources are tightly integrated into VM placement and resource allocation, located in the control plane.

Reclaiming memory resources for future migration operations may require squeezing the memory allocations of VMs which may or may not have experienced an overload, and care must be taken to ensure that the reclaiming process does not trigger more overloads in a cascading effect. Reclaiming space in a cooperative swap page repository, on the other hand, can be straightforward—if the VM reads a swap page after overload subsides, that swap page can most likely be deleted from network memory. Otherwise, if the swap page has not been read, the swap page will remain outside of the VM. However, the swap page can be copied to local disk and removed from network memory at any time,[8] although the performance of a future read of the page will suffer.

Resource reclaiming may also need to consider the fragmentation that occurs within excess space. For example, after a migration completes, some resources are allocated to the overloaded VM. The amount of resources awarded need not be the original requested amount nor must they include all of the available resources on the physical machine. More likely, there will be some resources remaining that are insufficient to host a new VM, but not needed by any of the running VMs. Filling the unused resources with cooperative swap page repositories is one option to combat fragmentation, but ultimately, some consolidation process involving migration will be necessary, once again tightly integrated with VM placement and resource allocation.

### 4.4 Implementation

We have implemented Overdriver to run on Xen. We leverage Xen's built-in support for live migration and implement cooperative swap clients and page repositories from scratch. The Overdriver agent, written in Python, locally monitors the paging rate of the VMs on its physical machine to detect and react to overload. Probability profiles are maintained in a straightforward manner, which updates the migration threshold. Page repositories implementing excess space for cooperative swap exist as C programs, executed in Xen's Domain 0, that pin memory to ensure that pages written and read do not encounter additional delays. While much of the implementation is straightforward, given space constraints, we focus on some interesting implementation considerations for the overload controller and the cooperative swap subsystem.

#### 4.4.1 Overload Controller

As described above, the overload controller within the agent initiates a migration operation after overload has been sustained past an adaptive threshold. However, the implementation must be able to identify overload from normal behavior. Based on our observations of VMs that are using the paging system, a very low paging rate tends to be innocuous, done occasionally by the operating system even if there is no visible performance degradation. While any paging may be a good indication of future overload, Overdriver uses a threshold on paging rate to determine whether overload is occurring.

Furthermore, the implementation must differentiate between a series of transient overloads and a sustained overload which has some oscillatory behavior. From our experiments inducing sustained overload, we observe oscillations that can last almost a minute long. In order to correctly classify periods of oscillation as sustained overload, Overdriver includes a sliding window, where a configurable number of time intervals within the window must have experienced overload.

#### 4.4.2 Cooperative Swap

A key factor in the performance of cooperative swap is where the client is implemented. In order to remain guest agnostic, we only consider implementations within the VMM or the control domain (Domain 0) of the hypervisor. We have experimented with two different architectures. In the first, we leverage the Xen block tap drivers [32] (`blktap`) as an easy way to implement cooperative swap clients in user-space of Domain 0. When a VM begins to swap, Xen forwards the page requests into userspace, where the block tap device can either read or write the pages to the network or the disk. We noticed that swapping to disk, using a `blktap` driver in Domain 0 for the disk, was significantly outperforming

[8] Depending on the fault tolerance policy, a swap page may already be on local disk (see Section 4.4.2).

cooperative swap. The reason for this unexpected result was that pages being written by the user-space disk driver were being passed into the kernel of Domain 0, where they would enter the Linux buffer cache, and wait to be written asynchronously. Asynchrony is especially well suited to cooperative swap, because, unlike writes to file systems or databases, the pages written out have no value in the case of a client failure. Furthermore, the buffer cache may be able to service some reads. In order to take advantage of the buffer cache, in addition to reducing context switch overhead, we decided to implement cooperative swap in the kernel underneath the buffer cache, similar to a network block device (`nbd`).

Regardless of the location of the cooperative swap client, the implementation must provide some additional functionality to reading and writing pages to and from page repositories across the network. We highlight two interesting aspects of the implementation. First, cooperative swap clients must be able to locate pages, which may or may not be on the same page repository, even after a VM has migrated. In-memory state consists mainly of a per-VM hash table, indexed by the sector number of the virtual swap device. Each entry includes the address of the page repository the page is stored at, a capability to access the page, and the location of the page on disk. Both the data structure and the swap pages on disk must be migrated with a VM. This is currently done during the stop-and-copy portion of live migration, although a pre-copy or post-copy live migration technique could be implemented for both the data structure and disk pages. Second, it is important to ensure that failure of a remote machine in the data center does not cause failure of a VM that may have stored a swap page there. Fortunately, there are several accepted mechanisms for reliability in a system that uses network memory. By far the simplest is to treat the page repositories across the network like a write-through cache [7, 8]. Since every page is available on the disk as well as to the network, the dependency on other machines is eliminated, and garbage collection policies are simplified. Reads enjoy the speed of network memory, while writes can be done efficiently through asynchrony. Alternative approaches exist, such as using full replication of swap pages or a RAID-like mirroring or parity scheme [19, 22], but they add considerable complexity to failure recovery as well as garbage collection.

## 5. Evaluation

We have argued that in order to enable high data center utilization using aggressive memory oversubscription, it is necessary to *react* to overload, both transient or sustained. In this section, we quantify the tradeoffs described in Section 3 in terms of the impact of each strategy on application performance, and in terms of overhead to the data center, most notably network and excess space overhead. We also show that Overdriver successfully navigates this tradeoff, maintaining application throughput to within 8% of a non-oversubscribed system, while using at most 150 MB of excess space for transient overloads.

At a high level we want to answer the following questions:

- Does Overdriver maintain application performance despite memory overloads?
- Does Overdriver generate low overhead for the data center despite memory overloads?

The answers to these questions clearly depend on the application, its traffic, and the level of oversubscription employed. Ideally, we would like to deploy Overdriver in a production data center and experiment with varying oversubscription levels to answer these questions. Unfortunately, we do not have access to a production data center, so we once again experiment with SPECweb2009. Instead of simulating a realistic client workload as described in Section 2, we run a steady base load of clients and inject bursts of various duration in order to evaluate the performance of Overdriver in the face of different types of memory overload.

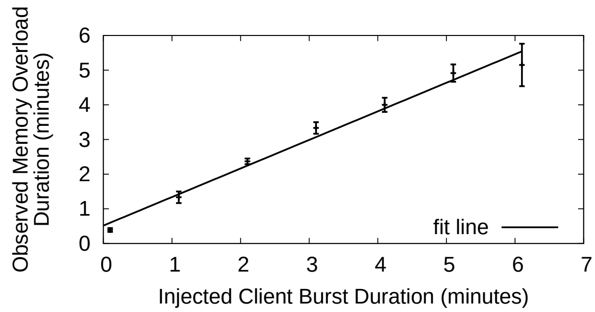
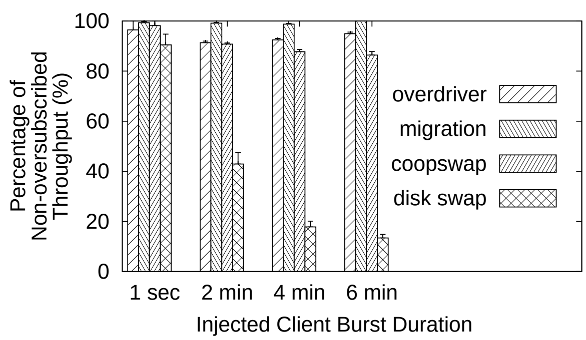



**Figure 7.** Observed memory overload duration roughly matches the duration of the injected client burst.

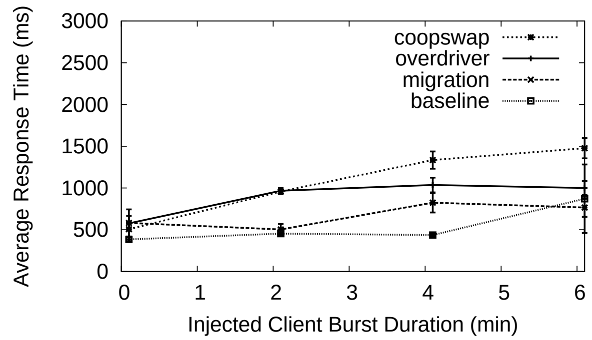
To be more precise, each SPECweb2009 experiment consists of a steady client load of 250 simultaneous sessions being imposed upon a web server VM. The VM, which initially requested 1024 MB, has only been allocated 512 MB of memory because of oversubscription, achieved by inflating the VM's memory balloon. The experiment lasts for 10 minutes. After waiting for 1 minute at the steady state, a client burst is injected, increasing the number of simultaneous sessions by 350 for a total of 600 sessions during bursts. The burst is sustained for a configurable amount of time before the simultaneous sessions return to 250 for the remainder of the experiment. A longer client burst roughly corresponds to a longer memory overload, as shown in Figure 7. The physical machines used in the experiments have 4 GB of memory each, while the memory allocation of Domain 0 is set at 700 MB.

Through the experiments we compare how Overdriver handles overload (identified as `overdriver` in each graph) to several other techniques. First, as a best-case, we measure the performance of the application, had its VM been well-provisioned. Then, the option of simply swapping pages out to disk (called `disk swap`) is provided as a worst case for application performance, and a baseline for data center overhead. In between these two extremes we run a solely cooperative swap approach (`coopswap`) and a solely migration-based approach (`migration`). The migration approach has a number of factors, discussed earlier, that make it difficult to compare against. Resource allocation and placement are central to migration strategies, as is some sort of migration trigger. To avoid these issues, we compare against an idealized migration scenario involving a different VM co-located on the same physical machine as the overloaded VM. On the first sign of overload, this other VM, which has a memory allocation of 1024 MB, is migrated to another physical machine, releasing its memory for use by the overloaded VM. In reality, VM allocations can be much larger than 1024 MB, resulting in longer delays before migration can complete and more impact to the network. So, in some sense, the migration strategy we compare against is a best-case scenario in terms of application impact and data center overhead.

Throughout the experiments, we configure Overdriver to monitor the VM every 10 seconds, and consider time intervals where the paging rate is above 200 operations per second as periods of overload. The threshold used to trigger migration is initially set at 120 seconds, with a sliding window parameter requiring 8 out of the 12 measurements to be overloads (i.e. 120 seconds comes from twelve 10 second monitoring periods and requires at least 80 seconds out of a window of 120 seconds to trigger a migration).




(a) Throughput




(b) Average response time

**Figure 8.** The effects of each mitigation technique on the SPECweb2009 application. Overdriver maintains application throughput within 8% of that of a non-oversubscribed VM and an average response time under 1 second for any length overload.

| Threshold(s) | Tput(%) | Response Time (ms) |
|---|---|---|
| 100 | 94.97 | 946 |
| 120 | 92.47 | 1249 |
| 150 | 93.76 | 1291 |
| 170 | 84.52 | 1189 |
| 200 | 84.85 | 1344 |

**Table 2.** As the threshold for migration increases, more performance loss is experienced. This table shows the lowest percentage of throughput achieved, and highest average response time, out of all durations of overload.

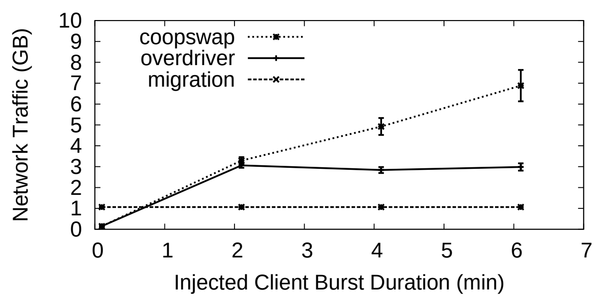



**Figure 9.** Network traffic induced by various length load spikes using cooperative swap versus disk swap.

### 5.1 Application Effects

Figure 8 shows the performance degradation experienced by the application under various overload mitigation strategies in terms of lost throughput and increase in average response time. First, we examine the aggregate throughput over the 10 minute experiment. The experiment was run on a well-provisioned VM to get a baseline for how many connections should have been handled if no overload occurred, which was between 23 and 38 thousand connections, depending on the length of the injected client burst. Figure 8(a) shows the percentage of that ideal throughput for each strategy. The first thing to notice is that the throughput while using disk swap drops off dramatically, whereas degradation is much more graceful using cooperative swap. Migration, on the other hand, completes with nearly full performance, except for very small spikes resulting in reductions in throughput performance for short periods of time, until migration completes and benefits of migration can be seen. A less aggressive solely migration-based strategy would degrade with disk swap until migration was triggered. Overdriver, on the other hand, begins to degrade gracefully along with cooperative swap, but then improves for longer overload as it switches to rely on migration. A similar pattern can be seen with application response time, shown in Figure 8(b). While longer overload periods cause cooperative swap to increase the average response time, Overdriver levels off when migration is used. Disk swap is not shown on Figure 8(b) because it performs significantly worse than other strategies; the average response time varies between 2.5 s and 16 s, depending on the overload duration. Overall, Overdriver achieves a throughput within 8% of a well-provisioned, non-oversubscribed VM, and an average response time under 1 second.

In terms of application throughput, we see that Overdriver degrades gracefully to a point, at which overload is sustained and warrants migration. The cost of Overdriver to the application, while fairly low, is higher than a very aggressive migration strategy, because Overdriver pays for the time spent deciding whether the spike will be sustained. Cooperative swap drastically improves performance while making this decision. As discussed earlier, especially given the prevalence of transient overload, the number of migration operations required is also drastically reduced, affecting both the amount of excess space required, and ultimately the number of VMs that can be supported in the data center.

The choice of threshold has an effect on the application performance, because a higher threshold translates into an increased reliance on cooperative swap. Table 2 shows application performance experienced by Overdriver as the migration threshold due to sustained overload varies. As the threshold increases above 120 seconds performance degrades. This performance degradation gives an idea of the performance that can be gained by adaptively learning a tighter threshold.

### 5.2 Data Center Effects

The most immediately quantifiable impact to the data center of the overload mitigation techniques we have described is in terms of the amount of traffic induced on the network, and in terms of the amount of excess space required for migration or cooperative swap. We show that Overdriver can handle transient overloads with a fraction of the cost of a purely migration-based solution. However, we also show that Overdriver incurs additional costs for sustained overloads.

Figure 9 shows the amount of traffic sent and received during the experiments described above for various length client bursts. The number of pages written to the cooperative swap page repositories increases fairly linearly as the length of the overload increases. However, cooperative swap also reads from the page repositories,

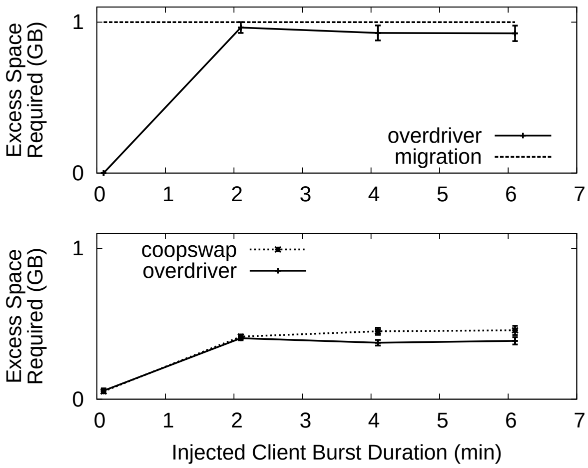



**Figure 10.** Amount of excess space required for migration and cooperative swap using Overdriver.

| Threshold(s) | Traffic(GB) | Space(MB) |
|---|---|---|
| 100 | 2.1 | 424 |
| 120 | 3.6 | 549 |
| 150 | 4.4 | 554 |
| 170 | 5.6 | 568 |
| 200 | 6.1 | 590 |

**Table 3.** For sustained overloads, as the threshold for migration increases, more overhead is accumulated while waiting for migration to happen. This table shows the maximum overhead for cooperative swap in terms of traffic generated and excess space required for a sustained overload.

which results in a further increase in network traffic. Migration, on the other hand, exerts a fixed constant, which is almost entirely written, regardless of the duration of overload. The fact that the value on the graph is fixed at 1 GB is because that is the memory allocation of the VM that is being migrated in this experiment. If a larger, 2 GB VM was migrated, we would see the migration line at 2 GB instead. Overdriver follows the cooperative swap line until the duration of overload exceeds its threshold, and it resorts to migration. This causes a sharp increase by the memory allocation of the VM (1 GB in this case) in the amount of data written to the network by Overdriver for sustained overload. In other words, to get the benefit for transient overloads, Overdriver pays a cost for sustained overloads that is proportional to the amount of time it used cooperative swap.

The amount of traffic over the network does not necessarily show the amount of excess space that is required for cooperative swap. For example, some pages may be written, read, then overwritten, without consuming more space at the page repository. Figure 10 quantifies the amount of excess space required for migration and cooperative swap. Even though cooperative swap may generate a virtually unbounded amount of network traffic for sustained overload, the amount of space that is required remains reasonable. In particular, the amount of space required does not increase above the amount of memory that was withheld because of oversubscription. For transient overloads, Overdriver must only use the modest amount of excess space for cooperative swap. For sustained overloads, however, Overdriver uses both.

Similarly to application performance, data center impact is also affected by the threshold that Overdriver uses. Table 3 quantifies the increases in overhead for a sustained overload that can be reduced by adaptively shrinking the threshold.

Finally, while Figures 9 and 10 quantify a tradeoff, and show how Overdriver navigates the tradeoff for a single overload, they only provide a glimpse into the tradeoffs that would appear if done at a data center scale. In particular, transient overloads are dominant, and so the modest savings that Overdriver achieves in this graph must be multiplied by the number of transient overloads, which should far outweigh the overhead incurred for the relatively rare sustained overloads; namely, Overdriver saves over a factor of four network bandwidth when compared to migration only. Furthermore, as discussed in Section 4, the differences in terms of how excess space is being used has far-reaching implications in terms of additional costs related to maintaining excess space.

## 6. Related Work

Overdriver uses aggressive memory oversubscription to achieve high data center utilization. This section describes some related work on memory oversubscription in a virtualized environment, VM migration, and network memory.

### 6.1 Memory Oversubscription

The ability to oversubscribe memory is common in modern virtual machine monitors. VMWare ESX server [31] was the first to describe memory balloon drivers which provided a mechanism to remove pages from a guest OS by installing a guest-specific driver. Xen [4] also contains a balloon driver. Balloon sizing can be done automatically with an idle memory tax [31] or as a more sophisticated driver inside the guest OS[12]. While ballooning allows changes to the allocation of machine pages, memory usage can also be reduced through page sharing techniques: mapping a single page in a copy-on-write fashion to multiple VMs. This technique was first described in VMWare ESX server [31] and has subsequently been extended to similarity detection and compression in Difference Engine [10]. Memory Buddies [35] is a migration-based strategy that tries to place VMs to optimize for page sharing. Satori [21] is examining new mechanisms to detect sharing opportunities from within the guest OS. These systems are great mechanisms for oversubscribing memory; however, a robust system to reactively handle and mitigate overload, like Overdriver, is necessary to maintain performance.

### 6.2 Migration

VM migration has become very popular and is touted as a solution free of residual dependency problems that have plagued process migration systems for years [20]. Stop-and-copy VM migration appeared in Internet suspend/resume [17]. Compression has been noted as a technique to improve the performance of migration, especially if the image contains lots of zeroes [26]. However, these techniques all impose significant downtimes for VMs. Live migration techniques, on the other hand, allow VMs to be migrated with minimal downtime. Push-based live migration techniques are the most popular; implementations include VMWare's VMotion [23], Xen [5], and KVM [16]. Pull-based live migration [12] has been evaluated and a similar mechanism underlies the fast cloning technique in SnowFlock [18].

We do not discuss addressing memory overload by spawning VMs. If a hosted service is structured to be trivially parallelizable, such that a newly initialized VM can handle new requests and share the load, spawning may be another viable mechanism to alleviate memory overload. Work has been done to maintain a synchronized hot spare [6] and to speed up cloning delays [18, 25].

There are many migration-based approaches that try to achieve various placement objectives, but they do not discuss maintaining excess space to handle overload. Khanna et al. [15], use heuristics to try to consolidate VMs on the fewest physical machines, while other approaches Entropy [11] and Van et al. [30] aim for an optimal consolidation with the fewest migrations. Sandpiper [34] uses migration to alleviate hotspots in consolidated data centers. In an effort to eliminate needless migrations, Andreolini et al. [3] use a trend analysis instead of triggering migration with a utilization threshold. Stage and Setzer [28] advocate long-term migration plans involving migrations of varying priority in order to avoid network link saturation.

### 6.3 Network Memory

Overdriver uses cooperative swap to address the paging bottleneck of overloaded VMs. Accessing remote memory on machines across a fast network has been recognized to perform better than disk for some time [2], and this concept has been applied to almost every level of a system. `memcached` [1] leverages network memory at the application level. Cooperative caching [7] gains an extra cache level in a file system from remote client memory. The Global Memory System [8] uses a remote memory cache deep in the virtual memory subsystem of the OS, naturally incorporating all memory usage including file systems and paging. Nswap [24] and the reliable remote paging system [19] focus specifically on sending swap pages across the network. Cellular Disco [9] is a hypervisor that uses network memory to borrow memory between fault-containment units called cells. Most similar to cooperative swap, MemX [13] implements swapping to the network for a guest VM from within the hypervisor. MemX is focused on extremely large working sets that do not fit in a physical machine's memory, whereas cooperative swap is designed to react quickly to overload bursts, many of which are transient. Other techniques to increase the performance of paging include the use of SSDs. However, addressing the paging bottleneck is not enough to handle the entire overload continuum, particularly sustained overloads.

## 7. Conclusion

As interest grows in oversubscribing resources in the cloud, effective overload management is needed to ensure that application performance remains comparable to performance on a non-oversubscribed cloud. Through analysis of traces from an enterprise data center and using controlled experiments with a realistic Web server workload confirmed that overload appears as a spectrum containing both transient and sustained overloads. More importantly, the vast majority of overload is transient, lasting for 2 minutes or less.

The existence of the overload continuum creates tradeoffs, in which various mitigation techniques are better suited to some overloads than others. The most popular approach, VM migration-based strategies, are well suited to sustained overload, because of the eventual performance that can be achieved. Cooperative swap applies to transient overloads, where it can maintain reasonable performance at a low cost.

We have presented Overdriver, a system that handles all durations of memory overload. Overdriver adapts its mitigation strategy to balance the tradeoffs between migration and cooperative swap. Overdriver also maintains application VM workload profiles which it uses to adjust its migration threshold and to estimate how much excess space is required in the data center to safely manage overload. We show, through experimentation, that Overdriver has a reasonably small impact on application performance, completing within 8% of the connections that a well-provisioned VM can complete under the continuum of overload, while requiring (under reasonable oversubscription ratios) only 15% of the excess space that a migration-based solution would need. Overdriver shows that safe oversubscription is possible, opening the door for new systems that effectively manage excess space in the data center for the purpose of handling overload, ultimately leading to a new class of highly efficient, oversubscribed data centers.

The Overdriver project webpage is available at: `http://overdriver.cs.cornell.edu`.

## Acknowledgments

This work was partially funded and supported by an IBM Faculty Award received by Hakim Weatherspoon and NSF TRUST. Also, this work was performed while Dan Williams was an intern at IBM T. J. Research Center in Hawthorne, NY.

## References

[1] memcached. `http://www.danga.com/memcached/`, May 2003.

[2] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, Feb. 1995.

[3] M. Andreolini, S. Casolari, M. Colajanni, and M. Messori. Dynamic load management of virtual machines in a cloud architecture. In *Proc. of ICST CLOUDCOMP*, Munich, Germany, Oct. 2009.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of ACM SOSP*, Bolton Landing, NY, Oct. 2003.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of USENIX NSDI*, Boston, MA, May 2005.

[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proc. of USENIX NSDI*, San Francisco, CA, Apr. 2008.

[7] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of USENIX OSDI*, Monterey, CA, Nov. 1994.

[8] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proc. of ACM SOSP*, Copper Mountain, CO, Dec. 1995.

[9] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proc. of ACM SOSP*, Charleston, SC, Dec. 1999.

[10] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proc. of USENIX OSDI*, San Diego, CA, Dec. 2008.

[11] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: a consolidation manager for clusters. In *Proc. of ACM VEE*, Washington, DC, Mar. 2009.

[12] M. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proc. of ACM VEE*, Washington, DC, Mar. 2009.

[13] M. R. Hines and K. Gopalan. MemX: supporting large memory workloads in Xen virtual machines. In *Proc. of IEEE VTDC*, Reno, NV, Nov. 2007.

[14] R. H. Katz. Tech Titans Building Boom. `http://www.spectrum.ieee.org/feb09/7327`, Feb. 2009.

[15] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performace management in virtualized server environments. In *Proc. of IEEE/IFIP NOMS*, Vancouver, Canada, Apr. 2006.

[16] A. Kivity, Y. Kamay, and D. Laor. KVM: The kernel-based virtual machine for Linux. In *Proc. of Ottawa Linux Symposium*, Ottawa, Canada, June 2007.

[17] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proc. of IEEE WMCSA*, Calicoon, NY, June 2002.

[18] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proc. of ACM EuroSys*, Nuremberg, Germany, Apr. 2009.

[19] E. P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *Proc. of USENIX Annual Technical Conf.*, San Diego, CA, Jan. 1996.

[20] D. S. Milojicic, F. Douglis, Y. Paindaveine, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, Sept. 2000.

[21] G. Miłoś, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *Proc. of USENIX Annual Technical Conf.*, San Diego, CA, June 2009.

[22] B. Mitchell, J. Rosse, and T. Newhall. Reliability algorithms for network swapping systems with page migration. In *Proc. of IEEE CLUSTER*, San Diego, CA, Sept. 2004.

[23] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proc. of USENIX Annual Technical Conf.*, Anaheim, CA, Apr. 2005.

[24] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A Network Swapping Module for Linux Clusters. In *Proc. of Euro-Par*, Klagenfurt, Austria, Aug. 2003.

[25] H. Qian, E. Miller, W. Zhang, M. Rabinovich, and C. E. Wills. Agility in virtualized utility computing. In *Proc. of IEEE VTDC*, Reno, NV, Nov. 2007.

[26] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proc. of USENIX OSDI*, Boston, MA, Dec. 2002.

[27] V. Shrivastava, P. Zerfos, K. won Lee, H. Jamjoom, Y.-H. Liu, and S. Banerjee. Application-aware virtual machine migration in data centers (to appear). In *Proc. of IEEE INFOCOM Mini-conference*, Shanghai, China, Apr. 2011.

[28] A. Stage and T. Setzer. Network-aware migration control and scheduling of differentiated virtual machine workloads. In *Proc. of ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Vancouver, Canada, May 2009.

[29] Standard Performance Evaluation Corporation. Specweb2009 release 1.10 banking workload design document. `http://www.spec.org/web2009/docs/design/BankingDesign.html`, Apr. 2009.

[30] H. N. Van, F. D. Tran, and J.-M. Menaud. Autonomic virtual resource management for service hosting platforms. In *Proc. of ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Vancouver, Canada, May 2009.

[31] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proc. of USENIX OSDI*, Boston, MA, Dec. 2002.

[32] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. In *Proc. of USENIX Annual Technical Conf.*, Anaheim, CA, Apr. 2005.

[33] S. Weber and R. Hariharan. A new synthetic web server trace generation methodology. In *Proc. of IEEE ISPASS*, Austin, TX, Mar. 2003.

[34] T. Wood, P. Shenoy, and A. Venkataramani. Black-box and gray-box strategies for virtual machine migration. In *Proc. of USENIX NSDI*, Cambridge, MA, Apr. 2007.

[35] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers. In *Proc. of ACM VEE*, Washington, DC, Mar. 2009.