Memory Elasticity Benchmark

Liran Funaro
Technion—Israel Institute of Technology
Haifa, Israel
funaro@cs.technion.ac.il

Orna Agmon Ben-Yehuda
Technion—Israel Institute of Technology
Haifa, Israel
ladypine@cs.technion.ac.il

Assaf Schuster
Technion—Israel Institute of Technology
Haifa, Israel
assaf@cs.technion.ac.il

ABSTRACT

Cloud computing handles a vast share of the world's computing, but it is not as efficient as it could be due to its lack of support for memory elasticity. An environment that supports memory elasticity can dynamically change the size of the application's memory while it's running, thereby optimizing the entire system's use of memory. However, this means at least some of the applications must be memory-elastic. A memory elastic application can deal with memory size changes enforced on it, making the most out of all of the memory it has available at any one time. The performance of an ideal memory-elastic application would not be hindered by frequent memory changes. Instead, it would depend on global values, such as the sum of memory it receives over time.

Memory elasticity has not been achieved thus far due to a circular dependency problem. On the one hand, it is difficult to develop computer systems for memory elasticity without proper benchmarking, driven by actual applications. On the other, application developers do not have an incentive to make their applications memory-elastic, when real-world systems do not support this property nor do they incentivize it economically.

To overcome this challenge, we propose a system of memory-elastic benchmarks and an evaluation methodology for an application's memory elasticity characteristics. We validate this methodology by using it to accurately predict the performance of an application, with a maximal deviation of 8% on average. The proposed benchmarks and methodology have the potential to help bootstrap computer systems and applications towards memory elasticity.

CCS CONCEPTS

• **General and reference → Metrics**; **Evaluation**; **Experimentation**; **Performance**; *Measurement*.

KEYWORDS

Cloud, Benchmark, Vertical Elasticity, RAM, Memory

ACM Reference Format:
Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. 2020. Memory Elasticity Benchmark. In *The 13th ACM International Systems and Storage Conference (SYSTOR '20), June 2–4, 2020, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3383669.3398277

1 INTRODUCTION

Today's cloud providers make every effort to improve their resource utilization and thereby make more money off the same hardware. Rigid allocation prevents them from utilizing the hardware efficiently, so they offer clients various options for resource elasticity [11]. These elastic options allow clients to change their resource consumption on the fly by exploiting resources that are momentarily unused by other clients.

Resource elasticity is seamless in services such as Application-as-a-Service (AaaS) and serverless computing. Here, clients rent a black-box execution environment that exposes a limited application programming interface (API) they can use. The environment's resource consumption and workload distribution are controlled by the provider. Thus, the client shares resources with other clients who occupy the same environment. In such services, the provider handles the client's resource elasticity, relieving the client of this burden.

However, these environments may not suit all applications. Some clients need a broader API, have a proprietary application, or use an uncommon application that is not supported by the provider. Other clients may have specific performance requirements that the provider is unable to guarantee. For example, clients might need to be physically closer to their data or maintain some continuity between runs.

Clients who require more than what is provided by AaaS and serverless computing will deploy their applications using Infrastructure-as-a-Service (IaaS) and Container-as-a-Service (CaaS). In such services, clients rent a bundle of rigid, exclusive, resources in the form of a single virtual machine

SYSTOR '20, June 2–4, 2020, Haifa, Israel
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7588-7/20/06…$15.00
https://doi.org/10.1145/3383669.3398277

(VM) or an OS container. Many IaaS and CaaS providers offer CPU elasticity in the form of *burstable performance*, which offers a basic level of CPU performance but can 'burst' to a higher level when required. Under certain conditions, this lets clients use more CPU than their initial allocation, in the same VM/container. These providers include Google [13], Amazon [3], Azure [24], CloudSigma [7], and RackSpace [27].

With the current proliferation of CPU elasticity schemes, CPU utilization is adequately optimized and more clients can be allocated to the same physical servers [11]. This leaves memory as the bottleneck resource: it is an expensive resource that limits machine occupancy. Memory elasticity schemes should be a natural extension to CPU elasticity, allowing clients to use more memory in the same VM/container than their initial memory allocation. Clients who can tolerate a temporary memory shortage could benefit from these schemes by lowering the amount of memory they rent exclusively. They could then compensate for the reduced requirement by bursting when they really need more memory. For example, clients who can postpone memory-intensive phases in their operations, can reserve a small initial amount of memory and make use of more memory whenever it is available (e.g., for maintenance operations). This enables clients to time-share memory and the provider can squeeze more applications onto the same hardware [11].

To achieve this, clients need *memory-elastic applications* that can change their maximal memory usage on the fly and whose performance is proportional to their memory usage. Unfortunately, memory-elastic applications are scarce. Although developers usually strive to make their application's performance proportionate to its CPU and bandwidth availability, most applications are not designed with memory elasticity in mind. Developers generally address only the maximal memory footprint of their application. They treat it as constant or a value dictated by the current application workload. The operating system's swapping mechanism allows seamless application operation when the available memory is insufficient, but this results in a graceless performance degradation; even a minor memory loss may degrade the performance significantly.

Why do developers toil towards making performance scale nicely with the CPU and bandwidth, but neglect doing this for memory? Developing memory-elastic applications requires more work. With a proliferation of memory-elastic systems, developers could be incentivized to make this effort, as they did with CPU elastic applications.

Research has been done into systems that allow frequent memory allocation changes [2, 14]. However, without memory-elastic applications, such systems cannot be used to their full potential and will not be accepted by the commercial community. As early as 2010, cloud provider CloudSigma allowed clients to change their memory allocation and billing during runtime. Unfortunately, other commercial cloud providers did not follow in their footsteps and this option is no longer promoted on the CloudSigma web-page.

We're seeing a circular dependency problem between memory-elastic applications and systems that require and incentivize such properties. When one of these elements is scarce, there is no incentive to develop the other because real benefit only comes when both elements exist.

A proof that memory-elastic applications exist or can be created is essential to break this circular dependency. **Our first contribution** is a set of memory-elastic applications with a memory to performance trade-off (Section 4.3).

Once the circular dependency problem is solved, and elastic memory systems and applications exist commercially, a language and method for quantifying application elasticity will help clients choose a resource bundle that best suits their application. In addition, quantifying an application's elasticity will help cloud providers test and optimize their systems. **Our second contribution** comprises a methodology and terminology for evaluating memory elasticity (Section 3). These specify how to determine a memory elasticity score for each application that can be used as a *memory elasticity benchmark*.

The methodology is implemented as an open-source *memory elasticity evaluation framework* (Section 4). We validated the evaluation process using our framework (Section 5) over a set of memory-elastic applications. The results show that our memory elasticity score can accurately predict an application's performance, with an average deviation of 8%.

The methods and applications we introduce, along with our memory elasticity metrics, will allow clients to choose less expensive memory elastic schemes and reduce costs. Accordingly, the number of IaaS clients per server will no longer be constrained by memory, allowing elastic CPU allocation schemes to demonstrate their full potential of nearly 80% CPU utilization [11].

2 MEMORY ELASTICITY METHODS

This section presents several application properties that can be used to allow memory elasticity.

2.1 Applications with Resource Trade-off

Mechanisms that were designed to allow trade-off between memory and other resources can be used to provide memory elasticity.

Memory as cache: Some applications use the RAM to cache computation results, network traffic, and so on (e.g., using memcached[1]). They can increase their memory footprint when memory is cheap or more available to the application. For example, an application might switch to caching mode for network requests when memory is abundant and avoid caching when high bandwidth is more available or cheaper than memory. Since caches are designed to drop data frequently, cache-enabled applications are already designed to withstand data loss when the memory footprint decreases.

Similarly, applications that rely on the operating system's page cache to reduce the storage latency (e.g., PostgreSQL[2]) may also be affected by how much memory is available to the operating system. They can seamlessly improve their performance when more memory is available to the operating system. For example, PostgreSQL exhibits performance proportional to the memory availability, as presented in Figure 1.

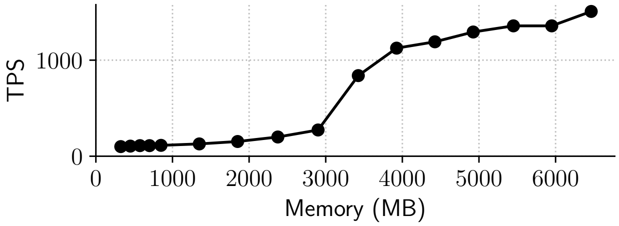


Figure 1: PostgreSQL tested with Pgbench [20] workload, with 40 clients, while changing the memory allocation on the fly.

Intermediate calculations: Applications that use huge amounts of on-disk data (e.g., databases, Hadoop) can use larger memory buffers to reduce disk access and speed up temporarily data-heavy operations, such as sorting and large matrix multiplication.

Garbage collected memory: Applications with automatic memory management (e.g., Java applications) may need fewer garbage-collection cycles with a larger heap, and improve their performance as depicted in Figure 2. On the other hand, when the memory is too large, the garbage collection might take longer, as shown by Soman et al. [30].

2.2 Memory-Aware Applications

Memory-aware applications adjust their memory consumption according to the available memory observed during their initiation period, but cannot adjust it during runtime. Specifically, most of the commonly used memory trade-offs we mentioned (subsection 2.1) are predefined and implemented as memory-aware applications. Memcached only allows the cache size to be set at startup, PostgreSQL's temporary buffers are defined using a static configuration file, and the Java-Virtual-Machine (JVM) allows setting the minimum and maximum heap size only using the command-line parameters at startup.

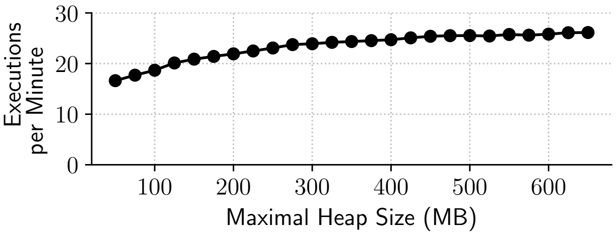


Figure 2: Lusearch benchmark from the DaCapo benchmark suite [5] with different limits on the maximal heap size in an off-the-shelf JVM. These measurements were produced by restarting the JVM for each maximal heap size.

These applications can be made memory-elastic by restarting them when the memory changes, but this solution is not suitable when the application needs to be continuously available. With a small effort, as shown in subsection 4.3, these applications can be tweaked to become memory-elastic.

2.3 Multiple Short-Lived Jobs

Some applications have multiple short-lived jobs, each with different memory requirements. For example, web servers might require a certain memory to handle each session. They may be able to handle more concurrent sessions when more memory is available. To deal with lack of memory, they can cap the number of concurrent sessions; thus, they trade off memory for latency and throughput.

Another example is batch workload schedulers, such as SLURM or Sun Grid Engine, which execute many short-lived jobs, each with a predefined resource requirement. The scheduler can adapt the concurrency according to the actual memory allocation, running more concurrent jobs when more memory is available. Alternatively, if the jobs are memory-aware, it can keep the concurrency constant and adapt the memory requirements of each job such that the combined memory requirements of all running jobs will match the allocation. Moreover, it can combine these two strategies.

2.4 Rigid Applications

Applications that cannot use any of the above techniques will resort to memory-swapping once the available memory is not sufficient for their memory required footprint. This

[1] Memcached is a popular, open-source, in-memory data store used to reduce the number of times an external data source must be read.
[2] PostgreSQL is a database application.

option is usually not advisable as it suffers from inadequate performance.

3 MEMORY ELASTICITY METRICS

Any developer who implements a mechanism from the previous section will naturally want to measure its effect on the application's memory elasticity. Developers are used to measuring and comparing metrics such as throughput, goodput, latency, jitter, and load capacity. Such metrics quantify the application's performance and enable its comparison to similar applications or to other versions of the same application. But can we use them to quantify the application's elasticity?

We could compare the performance of two applications under the same dynamic memory conditions and consider the one with the better results as more memory-elastic. However, the results may be sensitive to the order or frequency of memory allocations. A single scenario or even several scenarios do not necessarily indicate how the applications behave under untested scenarios. This is because we try to infer memory elasticity from observations of metrics that only hint about elasticity, but do not measure it directly.

Our goal is to quantify an application's behavior in a dynamic memory scenario and compare it to other applications, using metrics that directly relate to memory elasticity. We target metrics that capture the characteristics that make an application more memory-elastic.

In this section, we present our novel memory elasticity metrics, which predict how well an application can utilize momentarily available memory, assuming it has the necessary load that requires the memory. First we define a set of *static metrics* to describe the application's elasticity, without considering the implications of changing the memory allocation during runtime. This part determines the memory domain in which the application has the potential to be memory-elastic. If the application has such a domain, a second set of metrics can then be defined within this domain. These *dynamic metrics* quantify how well the application responds within the elasticity domain to dynamic memory changes—changes made during runtime. Finally, in section 4, we describe the experiments we designed to compute these metrics for each application.

In addition to the elasticity metrics, which are comparable across applications, we define elasticity characteristics that can be used by clients to configure their VM and their application.

3.1 Static Metrics

First, we define a static memory→performance function (P_{mem}) that describes the performance of the application given a static memory allocation. Then, we define the application's *elasticity domain*. We denote by mem_L the memory allocation that is sufficient for the application to yield the minimum required performance. This might be the memory below which thrashing occurs or it might be defined by a service level agreement (SLA). We denote by mem_H the maximal memory allocation that yields any performance improvement over a smaller memory allocation. If mem_L is identical to mem_H, the application is simply inelastic, in which case any other elasticity metric is irrelevant. If the application might be elastic, we define its *elasticity domain* as $[mem_L, mem_H]$ and its *elasticity range* as $mem_H - mem_L$. An application with a larger elasticity range can withstand more dynamic scenarios and thus is considered more elastic. Therefore, our first elasticity metric is the application's elasticity range.

We also define the *improvement factor per memory unit* (IFMU) in the elasticity domain as:

$$IFMU = \frac{\frac{P_{mem}(mem_H)}{P_{mem}(mem_L)}}{mem_H - mem_L} \ . \qquad (1)$$

An application with greater IFMU has the potential to gain more from a dynamic memory scenario. Hence, the application's IFMU is our second elasticity metric.

An example of a performance function (P_{mem}) is illustrated in Figure 3. In this example, the application needs at least 1 GB of RAM (mem_L = 1GB), and gains no performance improvement beyond 4.5 GB of RAM (mem_H = 4.5GB). Thus, its elasticity domain is 1 GB to 4.5 GB, its elasticity range is 3.5 GB, and its IFMU is about 3 per GB.

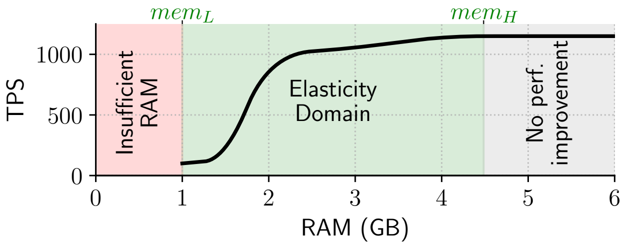


Figure 3: Illustrating performance measurements of an application.

3.2 Dynamic Metrics

When the memory changes during runtime, the performance is not necessarily affected immediately. An application might require some time to utilize the added memory. For example, it takes time to fill the cache and it takes even longer to notice an improved hit-rate due to the re-use of cached items. Upon memory reduction, an application might need to prepare for eviction a few seconds ahead of the change, to avoid memory swapping.

Agmon Ben-Yehuda et al. [2] defined T_{mem} as an upper bound on the time of the transient performance before stabilization. We extend this scalar definition of T_{mem} to a function with two variables $T_{mem}(s, d)$, where s denotes a source memory allocation and d denotes a target (destination) memory allocation.

Figure 4 illustrates T_{mem} in two scenarios: memory is increased in the first and decreased in the second. In phase A (starts at time $= t_0$) the memory allocation is α, with an average performance of $P_{mem}(\alpha)$. As phase B begins (time $= t_1$), the memory allocation changes to β, but the performance stabilizes in $P_{mem}(\beta)$ only after $t_2 - t_1$ seconds (time $= t_2$). This time is defined as $T_{mem}(s = \alpha, d = \beta) = t_2 - t_1$. To generalize, when $s < d$ and the memory is increasing, $T_{mem}(s, d)$ indicates the period starting with the allocation change and ending with the application reaching the statically measured performance ($P_{mem}(d)$).

To prepare for the decreased memory allocation in phase C (time $= t_3$), the application starts releasing memory ahead of the memory change (time $= t_4$). This time is defined as $T_{mem}(s = \beta, d = \alpha) = t_4 - t_3$. To generalize, when $s > d$ and the memory is decreasing, $T_{mem}(s, d)$ indicates how far ahead of the memory change the application started to modify its state to accommodate the updated memory allocation and reduce its performance accordingly. We assume here that the application properly prepares for the memory reduction and manages to release its memory before the allocation is applied. If this is not the case, and the application fails to release its memory, it is considered a bug or a misconfiguration.

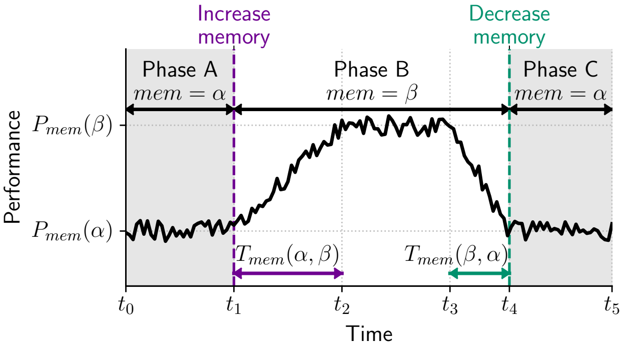
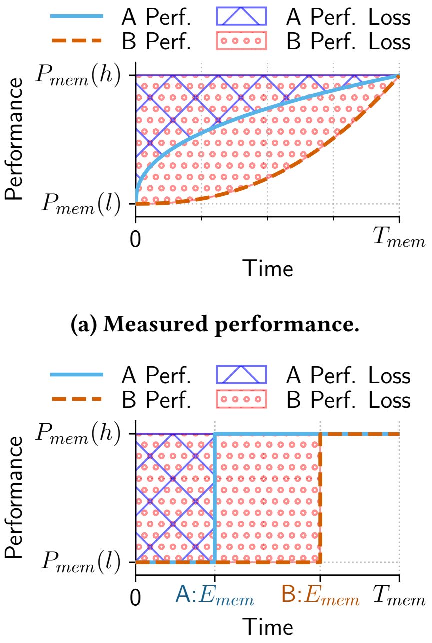


Figure 4: The definition of T_{mem} given an application's performance under dynamic conditions.

T_{mem} is not a good enough metric to compare different applications. During the transient period, application A may reach 90% of the maximal performance after a short period; after this period, it slowly increases to $P_{mem}(d)$. Application B may have the exact same T_{mem} and P_{mem} as A, but during the transient time its performance is low for most of the time. It only increases near the end of the transient, as illustrated in Figure 5a.



(a) Measured performance.

(b) Fictional performance.

Figure 5: Illustration of two applications with the same T_{mem} and P_{mem}, but with a different performance loss during the transient period.

The clients using these applications pay for memory that does not immediately translate to their expected performance for the same duration (T_{mem}). However, a client using application A loses more performance over time than a client using application B, as illustrated by the filled areas above the performance curves in Figure 5a. These filled areas represents the aggregate *performance loss* and are formally defined by

$$L_{mem}(s, d) = \int_0^{T_{mem}(s,d)} (P_{mem}(\max\{s, d\}) - p(x))dx\ , \quad (2)$$

where $p(x)$ is the performance of the application at time $x \in [0, T_{mem}(s, d)]$ during the memory transition.

To account for this misrepresentation of actual performance, we define the effective T_{mem}, denoted by $E_{mem}(s, d)$, which is comparable across applications. Consider a fictional scenario, in which the performance changes abruptly between the two performance levels with a time delay, as illustrated in Figure 5b. In this fictional scenario, the time delay is chosen so the performance loss in the fictional scenario is identical to the measured performance loss (Equation 2). $E_{mem}(s, d)$ is defined as that time delay. It is the aggregate performance loss for the memory change, divided by the performance difference between the levels. Formally,

$$E_{mem}(s,d) = \frac{L_{mem}(s,d)}{|P_{mem}(d) - P_{mem}(s)|} \ . \tag{3}$$

Similarly, upon a decrease in memory allocation, the performance drops in the fictional scenario to $P_{mem}(d)$ seconds before the memory allocation actually changes.

Application A has a shorter $E_{mem}(s,d)$ compared with B, as illustrated in Figure 5b. Because their P_{mem} functions are identical, we can directly infer that application A has less performance loss, and thus it is more memory-elastic than application B. Therefore, we consider an application with a shorter $E_{mem}(s,d)$ as being more memory-elastic. The application's E_{mem} function is our third and final memory elasticity metric.

4 EVALUATION

In this section we explain how developers can measure their own application's memory elasticity characteristics and metrics. Then, we design a validation method for these metrics. Finally, we discuss the applications we evaluated using these metrics.

4.1 Measuring the Elasticity Metrics

In section 2, we defined three key elasticity metrics that are comparable across applications: elasticity range, IFMU, and E_{mem}. In addition, we defined different characteristics such as elasticity domain, P_{mem} and T_{mem}, which can be used by clients to configure their VM and their application. The following are the experiments we designed to measure these metrics and characteristics for each application.

Static Metrics. To find each application's characteristics (mem_L, mem_H and P_{mem}), we perform a few incremental tests. First, we roughly estimate mem_L and mem_H, and then we determine their exact values.

We perform *static test*s in which we choose a high-enough static memory allocation as an initial guess for mem_H, and test the application with this allocation. This high-enough memory allocation can be estimated on the basis of preliminary knowledge (e.g., the application's working set size in the chosen workload). Otherwise, this test can be repeated with different higher memory allocations, until the performance is similar in at least two different memory allocations.

In this test we also measure the *warm-up time*: how long it takes the application to reach its maximal performance and maximal memory usage. The warm-up time is required for future experiments.

We need additional static tests in order to choose a low static memory allocation that will serve as an initial guess for mem_L. This is chosen such that the application can still function properly and the OS swapping mechanism is not activated. Avoiding swapping is a strict requirement for this test, since guest swapping may swap out other applications' memory, or even that of the operating system. This would increase the memory available to the tested application and possibly lead to unexpected results.

Next, we perform a *pyramid test*, in which an application starts working on a guest virtual machine (VM) with a maximal memory allocation (i.e., our initial guess of mem_H). We allow the application to warm up for the warm-up duration we found in the static tests. Then we gradually decrease the memory allocation, in steps, until we reach our initial guess of mem_L. Throughout each step, the memory allocation remains constant. To avoid measuring transient effects, each step includes enough time to measure the application's performance in a reliable manner after the warm-up time. The time that is considered to be sufficient is usually dictated by the workload. For example, PostgreSQL benchmark (pgbench) recommends running a test for at least a few minutes to get reproducible results.

To validate that the application's performance in a certain memory allocation step is not affected by the step from which it descended, we repeat the process in reverse—gradually increasing the allocation by steps to the maximum. We verify that the performance in the increasing phase is similar to the performance in the decreasing phase for each tested memory allocation.

We also record the application's average performance and its standard deviation for each memory allocation. We then use these measurements to determine the exact mem_L and mem_H values, and the P_{mem} function for the application.

For the results presented in this paper, we tested the guest memory allocations in steps of 512 MB: 1024 MB, 1536 MB, 2048 MB, and so forth, until there was no performance improvement. The lowest memory allocation we could measure without swapping was 896 MB.

Dynamic Metrics. First, for each application we determine its *safe retreat time*: how much time ahead of the memory drop the application must start reducing its memory consumption to avoid swapping. To this end, we conduct a *drop-test* in which the memory allocation drops from mem_H to mem_L. Here, the application starts changing its state as soon as it is notified of the memory allocation drop. Our default settings gave the application a prior notice of 30 seconds. We then measure how long it takes the application to reach the lower memory state. This duration is used as the application's safe retreat time for the subsequent tests.

To generate $T_{mem}(s,d)$ and $E_{mem}(s,d)$ for each application, we perform an *oscillation test.* In such a test, the memory allocation oscillates between two values for five cycles, and the performance is recorded. We performed an oscillation test for each application, for any pair of values taken from the values tested in the pyramid test.

As previously defined (subsection 3.2), when we increase the memory, the transition period lasts from the application of the new memory allocation to the stabilization of the performance. We define that the performance stabilizes when the application's average performance over a predefined time-window reaches $P_{mem}(d)$ for the first time after the allocation change. The time-window size is different for each application and is chosen according to the application's characteristics (e.g., PostgreSQL requires a window of a few minutes to mask the measurement noise).

When we decrease the memory, the transition period lasts from the time the application proactively prepares for the lower memory allocation until the new memory is allocated. This definition relies on the valid measurement of the safe retreat time. Indeed, we validated in our experiments that the safe retreat time was sufficient. We also log the application's internal memory state during the experiment for this purpose.

For each application, for each transition in each test, we compute $T_{mem}(s,d)$ and $E_{mem}(s,d)$ from the performance measurements. Then, for each application, we compute the average values of $T_{mem}(s,d)$ and $E_{mem}(s,d)$ for each pair of source and target (s,d), to be used as the $T_{mem}(s,d)$ and $E_{mem}(s,d)$ for that application.

4.2 Validating our Metrics

To validate our memory elasticity metrics, we needed to show they are able to predict the application's performance in any dynamic memory scenario. To start, we randomly generated multiple benchmark traces of differences in memory allocations; these indicated the histories of how much memory was added or taken from the previous allocation. The traces represented different scenarios, characterized by two properties:

(1) *Rate*: the number of memory changes per hour.
(2) *Amplitude*: the maximal difference between two consecutive memory allocations.

Since the purpose of this set of experiments was to validate the dynamic properties against trace results and not compare applications, we tested each application on several of these traces in which its elasticity could be expressed. The rates were limited by the application's T_{mem}. That is, the allocation cycle time ($\frac{360}{\text{rate}}$ seconds) had to be greater than the maximal T_{mem}. The amplitudes were limited by the application's elasticity range.

Then, we calculated the average actual performance of the application over the entire period of the experiment, excluding the warm-up time at the beginning of the experiment. We calculated the expected "ideal" average performance, using the *static profiler* by applying the static memory performance function (P_{mem}) to the application's memory allocation. We also calculated the expected "realistic" average performance, using the *elastic profiler*, which calculates the performance in the fictional scenario, inferred by the application's E_{mem} function.

To calculate how accurate each of our (ideal and realistic) predictions were compared with the actual performance, we used the following norm:

$$100 \cdot \left| \frac{\text{predicated performance}}{\text{actual performance}} - 1 \right| , \tag{4}$$

which is the maximal deviation (in percentage) from the actual performance.

Our memory elasticity metrics are meaningful if they can be used to accurately predict the application's performance with good probability. If these metrics are valid, they will allow us to evaluate and quantify an application's elasticity by performing four simple tests: static-test, pyramid-test, drop-test, and oscillation-test, without the need to repeat this validation process for future applications.

4.3 Applications and Benchmarks

We wanted to identify applications that could be used as elastic benchmarks. Our preference was for memory-elastic off-the-shelf applications, but we also modified applications or tweaked their settings to enable memory-elasticity.

Most benchmarks suites are composed of a benchmarking utility that runs different applications. These benchmarking utilities generally execute an application for a short period and measure the average performance over that period (e.g., DaCappo [5], SPEC CPU [6]). This operation is repeated to produce statistically significant results. However, this standard mode of operation is not fitting for the measurement of performance over different runtime phases, especially while changing memory allocations. To this end, the application needs to continuously run over a period of time that is significantly longer than the transient effects of a memory allocation change.

To evaluate elasticity we require a benchmarking utility that executes an application for any time period with a constant load and reports frequent performance statistics during the benchmark runtime. This could work well, for example, to test an elastic application that is an always-on, always-available service, responding to client requests. Ideally, the application being tested should be able to change its mode or adapt its memory utilization given notification of an upcoming memory allocation, or do so seamlessly without requiring a hint.

We tested the following applications.

Memcached [9] is a memory cache service that runs in the background alongside another application. It can be used to cache computation results, network responses, and so forth. The performance function (P_{mem}) of off-the-shelf memcached resembles a step function and is typical of the operating system's efforts to handle memory pressure through swapping [2]. We used the elastic adaptation of memcached[3] [1, 2, 25], which supports memory elasticity by changing its memory footprint upon receiving a command via a socket. Memcached has its memory arranged in linked lists of slabs, for which it maintains metadata, so that it can tell which slab to overwrite. The elastic memcached used this mechanism to choose which memory slabs to lose when the memory footprint needed to be decreased, along with the malloc_trim() function, which forces libc to release memory from the heap back to the operating system. As its workload driver, we used memaslap with concurrency of 20, a window size of 100K, and 90% get requests (the rest are set requests).

MemoryConsumer[4] [2] is a dedicated dynamic memory benchmark that accesses random pages in a predefined memory region. When allocated less memory, it is informed of the change and only attempts to access memory pages it can reach (without the risk of touching out-of-boundary pages). The application initiates a 'sleep' operation to artificially prevent access to any pages beyond the memory it knows it has; this reduces the throughput without causing any additional issues resulting from actually touching swapped out pages. Hence, its hit rate increases linearly with the available memory. The benefit of this artificial benchmark is that it is designed to have an almost zero T_{mem}, with highly reproducible performance measurements. We tested it with a constant load of 10 threads accessing the memory simultaneously.

4.4 Implementation Details

We implemented the benchmarking framework in Python 3.7. Some of the code is based on the evaluation framework for Ginseng [2, 10], which is based on the memory overcommitment manager (MOM) by Litke [22]. The code is available as open source at: github.com/liran-funaro/elastic-benchmarks.

4.5 Experimental Setup

We evaluated our framework on a machine with 16 GB of RAM and 2 Intel(R) Xeon(R) E5-2420 CPUs @ 1.90 GHz with 15 MB LLC. Each CPU had 6 hyper-threaded cores, for a total of 24 hardware threads. Each application was set up in advance in a qcow2 image of a guest virtual machine, running on a QEMU/KVM instance. Each guest VM was allocated with 4 cores, and was pinned to cores on a single NUMA node. We controlled the guest memory using the memory balloon module [31]. The host ran Ubuntu 16.04.1 with kernel 4.8.0-58-generic #63, and the guests ran Ubuntu 18.04.2 with kernel 4.15.0-50-generic #54. To reduce measurement noise, we disabled hyper-threading, pstate, and ksm in the host, and tested one benchmark at a time.

5 RESULTS

In this section, we analyze each of the applications by evaluating the application's static and dynamic metrics, and validating them.

5.1 Memcached

The static evaluation of the elastic memcached is presented in Figure 6. This application's memory domain is from 896 MB to 3584 MB, making its memory range 2688 MB. Our static evaluation suggested that memcached requires 7 minutes of warm-up time and its improvement factor (IFMU) is 1.8 per GB.

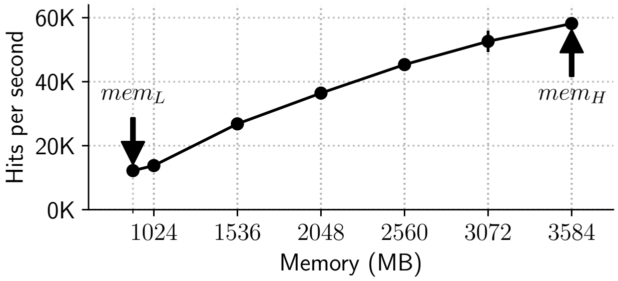


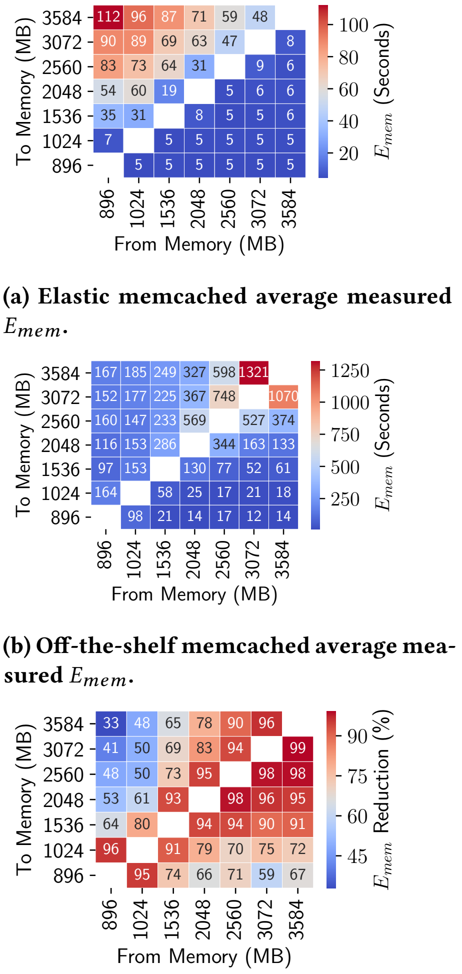
Figure 6: Memcached performance ("get" hits per second) as a function of allocated memory.

The dynamic evaluation of memcached is depicted in Figure 7. The drop-test suggests that memcached needs about 6 seconds of safe retreat time. We can see that for some parameters when $d < s$, $E_{mem}(s, d) > 6$. This is although $T_{mem}(s, d) = 6$ for these parameters. This is because memcached has to do extra work in order to choose the least used items to release. This reduces its performance below the average performance for the target memory and makes the effective transient period longer than the actual one. When $d > s$, larger memory changes induce longer E_{mem}, as expected.

Off-the-shelf memcached had the exact same range, IFMU, and P_{mem} as the elastic one. However, its dynamic evaluation showed that it suffers far more from transitions, as depicted in Figure 7b. The measured E_{mem} of the elastic memcached is at least 33% shorter than that of the off-the-shelf one, and for most of the transitions, it is at least 90% shorter (Figure 7c). Given that the static properties of the application did not change, we can clearly determine that the effort to modify memcached to be memory-elastic has been useful.

[3] Available from: https://github.com/liran-funaro/memcached.

[4] Available from: https://github.com/liran-funaro/memory-consumer-cpp.



(a) Elastic memcached average measured E_{mem}.

(b) Off-the-shelf memcached average measured E_{mem}.

(c) E_{mem} reduction (percentage) when using the elastic memcached compared to the non-elastic one.

Figure 7: Elastic and off-the-shelf memcached oscillation test results.

5.2 Memory Consumer

The static evaluation of memory consumer is depicted in Figure 8. This application’s memory domain is 896 MB to 2048 MB, making its memory range 1152 MB. The static evaluation suggests that memory consumer needs less than 1 minute of warm-up and it has an improvement factor (IFMU) of 2.2 per GB.

The dynamic evaluation of memory consumer is depicted in Figure 9. The drop-test suggested that memory consumer needs about 3 seconds of safe retreat time. Memory consumer has minor E_{mem} values (compared with any memcached version), because it immediately allocates and releases memory.

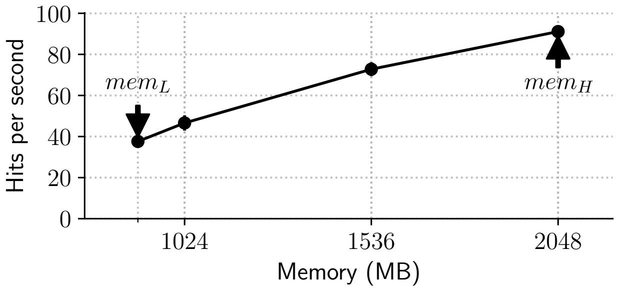


Figure 8: Memory consumer performance as a function of allocated memory.

Memory consumer was designed specifically to have such low E_{mem} values.

The E_{mem} drop values ($d < s$) are longer than the increase values ($d > s$), which is counterintuitive. In a real application, we expect that it takes more time to occupy the memory than to release it. This application, however, allocates the memory immediately, but releases the memory ahead of the memory allocation, to be on the safe side.

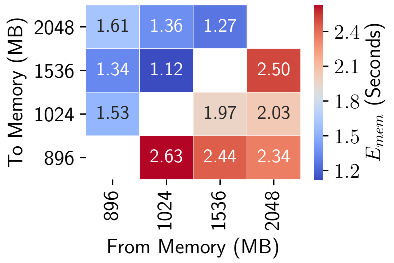


Figure 9: Memory consumer average measured E_{mem}.

5.3 Validation

In addition to the above tests, for each application, we validated the dynamic metrics according to the method described in subsection 4.2.

5.3.1 Memcached. The accuracy distribution of the static and the elastic profiler is presented in Figure 10. The static profiler’s accuracy has an average deviation of 13% from the actual performance. Its accuracy has a high variation of up to 40% as seen in Figure 10. The elastic profiler, however, is twice as accurate. It predicts the performance of memcached with an average deviation of 8% from the actual performance over all verification parameters, and with no deviation greater than 20%.

5.3.2 Memory Consumer. The accuracy distribution of the static and the elastic profiler of memory consumer is depicted

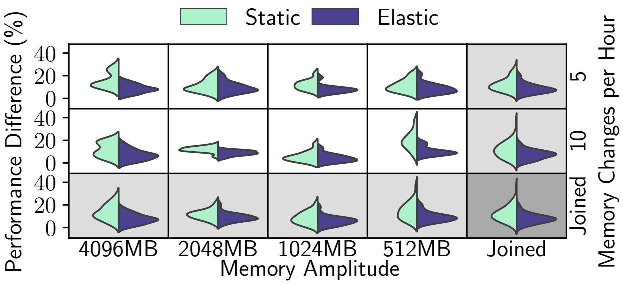


Figure 10: Memcached profiler prediction accuracy distribution. Each rectangle shows the results for different parameters according to its row and column. The joined row shows all the experiments with any memory amplitude, and the joined column shows all the experiments with any change rate.

in Figure 11. The static and the elastic profilers can predict the performance with an average deviation of less than 1% from the actual performance. The elastic profiler cannot improve much over the static one because the transient period for memory consumer is negligible. We only validated this application with one combination of parameters because it produced sufficient accuracy. Thus, we would not gain more insight from less dynamic scenarios.

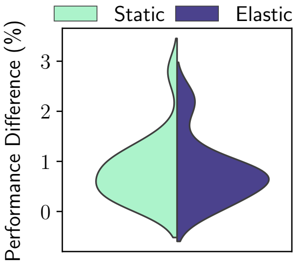


Figure 11: Memory consumer profiler prediction accuracy distribution with a rate of 300 memory changes per hour and a maximal amplitude of 2 GB.

6 RELATED WORK

Since the time memory balloons were developed by Waldspurger [31], several systems and concepts have been built to enable system administrators to dynamically reallocate RAM among virtual machines (e.g., Litke [22], Shrawankar and Eram Shaikh [29], Gordon et al. [14, 18], Nathuji et al. [26], Dolev et al. [8], and Heo et al. [15]).

Many researchers have addressed the issue of evaluating a system's elasticity [12, 16, 17, 19, 32]. However, little work has been done on the evaluation and development of memory elasticity for applications. This is in contrast with the application CPU elasticity, which was studied extensively, as reviewed by Kumar [21].

The nom profit maximizing operating system has an interactive, on-the-fly configurable network stack, which adapts to different monetary conditions according to its service level agreement [4]. All these systems could be better evaluated given a solid benchmark suite for elastic memory.

Salomie et al. [28] implemented a kernel module that supports ballooning pages right out of the application's memory to the host (i.e., application level ballooning) and demonstrated it on a Java-virtual-machine (JVM). Such a solution may be helpful for transferring memory faster, but it depends on a specific operating system and requires the installation of a kernel module. This approach increases the coupling between software layers, which complicates the adoption of the application.

The Automatically Tuned Linear Algebra Software (ATLAS) [33] is memory aware: it configures itself by benchmarking to optimally use the hardware it runs on, considering mainly the size of the cache. It does so upon installation and does not change the configuration on-the-fly.

7 CONCLUSIONS AND FUTURE WORK

We introduced a first-ever validated method for the measurement of application elasticity in a comparable manner. This method breaks the stalemate that has trapped the development of memory elastic applications and systems. It opens the door to an era of yet unseen optimizations, where memory can be considered an elastically used resource and its elasticity can be measured.

We developed a framework to demonstrate our method. Our evaluation demonstrated that the E_{mem} values of memory consumer are significantly lower than memcached values; this indicated that memory consumer has a higher elasticity with regard to the transient period. In addition, the elastic memcached had significantly shorter E_{mem} values compared to the non-elastic version.

We verified our framework by showing it can predict the performance of an application under dynamic memory changes with high accuracy. Our evaluations showed an average deviation of 8% and 1% of the actual performance for memcached and memory consumer, respectively.

Additional applications will allow more clients to use this framework to evaluate their memory elasticity bundle offerings, and allow providers to easily evaluate new memory-elastic systems. We note that PostgreSQL and the Java virtual machine are good candidates for these efforts. Adding these applications and more to the framework is left for future work.

ACKNOWLEDGMENTS

We thank Chani Sacharen, Tamar Camus, Eyal Posner, Danielle Movsowitz and Shunit Agmon for fruitful discussions. This work was partially funded by the Hasso Platner Institute, and by the Pazy Joint Research Foundation.

REFERENCES

[1] Shunit Agmon, Orna Agmon Ben-Yehuda, and Assaf Schuster. 2018. Preventing Collusion in Cloud Computing Auctions. In *Proceedings of the 15th International Conference on Economics of Grids, Clouds, Systems, and Services (GECON '18)*. Springer.

[2] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. 2014. Ginseng: Market-driven Memory Allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Vol. 49. ACM.

[3] Amazon. 2018. Amazon EC2 Burstable Performance Instances. https://aws.amazon.com/ec2/instance-types/#burst. Accessed: 2018-07-25.

[4] Muli Ben-Yehuda, Orna Agmon Ben-Yehuda, and Dan Tsafrir. 2016. The Nom Profit-Maximizing Operating System. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '16)*. ACM, New York, NY, USA, 145–160. https://doi.org/10.1145/2892242.2892250

[5] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, Eliot, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, 169–190. https://doi.org/10.1145/1167473.1167488

[6] James Bucek, Klaus-Dieter Lange, et al. 2018. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 41–42.

[7] CloudSigma. 2018. CloudSigma Cloud Pricing. https://www.cloudsigma.com/pricing/. Accessed: 2018-07-25.

[8] Danny Dolev, Dror G Feitelson, Joseph Y Halpern, Raz Kupferman, and Nathan Linial. 2012. No justified complaints: On fair sharing of multiple resources. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, ACM, 68–75.

[9] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[10] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. 2016. Ginseng: market-driven LLC allocation. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, ACM, 295–308.

[11] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. 2019. Stochastic Resource Allocation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19)*. USENIX Association, ACM.

[12] Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. 2013. Automated testing of cloud-based elastic systems with AUToCLES. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 714–717.

[13] Google. 2018. Google Cloud Compute Engine Pricing. https://cloud.google.com/compute/pricing. Accessed: 2018-07-25.

[14] Abel Gordon, Michael Hines, Dilma Da Silva, Muli Ben-Yehuda, Marcio Silva, and Gabriel Lizarraga. 2011. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE.

[15] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. 2009. Memory overbooking and dynamic control of Xen virtual machines in consolidated environments. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 630–637. https://doi.org/10.1109/inm.2009.5188871

[16] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. 2013. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (${$ICAC$}$ 13)*. 23–27.

[17] Nikolas Roman Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. 2015. BUNGEE: an elasticity benchmark for self-adaptive IaaS cloud environments. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 46–56.

[18] Michael R. Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Ryu, and Muli Ben-Yehuda. 2011. Applications Know Best: Performance-Driven Memory Overcommit with Ginkgo. In *2011 IEEE 3rd International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 130–137. https://doi.org/10.1109/cloudcom.2011.27

[19] Yazhou Hu, Bo Deng, Yu Yang, and Dongxia Wang. 2016. Elasticity Evaluation of IaaS Cloud Based on Mixed Workloads. In *Proceedings of the 15th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 157–164.

[20] Tatsuo Ishii. 2014. Pgbench PostgreSQL Benchmark. http://www.postgresql.org/docs/9.3/static/pgbench.html. http://www.postgresql.org/docs/9.3/static/pgbench.html

[21] Vipin Kumar. 2002. *Introduction to parallel computing*. Addison-Wesley Longman Publishing Co., Inc.

[22] Adam G. Litke. 2010. Memory overcommitment manager. https://github.com/aglitke/mom. Accessed: 2019-08-07.

[23] Dan Magenheimer et al. 2008. Memory overcommit... without the commitment. *Xen Summit* (2008), 1–3.

[24] Microsoft. 2018. Microsoft Azure AKS B-series Burstable VM. https://azure.microsoft.com/en-us/blog/introducing-b-series-our-new-burstable-vm-size/. Accessed: 2018-07-25.

[25] Danielle Movsowitz, Liran Funaro, Shunit Agmon, Orna Agmon Ben-Yehuda, and Orr Dunkelman. 2018. Why Are Repeated Auctions In RaaS Clouds Risky?. In *Proceedings of the 15th International Conference on Economics of Grids, Clouds, Systems, and Services (GECON '18)*. Springer.

[26] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*. ACM, 237–250. https://doi.org/10.1145/1755913.1755938

[27] Rackspace. 2018. Rackspace Cloud Flavors. https://developer.rackspace.com/docs/cloud-servers/v2/general-api-info/flavors/. Accessed: 2018-09-27.

[28] Tudor I. Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. 2013. Application Level Ballooning for Efficient Server Consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. ACM, 337–350. https://doi.org/10.1145/2465351.2465384

[29] Gauhar Eram Shaikh and Urmila Shrawankar. 2015. Dynamic Memory Allocation Technique for Virtual Machines. In *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, IEEE, 1–6. https://doi.org/10.1109/ICECCT.2015.7226091

[30] Sunil Soman, Chandra Krintz, and David F. Bacon. 2004. Dynamic selection of application-specific garbage collectors. In *4th International Symposium on Memory Management (ISMM)*. ACM, 49–60.

[31] Carl A. Waldspurger. 2002. Memory Resource Management in Vmware ESX Server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, Vol. 36. 181–194.

[32] Andreas Weber, Nikolas Herbst, Henning Groenda, and Samuel Kounev. 2014. Towards a resource elasticity benchmark for cloud environments. In *Proceedings of the 2nd International Workshop on Hot Topics in Cloud service Scalability*. ACM, 5.

[33] R. C. Whaley and J. J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 38–38. https://doi.org/10.1109/SC.1998.10004

[34] Weiming Zhao, Zhenlin Wang, and Yingwei Luo. 2009. Dynamic Memory Balancing for Virtual Machines. *SIGOPS Oper. Syst. Rev.* 43, 3 (2009), 37–47. https://doi.org/10.1145/1618525.1618530

[35] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic tracking of page miss ratio curve for memory management. In *ACM SIGOPS Operating Systems Review*, Vol. 38. ACM, 177–188.