M3: End-to-End Memory Management in Elastic System Software Stacks

David Lion
david.lion@mail.utoronto.ca
University of Toronto

Adrian Chiu
adrian.chiu@mail.utoronto.ca
University of Toronto

Ding Yuan
yuan@ece.utoronto.ca
University of Toronto

Abstract

This paper proposes M3, an end-to-end system that dynamically distributes memory resources among competing applications to maximize their overall performance. Today's data center workloads, can adapt to a wide range of memory sizes, and they are built on complex software stacks.

M3 consists of a set of mechanisms and policies allowing the layers of the system stack to make coordinated decisions. Applications continuously adapt to current resource availability, and resources are distributed to competing applications according to their needs. Experiments show that compared to the best possible static configurations, M3 achieves up to 3.05x speed-up.

1 Introduction

This paper addresses the problem of dynamically distributing memory resources among competing applications, to achieve the best performance. While this is a well-trodden problem [9], the interplay of two characteristics of today's data center workloads creates new challenges.

First, many data center applications are *elastic*. Conventional working set theory predicts that giving an application more memory than its working set does not improve performance, but that it will plummet with less. However, applications like data analytic frameworks or caches can run the same workload with a wide range of memory sizes. Their performance continuously improves with increasing amounts of memory, yet they still make (degraded) progress when provided with less.

This creates the problem of how to best distribute memory among elastic applications, to maximize overall performance. For example, given two jobs A and B, we could distribute the physical memory equally so they complete in, say, 10 seconds, or 70%/30% so they complete in 8 seconds, or 20%/80% at the beginning so that B finishes first; after which A has 100% of the memory, leading to an end-to-end completion time of 5 seconds. All these options result in different completion times, while all having a memory utilization of 100%.

In addition, many of these applications are built on complex software stacks, where each layer performs its own memory management (MM) in isolation from the other layers. For example, data analytics applications typically run on top of a framework, such as Hadoop [16] or Spark [33], which itself runs on a managed language runtime (e.g., Java Virtual Machine (JVM) [15]), all running on top of the OS. Each layer has its own set of mechanisms and policies for MM: The OS abstracts away physical memory, providing processes with nearly infinite virtual memory; the JVM then abstracts away virtual memory, with its own allocation and reclamation policies, i.e., garbage collection (GC); above that Spark automatically partitions the big data input into blocks, and selects a subset to fit into memory.

This further complicates the memory distribution problem. It is fundamentally impossible for applications to optimally distribute physical memory between themselves when physical memory is abstracted away. As a result, language runtimes and analytics frameworks rely on static user configuration. The use of static configuration makes it fundamentally impossible for applications to dynamically adjust the amount of allocated memory, in response to changes in physical memory availability. In addition, multiple layers performing MM in an uncoordinated manner may lead to tunnel-visioned decisions.

Prior work provides different approaches to dynamically distribute memory in a way that outperforms optimal static settings for applications that are both elastic and run on complex software stacks. Apart from MemOpLight [22], only resource deflation [30] and application ballooning [28] address complex software stacks. Resource deflation aims to reduce the memory usage of preemptable virtual machine (VM), and it can further deflate the applications inside the VMs. However, it merely focuses on avoidance of resource exhaustion, and does not address the problem of how to *distribute* the memory resources to maximize performance. Application ballooning

extends the original VM ballooning [36] to multiple application layers. It provides only mechanisms instead of policy for memory distribution. Similar to M3, MemOpLight also distributes policy across the applications, recognizing that applications have the ideal understanding of their performance characteristics. However, in MemOpLight the applications signal the kernel to perform the work, while in M3 the kernel signals the applications to perform the work. MemOpLight also only considers two layers, the kernel and the container.

This paper proposes M3, an end-to-end system that dynamically distributes memory among elastic, stacked applications running on bare-metal kernels.[1] Its goal is to dynamically adapt memory distribution among competing applications, based on their changing needs, to maximize the overall system performance (i.e., throughput). It consists of a set of mechanisms and policies that are inserted into each layer. We implemented M3 in a variety of application stacks, such as Linux (the OS), OpenJDK JVM and Go (language runtimes), Spark, our own key-value memory cache Go-Cache, and Memcached (applications).

M3 is driven by three design principles: an end-to-end approach, adapting dynamically, and non-intrusiveness. According to the end-to-end argument [29], the lowest layer (OS) only notifies the upper layers upon (physical) memory pressure, and leaves the decision of *how*, *when*, and *by how much* memory to allocate and reclaim to the higher level applications. Fundamentally, the application, instead of the OS, is in the best position to make the such decisions [29].

The second principle is to be *adaptive*. The distribution of memory is governed by an *adaptive allocation protocol* that each application on M3 is modified to run. The protocol has two goals. The first is to slow the growth of an application's memory usage when the system is under pressure, when allowing applications to grow freely would lead to thrashing. The second is to distribute available memory resources based on each application's memory demands.

As a result, memory distribution under M3 continuously adapts to changes in application memory demands and system memory availability. The policy used by the OS to decide when to notify the applications is also adaptive. Instead of using a fixed threshold for memory pressure, we propose an algorithm that dynamically adjusts the threshold.

The third design principle is practicality. M3 opts to leverage existing mechanisms, in order to minimize deployment hurdles. This is possible as elastic applications already have mechanisms for memory reclamation in place, such as GC. While applications still require modification to implement the polices, little modification is required for the mechanisms.
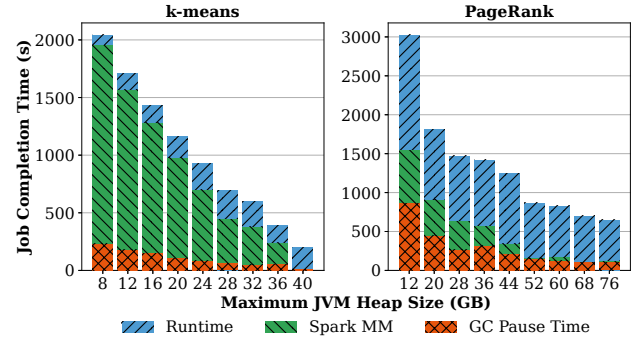

Figure 1. Spark's performance improvement of with increased memory. The experimental set up is the same as §7, except that system memory is allowed to be large enough to fit the entire workload.

We evaluate M3 by thoroughly comparing it with the best static configuration on stock systems under sixteen workloads. With each workload, wetested as many static configurations as possible, and compare M3 with the best one for each workload. We aim to challenge M3, as the configurations require knowledge of future scheduling, require modifying settings that are not recommended to be changed, and are highly sensitive to the benchmark. Yet M3 can still achieve a 1.60x speed-up on average, and 3.05x in the best case, because it adapts to the dynamic changes of a workload's resource needs. We also show that M3 is efficient, as it introduces only an average of 3.77% overhead on workloads that do not benefit from dynamic resource distribution. The source code of all M3 components is publicly available at https://github.com/dsrg-uoft.

M3 has the following limitations. First, not all workloads will benefit from M3. If a static memory distribution will provide the best performance, such workloads will not benefit from M3. In addition, we cannot guarantee that M3's memory distribution is optimal. Finally, M3 is designed for cooperative applications that will follow the policies of M3 and do not maliciously allocate memory, or hold onto memory to starve other applications.

2 Motivation

This section explains the elasticity and complexity of a data center application stack in more detail. It then discusses the motivation of the M3 design.

2.1 Elasticity of Data Analytics Workload

We focus our discussion on data analytics workloads, as the elasticity of cache workloads has been well studied [4, 7, 38].

Figure 1 shows the performance of Spark, running two benchmarks from HiBench [18], as we vary max heap size settings. Job completion time improves over a wide range of heap sizes. For k-means, performance improves with heap size ranging from 8 GB to 40 GB. PageRank's performance also improves over a wide range, between 12 GB and 76 GB.
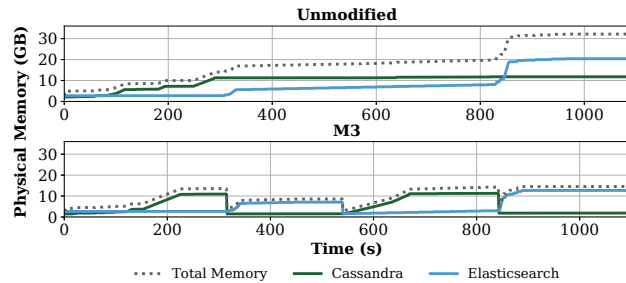
---

[1]M3 stands for Monolithic Memory Management; "3" also indicates that it is capable of handling complex software stack with three or more layers.

Figure 2. A Cassandra server and an Elasticsearch server (running on the JVM) with alternating loads. In an unmodified environment the JVMs climb to their peak and stay at a combined usage of 30 GB. Running on M3 only 15 GB of physical memory is required.

Performance stops improving only beyond 40 GB and 76 GB respectively. This is when the heap can store the entire input. If we use a larger input size, the job completion time will improve over an even wider range.

This elasticity stems from the MM design of both Spark and the JVM. Spark is designed to process data much larger than physical memory. Therefore it partitions input data into blocks, and keeps only a subset in an in-memory cache for processing. At startup, Spark reserves a portion of the JVM max heap size for the block cache. If this cache is full when allocating a new block, Spark will create space for the new block by evicting existing blocks, through a replacement policy.

We measure the time Spark spends in handling capacity misses of its in-memory cache, and show it as bars with back slashes in Figure 1. For the k-means workload, the majority of the completion time is spent on Spark's MM, where smaller heap sizes cause it to constantly swap its cached data.

GC also results in memory elasticity. The max heap size setting embodies a trade-off between completion time and memory usage, which has a wide range of possibilities. A large heap size setting leads to less time spent on GC, whereas a small one leads to longer GC time. This can be seen in the bars with crossing lines in Figure 1, which measure the JVM's GC pause time.[2]

Elasticity of applications complicates the policy for MM. If an application behaves following the working set model, a developer can use a simple all-or-nothing policy [9]: run an application if and only if there is enough RAM for its working set; when more resources become available, begin to run a new application rather than distributing memory to existing ones. Elastic applications bring up a whole new dimension: a developer needs to further consider how to *distribute* memory among applications to achieve the best overall performance.

---

[2]Even when the max heap size is set to be very large, there is still a constant cost of GC. For example, PageRank spends at least 328.62s on GC, regardless of the setting. JVM maintains an internal heap size watermark regardless of the max heap size, and each time the heap usage grows past this watermark it performs GC and increases this watermark.

2.2 MM Issues in the System Stack

The MM issues in complex application stacks stem from three sources: the use of static settings, opaque memory utilization between layers, and uncoordinated MM activities. Let us examine each in detail.

Problem 1: Static Settings. Static settings fundamentally cannot adapt to dynamic changes of physical memory availability, or changes in application demands for memory. When physical memory becomes available, existing applications are unable to allocate this memory. Furthermore, static settings force the operator to overprovision memory for an application's peak memory usage [26, 39]. However, this is at the expense of system utilization.

Figure 2 shows an example of two JVM applications that have alternating memory peaks. The JVM will hold onto its peak memory amount without returning it back to the OS, even when the application only peaks for a short amount of time. This leads to poor *effective* memory utilization. In this workload, the *effective* memory requirement is 15 GB. However, the administrator needs to provision 30 GB of memory, half of which is essentially wasted, and cannot be used by any other application. In comparison, running the same workload on M3 requires half the memory resources in order to achieve the same completion time.

Tuning these static settings also requires tremendous expertise. Numerous articles provide "best practices" that often offer hand-waving or even contradicting information. For example, a blog post on Spark performance tuning [20] suggests to *"avoid using executors with too much memory"* while later suggesting *"avoid using small executors to be able to benefit from running multiple tasks."* Another blog post states that memory overcommitment is a common technique, based on the assumption that not all provisioned memory is needed [39]. However, it also says that this assumption may result in memory exhaustion and processes being killed. In fact, JVM developers have posted numerous bug reports over the years that propose to eliminate the static heap size (all were closed without fix) [8].

Rather than a static heap size, other language runtimes such as Go, Python, Ruby, and CLR control the frequency of GC. In Go, whenever newly allocated data grows by a configured proportion `GOGC`, it will run GC. However, the frequency of GC does not have a clear relationship with physical memory usage. Therefore GC can still be performed unnecessarily when memory is abundant, or be not performed when it should upon memory pressure, leading to thrashing or even being killed. In fact, many bug reports ask for a static max heap size, and Go developers are working on providing it [13, 14].

Problem 2: Opaque Memory Utilization Information. It is difficult for lower layers to understand the amount of memory required by an application, as upper layers that perform MM obfuscate this information. For example, a language runtime such as the JVM will greedily use up its entire max heap size

before aggressively performing GC. To the OS, this memory appears in use, even though the vast majority of it consists of garbage. This leads to an *effective underutilization* of memory resources, which in turn leads to poor performance, as such memory cannot be allocated to other applications. It could further trigger expensive OS swapping, when a much more efficient solution would be for the JVM to perform GC instead. Similarly, the JVM has no knowledge of the block cache in Spark, and must assume for correctness that all memory held by the cache is required. This problem is compounded with multiple layers, as each layer holds onto unused memory.

Problem 3: Uncoordinated MM. Layers performing MM in an isolated and uncoordinated manner will also lead to issues. Being unaware of the MM activities in the upper layers, the lower layer can make unnecessary or non-optimal decisions that hurt performance. For example, if the JVM performs GC before Spark, the upper layer, has released memory, there will less opportunity for the GC cycle to reclaim.

2.3 M3: End-to-End Approach Made Practical

Solving any problem in a layered system faces the question of "which layer should be responsible for what." Prior works have articulated the benefit of an end-to-end principle [12, 21, 29], i.e., resource management decisions should be made by the upper layer applications: "as they know better than operating systems what the goal of their resource management decisions should be and therefore, they should be given as much control as possible over those decisions" [12]. Lampson and Sproul [21] note that general-purpose abstractions unavoidably force applications that do not need them to pay a cost. In fact, these applications use static settings to work around the abstraction provided by the OS. While abstracting away physical memory information is desirable for most applications, it handcuffs applications that do need to make decisions based on physical memory usage.

Unfortunately, an end-to-end design doesn't often go hand-in-hand with practicality, as it typically requires complete changes of the interfaces and breaks the application ecosystem. M3 follows the end-to-end argument, yet it's still practical. We observe that the only decision that requires global information, known only by the OS, is to determine when system memory resources are under pressure. All other decisions are left to the application layers. We further observe that existing systems already have the required mechanisms, without the need of a complete overhaul.[3] For example, the OS already exposes physical memory information and signals; runtimes and applications typically have well-tuned mechanisms to reclaim memory. M3 provides policies that coordinate and control the use of application mechanisms, to balance system memory usage in order to maximize performance.
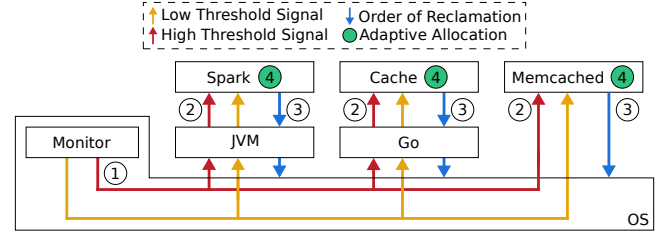

Figure 3. M3's architecture.

3 Overview of M3

Figure 3 shows the M3 architecture. We introduce a monitor that continuously observes the system's physical memory usage. M3 uses two signals, each associated with a threshold that the monitor adjusts dynamically. If memory usage grows past these thresholds, the monitor sends signals to processes so that they can return memory. The *low* threshold signal (yellow arrow in Figure 3) is an early warning, sent when system memory begins to become scarce. If memory pressure continues to increase and the *high* threshold is reached, M3 sends a high threshold signal (red arrow in Figure 3) to selected processes.

We use multiple thresholds instead of one to allow flexibility. An application needs to consider the trade-off among three factors when deciding how to reclaim memory: speed of reclamation, amount of memory reclaimed, and future performance impact. For example, generational GC can perform a young collection, which garbage-collects only newly allocated objects and hence completes quickly, or a full collection, which is slower, but returns much more memory, as it scans the entire heap [2]. Spark or a memory cache can evict blocks/items to return even more memory, at the expense of future performance. Having two signals allows applications to trade-off different reclamation algorithms.

The signals are propagated to each layer that registered to handle them; each layer decides for itself whether and how it should handle each signal. For example, when the JVM receives a signal, it forwards it to Spark first, and performs GC after Spark evicted data from its cache. This maximizes the effectiveness of GC, as there is now a large portion of garbage memory to reclaim. The downward arrows in Figure 3 shows the order of memory reclamation.

The numbers in Figure 3 show the order of events that take place when the system experiences memory pressure. First, the monitor sends signals to processes registered with M3. The lowest application layer then forwards the signal to upper levels (step 2). The upper layers perform reclamation before the lower level (step 3). Afterwards, applications slow down their memory growth, using M3's adaptive allocation protocol, until no more signals have been received for some period of time.

Target workloads. In general, workloads will benefit from M3 when a static setting is not optimal for overall system

---

[3]We refer to the functionality provided by applications to coordinate memory management across layers as mechanisms.

Layer	App.	Policy			Mechanisms	
		Low Signal	High Signal	Adaptive Allocation	Return Memory	API to Upper Layer
Upper	Spark	Call down to JVM	Evict blocks + call JVM	✓	Call JVM	Not necessary
	Go-Cache	Light eviction + call Go	Heavy eviction + call Go	✓	Call Go	(Item delete)
	Memcached	Light eviction	Heavy eviction	✓	Use jemalloc	(Item delete)
Lower	JVM	Young GC	Mixed GC	Not necessary	`madvise`	Fine grain GC API
	Go	GC	GC	Not necessary	`madvise`	Fine grain GC API

Table 1. Policies and mechanisms designed for various applications that we modified/implemented. Parentheses around "Item delete" indicates that the API already exists.

performance. Specifically, they should have the following three characteristics:

- *Multitenancy:* memory is shared among *multiple* applications running at the same time.
- *Large peak usage:* the sum of the peak of memory usages is greater than the total amount of physical memory. Otherwise one could statically allocate each application with its peak memory size.
- *Changing needs:* the memory needs of each application change dynamically and independently of each other. If their needs don't change, e.g., each application has a constant working set size, there is no opportunity for adaptation when M3 dynamically redistributes the memory. If their memory needs always change in-sync, then a static partition will be ideal.

In practice, applications running on language runtimes and caches are ideal targets. These applications are typically elastic, hence by definition they will have a large peak usage, as their performance would continue to improve with more memory.

4 Application Policy and Mechanism

This section describes the mechanisms and policies designed and implemented in M3. Mechanisms refer to technical tools that perform a specific functionality used by M3. A policy describes how and when such mechanism are used. Table 1 summarizes the mechanisms and policies we designed and implemented, for four real-world applications and our self-made Go-Cache. Note that the number of layers can vary; for instance Memcached has no "Lower" runtime layer. It is also possible for applications to benefit from M3 without every layer needing modification. For example, in Figure 2 from §2.2, Cassandra and ElasticSearch leverage M3 without being modified, by leveraging the modifications already made to the underlying JVM.

4.1 Mechanisms

In M3 a MM layer must be capable of returning memory to the layer below it. For example, a runtime must return memory to the OS. If this mechanism does not already exist, the layer should implement it to be integrated into M3. In practice, we

found that applications rarely return memory to the OS, but can be modified to do so, using existing mechanisms, e.g., `madvise` to return memory freed from the application to the OS.

For example, the JVM rarely usually holds onto memory freed by GC. Therefore, we modify the JVM to return memory to the OS by using the `madvise` system call whenever a heap region is freed. Similarly, Go returns free regions to the OS, if they have not been used for 5 minutes. As this is not sufficient to respond to memory pressure, we modify Go to return heap regions to the OS with `madvise` as soon as they are collected. It is the policies of M3 that control when to reclaim memory, and therefore the use of a mechanism. Similarly, Memcached uses `malloc/free` by default, which does not return free memory back to the OS, therefore we replaced it with `jemalloc`, which does.

The granularity of memory reclamation is an issue. For example, a typical memory cache's eviction policy is at the granularity of individual items. However, memory can be returned to the OS only with page granularity. Therefore, for Go-Cache and Memcached, we evict an entire slab of key-value pairs to ensure we have contiguous memory to return to the OS.

M3 also requires that a layer to notify the layer below it, when it has finished reclamation. This allows memory reclamation to be done in the correct order. For an application to completely benefit from M3, all of its layers should participate in memory reclamation. If a lower layer, such as the JVM, is integrated into M3, an unmodified application above it still benefits as the JVM will still return memory to the OS. However, the benefit will be limited, if the application itself does not reclaim memory when necessary. In practice, such mechanisms often already exist. The JVM already expose APIs for applications to trigger GC, but only at the granularity of a full GC. We added another API to trigger less expensive, incremental GC, to be used on a low threshold signal. Similarly, memory caches (Go-Cache and Memcached) already provide APIs for the application to delete items explicitly.

4.2 Policies and the Adaptive Allocation Protocol

An application policy involves three parts: (1) handling low-threshold signals, (2) handling high-threshold signals, and (3)

adaptively controlling allocation when the system is under pressure (i.e., upon receiving a high-threshold signal). Signal handling is relatively straightforward. On a low threshold signal, the application performs light-weight reclamation, prioritizing speed over the quantity of memory reclaimed. On a high threshold signal, heavy-weight reclamation prioritizes the quantity of memory reclaimed, to avoid exhaustion.

For instance, when the Spark stack receives a low-threshold signal, it does *not* evict from its cache: it merely calls the JVM to perform a fast young collection. It handles a high threshold signal differently, by evicting blocks from its cache (at the expense of increased cache misses). It then calls down to the JVM, collecting both young and old regions in a "mixed GC," collecting more memory than a young GC. Similar policies are used for Go-cache and Memcached, as shown in Table 1.

We design an adaptive allocation protocol to slow down applications' memory allocation after a high-threshold signal is received. This is needed because for most applications, allocating memory is faster than reclaiming memory, so the rate of allocation can outpace concurrent reclamation. Hence, when a high-threshold signal is received, indicating that the system is under memory pressure, we need to slow down applications' allocation to avoid memory exhaustion.

The protocol works by adaptively calculating the *rate of allowed allocation*. The value of this rate is within the range of $[0, 100\%]$. For example, if this rate is 10%, then only 10% of the allocations are allowed, with the other allocations being delayed. When the system is not under pressure, the rate is 100%, meaning every memory allocation is allowed. Right after the application receives a high-threshold signal, this rate drops to nearly zero, meaning almost all allocations are delayed. The rate then gradually increases as time elapses.

This rate of allowed allocation is determined as:

$$\text{Allow rate} = \min\left(\frac{\text{Time elapsed since last signal}}{\text{Epoch length} \times NUM_{epochs}}, 100\%\right)$$

The rate depends on three parameters: (1) time elapsed since receiving the last high-threshold signal, (2) the length of the *epoch*, which is the time spent handling the last high-threshold signal, and (3) the number of *epochs* (NUM_{epochs}). The first 2 parameters are automatically computed by M3, whereas NUM_{epochs} is a static user configuration. Next, we discuss each parameter in turn:

1. The rate increases linearly with the time elapsed since the last high-threshold signal was received, until it reaches 100%. When a new signal is received, this rate is reset to 0.[4]

2. An epoch is defined as starting from when the application receives the signal, to when memory has been returned. Using the epoch length as the denominator rewards applications that reclaim memory fast: the faster

---

[4]We experimented with other strategies, such as exponential growth instead of linear, and found that this protocol is the most effective.

an application can reclaim memory, the faster it is allowed to grow.

3. NUM_{epochs} controls how long the allocation control should be applied after each signal: the allow rate reaches 100% after NUM_{epochs} epochs of time elapses. Hence, the larger NUM_{epochs} is, the longer allocation control is applied. We set this value to 1 in Spark in our experiment, and 5 for the Go-Cache and Memcached. This is because Spark stack takes longer to reclaim memory, therefore a small NUM_{epochs} value 1 is sufficient and achieves the best performance.

We implement this protocol by modifying the allocation function(s), generically referred to as `alloc()`, used by the application. Only the $\lfloor \frac{1}{r} \rfloor$-th `alloc()` is allowed to proceed as normal, where r is the allow rate. For example, if the allow rate is 10%, then only the 10th allocation will be allowed, with other allocations delayed. We recalculate the allow rate every time the allocation function is invoked.

When an allocation is delayed, we first invoke the memory eviction function(s) inside `alloc()`, to free space that is enough to satisfy this allocation. After eviction completes, `alloc()` will proceed as normal. This eviction is performed by the same thread that invoked `alloc()`. As a result, delayed allocation does *not* affect correctness, as it never fails an allocation; the thread that invokes a delayed `alloc()` will only experience a longer return time. In fact, delayed allocation already exists in unmodified elastic applications: when memory usage reaches the static maximum size, they will first perform eviction until enough space is created to service the allocation, such that usage does not increase past maximum size.

Note that shrinking is performed by signal handlers and the adaptive protocol does *not* shrink an application's memory usage; instead, it only prevents it from expanding (on those allocations that get delayed). The evicted memory is *not* returned to the OS; instead it is *replaced* with the newly allocated data.

Adaptive allocation is relevant only for the top-most memory management layer (Spark, Go-Cache, and Memcached), as that is where memory allocations originate. For example, in a Spark stack, the allocations come from Spark itself, and if allocations are already delayed at the Spark layer, it's not necessary to delay them further in the JVM. Additionally, this is also the layer with the best domain knowledge. For example, the Go-Cache can decide between rejecting a request to store key-value pair entirely or servicing the request after first evicting other items.

With every application stack running this adaptive protocol, M3 achieves a near-optimal overall memory distribution in a decentralized manner. An application with low signal handling overhead is rewarded with faster growth, as it will have a higher allow rate. If the memory growth causes signals by crossing the thresholds, the application has shown

it will handle these signals promptly and reclaim memory well before exhaustion. Thus overall system performance is improved with no consequence.

The protocol also adapts to the application's need for resources. Two applications with similar signal handling time will have similar allow rates. However, the application experiencing higher memory demand will have more `alloc()` calls. This leads to more `alloc()` calls without delay, that grow the application's memory as they do not perform eviction. Thus, the application with higher memory demand sees more memory growth.

We run the adaptive protocol on high-threshold signals only. The reasons are twofold: (1) the protocol can significantly impact application performance, so it is unnecessary when the resource usage is not in the "red zone", and (2) it is very effective in curbing growth, so it's safe to only run it on high-threshold signals.

4.3 Guidelines for Porting Applications

Porting other applications will involve similar policies and mechanisms as in Table 1. We speculate that the most complicated task would be to design and add reclamation algorithms, such as GC, to an application that does not already have them. As explained in §3, memory reclamation needs to trade speed for amount reclaimed and performance impact.

In our experience, an effective approach is to handle the signal quickly, even if it returns only a small amount of memory. There are multiple advantages. First, handling a signal quickly reduces memory pressure immediately, preventing exhaustion. The system operates at higher utilization, because M3 will adaptively increase the thresholds if memory pressure is relieved (see §5). Returning a small amount of memory will reduce negative performance impact. This leads to a more agile system, where applications are capable of handling many signals and incrementally return memory. This memory can be quickly redistributed among applications, based on their changing needs at fine granularity. In fact, we find modern GC algorithms are also moving toward this direction. For example, HotSpot JVM's implements the Garbage-first (G1) [10] GC algorithm which performs quick, incremental cycles that only collect a subset of memory, as opposed to the traditional stop-the-world full-heap GC cycle.

In practice, we found using the existing memory reclamation utilities in applications to be sufficient. The only problematic case was Memcached's use of `malloc`. However, we found `jemalloc` to be a suitable drop-in replacement, and believe it can be used for other applications using `malloc`.

Applications based on managed runtime environments pose different challenges. A runtime provides its own memory management system, such as in the JVM or Go, and will reclaim application memory. However, this memory must still be freed by the runtime in order to be returned to the OS. This, in turn, requires that the runtime supports M3. This is a one-time cost, across all applications. On the other
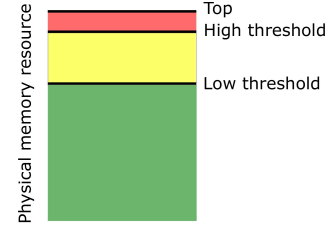

Figure 4. The high and low memory usage thresholds.

Algorithm 1: Sending the high threshold signal.

1   **Array** *reg_procs* ← all registered processes;
2   Sort *reg_procs*;
3   **Array** *proc_group* ← ∅;
4   **Int** *expected* ← 0;
5   **Int** *target* ← *memory_used* − *high_threshold*;
6   **if** *target* <= *0* **then**
7   **return**;
8   **end**
9   **for** *p* in *reg_procs* **do until** *expected* >= *target*
10   Signal *p*;
11   *expected* ← *expected* + (*p*'s expected reclamation amount);
12   **end**
13   **return**;

---

hand, we found that porting the managed applications to be simpler compared to unmanaged ones. As the runtime handles memory management, the application logic does not contain deallocation code.

5 Design of the M3 Monitor

The primary job of the monitor is to alert processes of scarce physical memory. It uses two dynamically adjusted thresholds, namely the high and low threshold. It polls system memory, and when it sees memory past a threshold, it sends a signal it.

Figure 4 shows the two thresholds. The monitor is configured with a value for the *top* of memory, the acceptable amount that can be used by applications running on M3. Typically, this is set at or just below total physical memory. The two thresholds partition it into three zones: green when memory usage is under the low threshold, yellow between the low and high thresholds, and red past the high threshold.

5.1 Selective Notification

The monitor should signal only select processes for the high threshold signal, in order to minimize the overhead of signal handling. The goal of the selection algorithm (shown in Algorithm 1) is to drop the system below the high threshold. It runs once per polling period, while the usage is above the high threshold.

The algorithm sorts the processes, based on a configurable sorting order. The higher ranked processes will be signaled. M3 currently supports four sorting orders: newest to oldest (favors batch jobs), oldest to newest (favors interactive jobs), memory usage (largest to smallest), or expected reclamation amount (largest to smallest), computed as the average reclamation of this process over the last five signals. We use newest to oldest in our evaluation.

The loop from line 9 to 12 selects the processes to be signaled. It goes through the sorted processes and signals each one, until the sum of the expected reclamation amount is larger than $target$.

Note that if memory consumption continues to rise above the user configured *top* of memory, the monitor will signal *all* processes with the high threshold signal in hopes of reclaiming all possible memory. After a configured amount of time, if system memory is still above the top of memory, processes must be killed. M3 uses Algorithm 1 again to select processes to kill, until system memory drops below the top.

5.2 Adaptive Thresholds

Statically configured thresholds are undesirable. If the thresholds are set low, applications that reclaim memory quickly are forced to do so before there is any danger of exhausting memory, leading to resource under-utilization. Conversely, if the thresholds are set too high, applications may not finish reclamation before the system exceeds the top of memory, exhausting memory and causing swapping. Furthermore, there is no optimal static setting when there is a variety of workloads on the system. For example, say there are two applications, *A* and *B*, where *A* reclaims memory slowly and *B* reclaims memory quickly. These applications have different ideal threshold settings, as *A* requires low thresholds to ensure enough time to reclaim memory, while *B* would benefit from high thresholds due to its quick reclamation allowing it to avoid the overhead of handling unnecessary signals. If these applications are run one after the other, it is not possible to statically configure the ideal thresholds for both applications. Therefore, M3 updates the thresholds dynamically.

The low threshold should temper how often system memory reaches the *high* threshold. Intuitively, if the high threshold is reached more often than expected, the low threshold should be lowered by the monitor. On the other hand, if the high threshold is reached less often than expected, the low threshold should increase. Let T_{red} be the time spent above the high threshold and $T_{\text{green or yellow}}$ be the time spend below the high threshold. These times are calculated over a configurable sliding window. If the ratio of $T_{\text{red}} : T_{\text{green or yellow}}$ is larger than a configured target, the low signal threshold is lowered. If this ratio is less than the target, the low signal threshold is increased by the monitor.

The low threshold is updated only under certain conditions to avoid over fitting. It is lowered only if the system is above the high threshold (in addition to the ratio being above the

target). If system memory has already dropped below the high threshold, the means that memory pressure is already relieved; hence there is no reason to further lower the low threshold. Similarly, if the system is already below the low threshold, there is no reason to raise the low threshold because no signals are sent anyway. Additionally, the low threshold cannot be greater than the high threshold.

Adapting the high threshold is similar to the low threshold, only that it aims to achieve the ratio of time spent above and below the *top*. The logic on when to raise and lower the threshold based on a ratio is exactly the same, but now based on the top value. While constantly operating above the top of memory is undesirable, if system memory never reaches top, there is opportunity to increase memory utilization by raising the high threshold. If system memory spends too much time above the top, applications are already continuously receiving signals. Lowering the high threshold will not change the number of signals sent or amount of memory applications reclaim. However, it will ensure that applications receive signals sooner, and begin reclaiming memory earlier.

Note that M3 does not adjust thresholds when the system is operating in the green or yellow zone. Operating in the yellow zone is not problematic, as long as the system is below the high threshold.

6 Implementation

We implement a monitor for M3, and modify the JVM, Spark, the Go runtime, and Memcached to function under M3. We use the HotSpot JVM in OpenJDK 1.8 build 25.71, Spark v2.3.2, Go v1.11rc1, and Memcached v1.6.7. We also built Go-Cache, a cache that is capable of expanding and returning memory to the OS. Next we describe our implementation details and parameters.

The monitor. We implement the monitor as a user-space process on Linux. Processes register by creating PID files in a known directory. The monitor polls `MemAvailable` in `/proc/meminfo` once every second. The two signals are real-time signal numbers, provided by Linux for application-defined purposes. The monitor is implemented in approximately 600 lines of C++ code.

In the evaluation system, the top of memory is set to 62 GB (out of 64 GB available), while the low and high thresholds are initialized to 50 GB and 55 GB respectively and are adjusted dynamically. Both the low and high threshold ratios are set to 1:32. These ratios are calculated over a configurable sliding window, defaulting to the past 32 polled memory values. Each time the thresholds are moved, they are adjusted by a configurable 2% of the top of memory.

JVM modifications. We implement M3 on HotSpot's Garbage-first (G1) GC algorithm [10], the default starting in OpenJDK 9 [27]. In order to implement the threshold signal handling and expose API to upper layers we modified approximately 200 lines of C++ code and 20 lines of Java code in the JVM.

To run Elasticsearch for Figure 2 in §2.2, we also ported the changes to OpenJDK 12.

Spark modifications. Recall that Spark sizes its block cache based on JVM's max heap size setting. We modified Spark so that the block cache is set to a very large size. Therefore Spark will continue to add blocks until it is limited by M3's signals. Every time a high threshold signal is received, Spark evicts $\frac{1}{8}$ of the blocks with an LRU policy. We added approximately 250 lines of Scala and Java code for Spark.

Go modifications. It took approximately 50 lines to implement the changes described in §4.

Go-Cache. Go-Cache is implemented as a library that applications can import, a practice widely adopted by industry (for example, both Google's LevelDB and Facebook's Cache-Lib [4] are designed as library). On a high threshold signal, 4% of slabs are evicted with a LRU policy, while the low threshold signal evicts 1%. The library and benchmark contain approximately 300 and 200 lines of code respectively.

Memcached implementation. Similarly to Go-Cache, Memcached also uses slabs. We modified Memcached to evict the same proportion of slabs on signals, in approximately 170 lines of code.

7 Evaluation

We run the following experiments on M3 evaluating its performance and adaptability: (1) the speedup compared to the unmodified systems that are *optimally configured*; (2) the worst-case overhead; (3) M3's ability to utilize memory more effectively; (4) the effectiveness of M3's dynamic thresholds.

7.1 Methodology

All tests are performed on an in-house cluster with 8 worker servers and 1 management server. Each server has a Xeon E5-2630V3, 16 virtual core, 2.4GHz CPU. Each server has 64 GB of available memory set by a Linux control group. We chose 64 GB to mirror the memory to virtual core ratio on Amazon EC2 M5 general purpose servers [3]. Each server has one 7,200 RPM hard drives, is connected via 10Gbps interconnect, and runs Linux 4.15.0.

We setup a Spark cluster in standalone mode on top of an HDFS cluster (v2.8.5) used to store input data. The 8 worker servers each run a Spark Worker and an HDFS Datanode. For each job, Spark spawns a single multi-threaded JVM process, known as an executor, on each node scheduled to run the job. The Go-Cache benchmark consists of a single process using the Go-Cache library on each node. We give each job 5 cores per node as most of our workloads consist of 3 concurrent jobs.

7.1.1 Workloads.
Our tests are run using the HiBench [18] benchmarking suite, which consists of 15 Spark tests. Three of them are used in this study: PageRank, n-weight, and k-means. We chose these three benchmarks as they benefit from having more memory. PageRank, n-weight, and k-means have 5.7 GB, 1.8 GB, and 89.8 GB of input data on HDFS.

The Go-Cache benchmark first preloads Go-Cache with 85% of a key space of 12 million keys. This is done to simulate an in-use system, and avoid cold-misses which inflate the runtime. It then performs 6.5 million uniform random get requests over the key space. If the key is not present, the goroutine sleeps for 1 millisecond to simulate a backend lookup, then performs a put. Although, it is not common practice to run a dedicated caching server alongside analytics jobs, it is common for various applications to have their own internal cache through a library [4]. As we run a benchmark that imports the Go-Cache library, it is not an ideal real world application. However, an application importing the Go-Cache library running alongside other applications is a realistic scenario that we try to replicate.

We made a best effort to cover a comprehensive set of workloads, totaling to 16 workloads. A workload consists of multiple applications, each scheduled with varying amounts of delay. We run each workload and configuration pair 5 times.

We ensured to evaluate M3's theoretical worst cases. M3 has no opportunity to improve workloads during periods where there is no available system memory and memory is partitioned optimally, or when all applications have sufficient memory to run optimally. Workloads consisting of identical applications, with no delay, guarantee that there is no possibility for improvement. The optimal resource distribution is known (equal partitioning), and at any point in time the applications' memory demands are identical. Of the workloads, four consist of identical applications with no delay.

7.1.2 Configuration settings.
M3 is compared against four different settings using unmodified applications. The *Default* setting uses default configurations of each application. It replicates a naive setting. The *Globally Optimal*, *Oracle*, and *Oracle With Spark configuration (OWS)* settings challenge M3 against increasingly unrealistic degrees of configuration tuning.

In the *Default* setting, the default JVM heap size is 16 GB on a system with 64 GB of memory. The default GOGC environment variable is 100, meaning GC will be performed every time the heap grows by 100%. For Go-Cache we set the cache size to 16 GB, mimicking the JVM. We also leave all Spark configurations at their defaults.

The *Globally Optimal* setting assigns each application a single set of configurations that minimizes the *average runtime of all 16 workloads*. In other words, it is optimized for these 16 workloads as a whole. This requires extensive application profiling and configuration tuning, as well as knowledge of all possible workloads. Foreknowledge of all possible workloads is practically impossible, while extensive configuration tuning is extremely time consuming. The two Spark parameters we test are `memory.fraction`, and
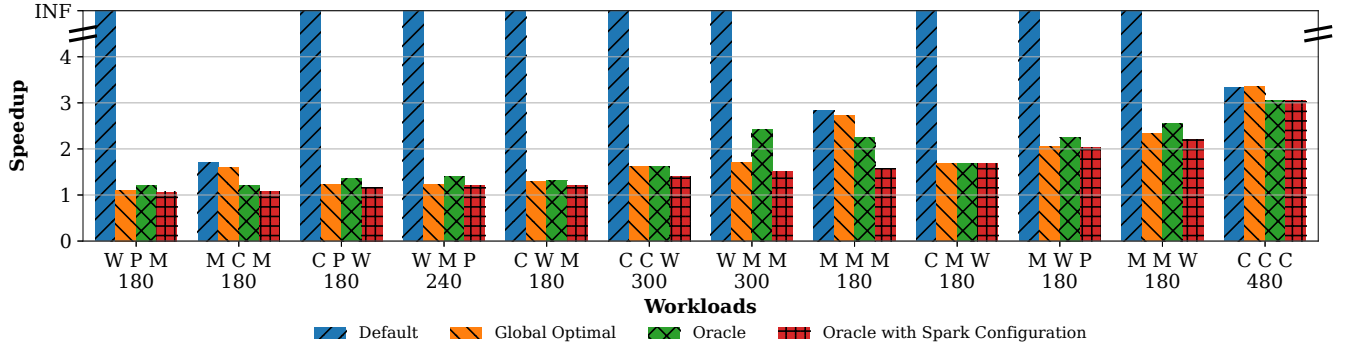
Figure 5. Runtime performance of various workloads, comparing M3 with the 4 unmodified settings. Applications W, P, C, and M refer to n-weight, PageRank, Go-Cache, and k-means respectively. The number below is the delay between each job in seconds. We report the average speedup of all jobs in the workload compared to M3. The workloads are sorted by M3's speedup (or slow-down) compared to the *oracle with Spark configuration* setting.

`memory.storageFraction`. Note that Spark recommends leaving these values at their defaults, as changing them can have unexpected affects on performance [32].

The *Oracle* setting uses the best found static memory partitioning for *each individual workload* to maximizes its performance. This differs from the *Globally Optimal* setting as now the configurations differ in each workload. It is completely impossible to implement in practice as it requires future knowledge of the specific scheduling and combination of applications to be run. We modify the JVM heap size and the `GOGC` parameter. The *OWS* setting further adds tuning to the Spark configuration to the *Oracle* setting. This represents the ultimate challenge for M3 as all configuration settings of each application are tuned specifically for the particular workload and scheduling being run.

We did a thorough search for the optimal configurations in each of the last three settings trying combinations of the four parameters: JVM heap size, the two Spark parameters, and `GOGC`. In theory, the search space is infinite, because each of the four parameters can have infinite values. We use our domain knowledge to limit the search space to only likely candidates. Over the course of four months, we performed more than 3400 tests over sixteen workloads. We selected the configuration combinations that result in the best performance for each setting.

7.2 Speedup

Figure 5 compares M3 to the four environments over the twelve workloads, out of the sixteen, that are not the theoretical worst case for M3. We report the average speedup of each application's running time within a workload. Specifically, we first calculate the speedup of each of the three application within a workload. Then we take the average of these three speedups, and further average them across the five runs. Reporting the speedup of each application is more meaningful than comparing the end-to-end completion time of all three applications because (1) the end-to-end completion time can

be significantly affected by the scheduling delay we added between two applications, and (2) it can also be dominated by a single long-running application (hiding the effect to other applications).

M3 achieves an average of 1.60x speedup over the OWS setting among the twelve workloads, with a best case speedup of 3.05x. Recall that this setting represents the best possible configuration for each workload.

Compared to the Oracle setting, M3 achieves an average speedup 1.86x, and 3.05x in the best case. Compared to the Globally Optimal setting, M3's average speedup is 1.83x with a 3.35x best case. The Global Optimal setting has better performance than the oracle setting, because it additionally uses the optimal Spark settings.

With the Default settings, nine of the twelve workloads cannot even run. For instance, n-weight cannot complete with the default heap size. Compared to the workloads that did finish, M3 has 2.62x speedup.

M3 speeds up performance, because even under the best static setting, a workload does not always perfectly utilize memory. With static settings, applications must be partitioned such that the combined peak memory usage fits in system memory. If this peak is not maintained for the entire workload, then memory will be underutilized, allowing M3 to perform better. Additionally, the globally optimal environment requires certain workloads to be underutilized, as memory must be partitioned according to other workloads.

7.2.1 Understanding M3's Speedup. Figure 6 shows workload MMW 180, one of the workloads with the best speedup, under M3 and under the OWS setting.[5] The memory profile is taken from one node of our tests in Figure 5. The workload consists of two k-means jobs followed by an n-weight job, each with a 3 minute delay. Under M3, the two k-means are

---

[5]Note that in Figures 6, 7 and 10, the benchmarks finish earlier under M3. This confirms that M3 makes better utilization of the memory resources.
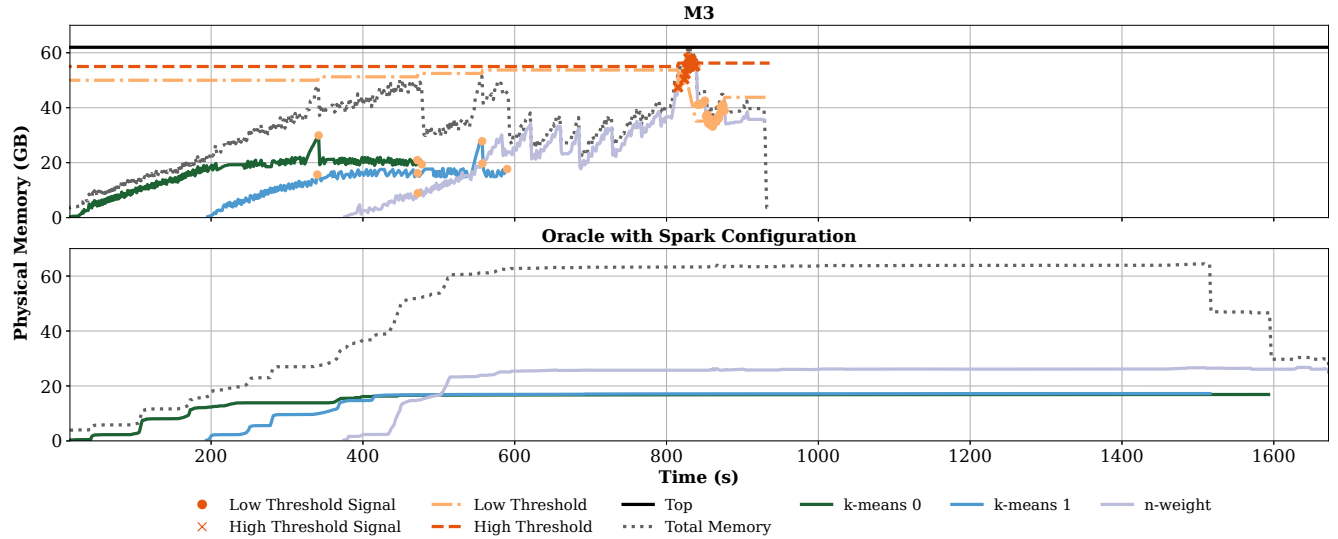
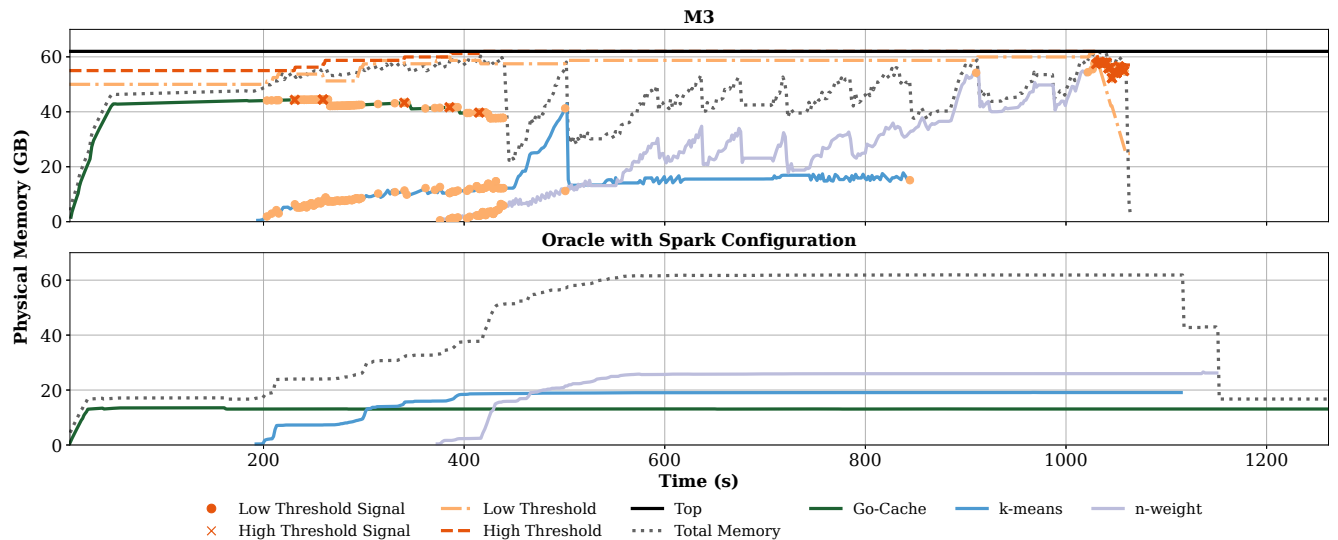**Figure 6.** Memory profile from the MMW 180 workload. Top: M3. Bottom: OWS.

Figure 7. Memory profile taken from the CMW 180 workload. Top: M3. Bottom: OWS.

able to quickly utilize more memory. For them, we measured that Spark is able to cache 55% more blocks under M3.

M3 can accommodate for spikes in memory which do not overlap. Both k-means jobs peak at around 30 GB, but the best static setting in the unmodified systems must provision for the combined peaks, allowing only a physical memory usage of 16 GB each. Unmodified Spark must aggressively evict blocks from its block cache, spending over three times more than M3.

Under M3, n-weight is able to use much more memory after the k-means applications finish, and thus avoid GC. The JVM spends approximately 90 seconds in stop-the-world GC under M3, compared to 200 seconds in the unmodified systems. Finally, in an unmodified system, there is a compounding

effect, as the jobs overlap and additionally suffer from disk contention. M3, applications overlap less, as they finish faster, minimizing disk contention.

Figure 6 also shows the effectiveness of the monitor's threshold adjustment. Both the low and high thresholds gradually increase at the beginning, as the system operates under the high threshold. Starting around the 815 second mark, usage repeatedly reaches the high threshold, causing the low threshold to drop. However, the high threshold keeps increasing, as the system still operates underneath the top of memory.

Figure 6 further shows the effectiveness of application policies. First, applications reclaiming memory is effective in relieving pressure. Even the low threshold signals (e.g.,
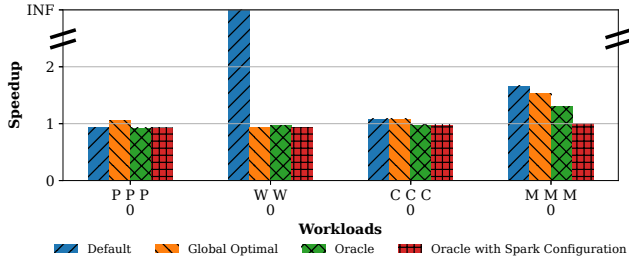
Figure 8. Runtime performance of the four theoretical worst-case workloads where each workload consists of identical applications started with no delay.

Figure 9. Runtime performance of a workload consisting of a Spark k-means job followed after a 4 minute delay by a Memcached benchmark. The left bar shows the runtime under M3, while the right bar shows an unmodified version.

the ones that occur at the 341 and 557 second marks) effectively reclaim significant amount of memory. In addition, even though usage creeps up to the red zone, the adaptive allocation protocol is effective in slowing down allocation to keep operating underneath the top, as shown by the continuous rise of the high threshold.

Figure 7 graphs M3 and the OWS setting for the CMW 180 workload. It shows that M3 partitions memory according to the applications' needs. The k-means job uses less memory than the Go-Cache job as it has lower demand. After Go-Cache finishes at 434 second mark, k-means quickly consumes a lot of memory, as memory pressure has decreased and signals stop. Later in the benchmark, n-weights acquires more memory than k-means, as it has a higher demand.

Note that the peaks of the three jobs are at 44.48 GB, 42.83 GB, and 58.15 GB respectively. Their sum is well above system memory, at 145.46 GB. Static partitioning would not allow this, but under M3 these jobs run without issues, as their peaks do not coincide. Under M3 all three jobs finish faster than under the OWS setting.

Overhead of M3 in the Worst Case. Figure 8 shows M3's performance in the worst-case scenarios for M3. The workloads start identical jobs all at the same time, creating a scenario where using static configuration should provide the best performance. M3 achieves an average slow-down of 3.77%, compared to the OWS setting, with a worst-case slow-down of 7.00%. This is because in an unmodified environment, a similar amount of GC and eviction would need to be performed to keep the heap size below the static limit, yet M3 adds overhead in its handling signals.

M3 can still beat the Oracle setting, where equal memory partitioning is optimal, as seen in the workload MMM 0. With the default Spark parameters, Spark will not use more than 60% of the heap for storage space. Therefore, the remaining 40% of the heap is under-utilized, as Spark has limited its memory usage based on static settings.

Memcached. To show that M3 is fully capable of working with a wide variety of applications, including native ones, we include a workload using Memcached. To evaluate Memcached with M3, we run a workload consisting of a Spark k-means job followed after a 4 minute delay by a
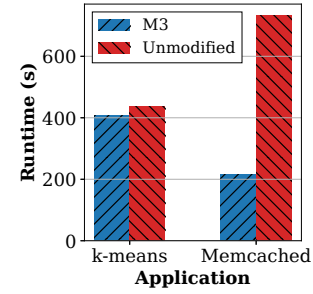
memtier [19] Memcached benchmark, shown in Figure 9. We didn't have enough time to evaluate Memcached with a variety of workloads. Similarly, we were unable to comprehensively cover many static settings and used a best effort approach. The Spark job and the Memcached server run on a single node with 8 GB of memory. The Spark Master and memtier benchmark run on a separate server. M3 achieved an average application speedup of 2.23.

7.3 Optimizing Effective Memory Utilization

M3 can improve *effective* memory utilization. Applications often hold onto memory that they do not require, e.g., the JVM rarely returns freed memory to the OS. Hence, the utilization observed by the OS can be misleadingly larger than the effective utilization.

This can be observed in Figure 6 and 7, where in unmodified systems the total memory used observed by the OS (grey dotted line) is close to RAM size (64 GB) most of the time, whereas it's much lower in M3. Specifically, the average memory utilization, measured in RSS, is 63 GB and 54 GB for unmodified systems for Figures 6 and 7 respectively, whereas it is 38 GB and 48 GB for M3. Hence, the unmodified workloads waste 25 GB and 6 GB respectively. The memory saved by M3 can allow *additional applications* to run, increasing both effective utilization and system throughput. Figure 2 in §2.2 shows a similar issue.

7.4 Dynamic Threshold Analysis

Figure 10 compares dynamic high and low thresholds with static ones. Both the static low threshold and the dynamic initial low threshold are set to 40 GB. The static high threshold and the dynamic initial high threshold are set to 45 GB. M3 detects that the applications are able to return memory, and raises both the thresholds. Because of the improved memory usage, the workload with dynamic thresholds terminates 1.93x earlier than the static one.
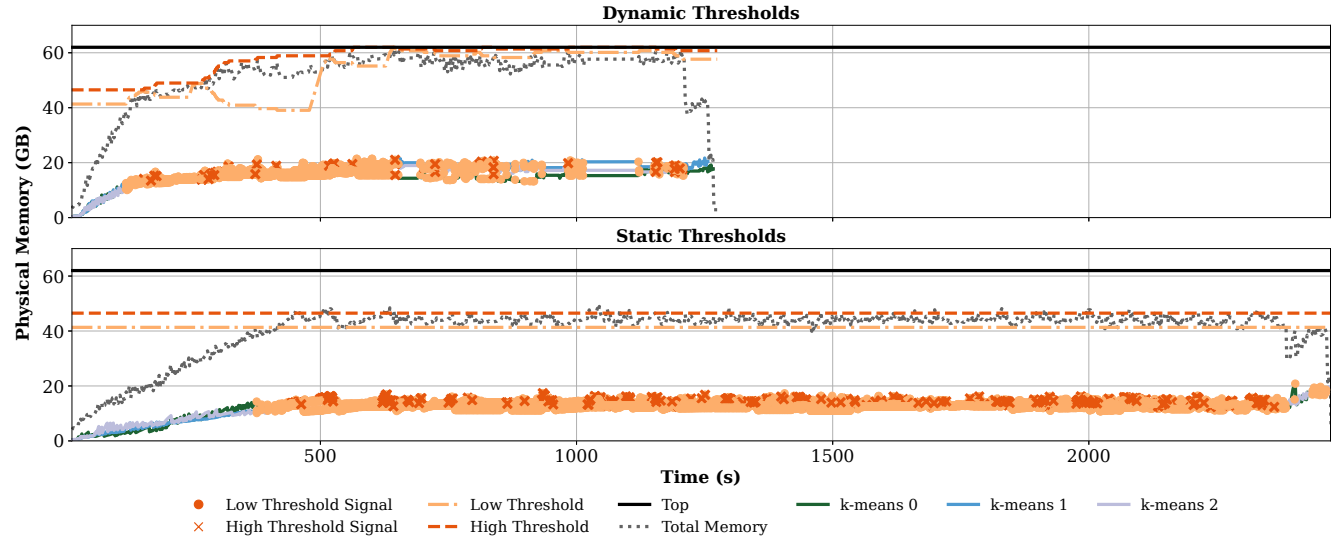
Figure 10. Memory profiles of M3 running a workload of three k-means applications with no delay. Top: dynamic signal thresholds. Bottom: static thresholds.

8 Related Work

M3 provides a unique approach to balancing memory among elastic, stacked applications, in order to maximize their collective performance. In M3 application allocation tends towards a global optimum thanks to the cooperation of policies implemented in each application. By propagating reclaimed memory down through each memory management layer to the OS, M3 leverages each layers domain specific knowledge for fine grained memory collection, while also avoiding requiring modifications to the kernel.

MM across multiple application layers. Sharma *et al.* propose "resource deflation" to deflate preemptable virtual machines under memory pressure [30]. The deflation controller notifies both the VM and the applications inside the VM to evict memory. This approach is implemented with the Spark stack and with Memcached. M3 has the following differences. First, the goal of resource deflation is to prevent VMs from being killed, instead of dynamically distributing resources for performance. When resources become available, resource deflation prioritizes adding in more VMs, instead of increasing the performance of existing ones. It does not have policy to control the memory growth of applications, whereas M3's adaptive allocation protocol curbs memory growth. In addition,VM resizing occurs only when a regular VM is created, whereas M3 redistributes memory in real-time. Finally, in Sharma *et al.*, Spark releases memory by killing a task, whereas in M3 we signal it to evict blocks. Killing a task won't be helpful, as Spark will restart the same task, which will reuse the memory just freed. Since M3 keeps all nodes in the cluster at a high resource utilization, restarting the task on a different node will encounter the same problem.

Laniel *et al.* present MemOpLight, which has similar goals to M3 but a different approach [22]. MemOpLight recognize that, as Linux relies on static memory limits to isolate containers, it cannot adapt to dynamic application behavior. Therefore, similar to M3, MemOpLight leaves policy decisions to applications, which have a much better understanding of their performance characteristics. In both MemOpLight and M3 the distributed policy across applications coordinates their efforts towards global optimal memory utilization. In contrast to M3, in MemOpLight it is the applications that signal the kernel, which then deallocates physical pages. On the other hand, in M3 the applications perform the deallocation themselves after being signaled by the kernel, allowing for finer grained memory control. Unlike M3, MemOpLight only targets two layers, the kernel and the container.

Other works provide mechanisms, instead of policies, to enable cross-layer MM. Waldspurger's ballooning, enables VMs to dynamically expand and shrink memory usage, through a device driver mechanism [36]. Salomie *et al.* extend VM ballooning [36] to multiple layers in the application stack inside of the VM, but defers to future work [28]. M3 could use these mechanisms to integrate with the VMs and hypervisor. However, this would introduce additional policy challenges, as memory must be balanced across the applications inside a guest OS, and across the VMs. If only a single application were to run inside a VM, then the policy to manage VMs on a hypervisor would be very similar to managing applications in an OS.

Windows [25], Linux [11], and Facebook's oomd project [39] propose mechanisms to notify applications upon memory pressure. Microsoft's Common Language Runtime provides APIs to expose this notification to the application [24]. Similarly, Oracle provides Cooperative Memory Management

in their elastic cloud environment, where a system memory monitor informs registered applications about memory pressure levels. These works are strictly mechanisms and leave all policy decisions to the developers.

Memory coordination. Other works dynamically resize the heap size of applications, but cannot be applied to complex application stacks, where multiple layers perform MM. They either rely on working sets [9], miss ratio curves (MRCs) [7, 31, 37, 38, 43], or the GC frequency in JVM [6].

Some proposals dynamically resize the JVM heap size, based on system memory availability. Appel *et al.* design a garbage collector that resizes the JVM's heap based on suggestions from an advisor [1]. CRAMM builds on this work by designing a JVM garbage collector that calls down to the OS to resize its heap based on the working set size [40]. These works consider only GC and completely ignore MM by upper layers. They do not control the memory growth when there is memory pressure. Therefore, elastic applications cannot leverage resizing based on their semantics. In addition, in these works the policy decision is made by the lowest layer (advisor or OS), instead of upper layers as in M3. Finally, CRAMM requires a completely new virtual memory system to collect detailed page reference information.

Taurus [23] synchronizes GC pauses across different JVMs running the same Spark job. Taurus does not deal with distributing memory or complex multi-layer stacks, but solve a different problem, that of coordinating multiple JVMs.

Cluster management frameworks. Existing cluster management frameworks, such as Borg [35], Mesos [17], YARN [34], Hurricane [5], and Spark [41, 42], are able to dynamically schedule workloads at the granularity of containers (or executors in Spark's case). However, scheduling relies on a statically configured memory size.

Borg [35] allows resource overcommitment by scheduling more containers than the total size of physical memory. However, it does not provide a mechanism to notify containers to shrink, and it resorts to killing jobs when physical resources run out.

9 Concluding Remarks

Many applications today are elastic, and built on complex software stack consisting of the OS, language runtimes, and cache-like applications. Each layer abstracts physical memory resources away to ease development. An unfortunate consequence for performance is that memory resources are statically distributed based on user settings, such as the max heap size in the JVM. These settings are hard to tune; and the static nature means that fundamentally, they cannot react to workload or utilization changes.

We developed M3, which bridges memory abstractions between layers. When there is no memory pressure, applications can freely allocate memory and expand. When the system falls under memory pressure, signals are propagated

to the upper layers of applications. Upon receiving signals, the upper layers reclaim memory and adaptively control their memory allocation, with different layers operating in a coordinated manner to continuously adapt to the current resource availability. Compared to optimally configured stock systems, M3 achieves up to 3.05x speed-up, while having an average speedup of 1.60x.

This paper also leaves some questions for future work. M3 does not guarantee that the memory distribution is optimal (defined by maximum system throughput). Ideally, we could measure the optimal memory distribution for each workload used in our evaluation and compare it with M3. However, searching for the optimal distribution is challenging, given the complex nature of today's systems software. Therefore, it is unclear whether M3 achieves optimality, and if not, how far away it is.

In addition, we would also like to understand M3's effectiveness on a broader set of applications. Are there other class of applications that are also elastic and hence can benefit on M3? Do they exhibit different characteristics that require redesign of M3? Also, can M3 be extended to virtual machines or containers? These are the questions we hope to answer in future work.

Acknowledgements

References

[1] Raphael Alonso and Andrew W. Appel. 1990. An Advisor for Flexible Working Sets. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Univ. of Colorado, Boulder, Colorado, USA) *(SIGMETRICS '90)*. ACM, New York, NY, USA, 153–162. https://doi.org/10.1145/98457.98753

[2] A. W. Appel. 1989. Simple Generational Garbage Collection and Fast Allocation. *Softw. Pract. Exper.* 19, 2 (Feb. 1989), 171–183. https://doi.org/10.1002/spe.4380190206

[3] Amazon Web Services (AWS). 2021. EC2 Instance Pricing - Amazon Web Services (AWS). https://aws.amazon.com/ec2/pricing/on-demand/.

[4] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation* (Virtual Event) *(OSDI '20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[5] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 20, 15 pages. https://doi.org/10.1145/3190508.3190532

[6] Rodrigo Bruno, Paulo Ferreira, Ruslan Synytsky, Tetiana Fydorenchyk, Jia Rao, Hang Huang, and Song Wu. 2018. Dynamic Vertical Memory Scalability for OpenJDK Cloud Applications. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management* (Philadelphia, PA, USA) *(ISMM '18)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/3210563.3210567

[7] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2018. mPart: Miss-ratio Curve Guided Partitioning in Key-value Stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management* (Philadelphia, PA, USA) *(ISMM '18)*. ACM, New York, NY, USA, 84–95. https://doi.org/10.1145/3210563.3210571

[8] Oracle Java Bug Database. 2014. JDK-4408373 : Can we eliminate the -Xmx max heap "glass ceiling"? https://bugs.java.com/view_bug.do?bug_id=4408373.

[9] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* 11, 5 (May 1968), 323–333. https://doi.org/10.1145/363095.363141

[10] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

[11] Jake Edge. 2008. Avoiding the OOM killer with mem_notify. https://lwn.net/Articles/267013/.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[13] Go Language GitHub. 2015. Issue 9849: runtime: make max heap size configurable. https://github.com/golang/go/issues/9849.

[14] Go Language GitHub. 2016. Issue 16843: runtime: mechanism for monitoring heap size. https://github.com/golang/go/issues/16843.

[15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. The Java®Virtual Machine Specification - Java SE 8 Edition. https://docs.oracle.com/javase/specs/jvms/se8/html/.

[16] Apache Hadoop. 2021. The Apache™ Hadoop® Project . https://hadoop.apache.org.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '11)*. USENIX Association, Berkeley, CA, USA, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488

[18] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. 2010. The Hi-Bench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)* (Long Beach, CA, USA). 41–51. https://doi.org/10.1109/ICDEW.2010.5452747

[19] Redis Labs. 2021. GitHub – RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark.

[20] Zero Gravity Labs. 2017. Spark Performance Tuning: A Checklist. https://medium.com/zero-gravity-labs/spark-performance-tuning-a-checklist-abb3c80efb44.

[21] Butler W. Lampson and Robert F. Sproull. 1979. An Open Operating System for a Single-user Machine. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (Pacific Grove, CA, USA) *(SOSP '79)*. ACM, 98–105. https://doi.org/10.1145/800215.806575

[22] F. Laniel, D. Carver, J. Sopena, F. Wajsburt, J. Lejeune, and M. Shapiro. 2020. MemOpLight: Leveraging application feedback to improve container memory consolidation. In *19th IEEE International Symposium on Network Computing and Applications (NCA)* (Cambridge, MA, USA). 1–10. https://doi.org/10.1109/NCA51143.2020.9306717

[23] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 457–471. https://doi.org/10.1145/2872362.2872386

[24] Microsoft. 2017. Microsoft Docs – .NET – ICLRMemoryNotificationCallback Interface. https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/hosting/iclrmemorynotificationcallback-interface.

[25] Microsoft. 2018. Microsoft Docs – Win32 – CreateMemoryResourceNotification. https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-creatememoryresourcenotification.

[26] Chakri Padala. 2017. Time for memory disaggregation? https://www.ericsson.com/en/blog/2017/5/time-for-memory-disaggregation.

[27] OpenJDK JDK Enhancement Proposals. 2017. JEP 248: Make G1 the Default Garbage Collector. http://openjdk.java.net/jeps/248.

[28] Tudor-Ioan 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* (Prague, Czech Republic) *(EuroSys '13)*. ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/2465351.2465384

[29] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (Nov. 1984), 277–288. https://doi.org/10.1145/357401.357402

[30] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. 2019. Resource Deflation: A New Approach For Transient Resource Reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 33, 17 pages. https://doi.org/10.1145/3302424.3303945

[31] Alan J. Smith. 1983. *Disk Cache – Miss Ratio Analysis and Design Considerations*. Technical Report UCB/CSD-83-120. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1983/6336.html

[32] Apache Spark. 2020. Configuration – Spark 2.4.1 Documentation. https://spark.apache.org/docs/2.4.1/configuration.html#memory-management.

[33] Apache Spark. 2021. Apache Spark™– Unified Analytics Engine for Big Data. http://spark.apache.org.

[34] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[35] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. ACM, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[36] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading* (Boston, MA, USA) *(OSDI '02)*. USENIX Association, Berkeley, CA, USA, 181–194. http://dl.acm.org/citation.cfm?id=1060289.1060307

[37] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST '15)*. USENIX Association, USA, 95–110.

[38] Carl A. Waldspurger, Trausti Saemundson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization Using Miniature Simulations. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 487–498.

[39] Daniel Xu. 2018. Open-sourcing oomd, a new approach to handling OOMs. https://code.fb.com/production-engineering/oomd/.

[40] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2006. CRAMM: Virtual Memory Support for Garbage-collected Applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington, USA) *(OSDI*

'06)*. USENIX Association, Berkeley, CA, USA, 103–116. http://dl.acm.org/citation.cfm?id=1298455.1298466

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA, USA) *(NSDI '12)*. USENIX Association, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[42] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA, USA) *(HotCloud '10)*. USENIX Association, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

[43] 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 *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, USA) *(ASPLOS XI)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/1024393.1024415