Title
Understanding Memory Configurations for In-Memory Analytics

Author
Reiss, Charles Albert

Publication Date
2016

Understanding Memory Configurations for In-Memory Analytics

by

Charles Albert Reiss

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Randy H. Katz, Chair
Professor Ion Stoica
Professor Rhonda Righter

Summer 2016

Abstract

Understanding Memory Configurations for In-Memory Analytics

by

Charles Albert Reiss

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy H. Katz, Chair

A proliferation of frameworks have emerged to handle the challenges of making distributed computations reliable and scalable. These enable users to easily perform analysis of large datasets on commodity clusters. As users have demanded better response times for these computations, newer versions of these frameworks have focused on efficiently keeping computation in memory. A major challenge in deploying such frameworks is understanding application memory requirements. Just as the layers of abstraction of frameworks assist in writing efficient and robust applications, they hide the true memory requirements.

In this dissertation, I describe and evaluate SLAMR, a tool I have developed for providing users with memory recommendations for programs written for the Apache Spark analytics stack. These recommendations practically address the lack of visibility users have into memory requirements even as they are asked to assign resources to deploy their analytics programs. My tool records activity from an example execution, both from the framework and the garbage collector of its underlying language runtime. Given this instrumentation, it estimates a memory configuration that effectively keeps the entire computation in memory, such that allocating more memory would have minimal benefit on performance. Because in-memory analytics frameworks are built to take advantage of the memory available to them, simply observing actual memory usage is not an effective way to produce such estimates. A challenge, then, is to produce these estimates without performing effort similar to trying configurations around the ultimate recommendation.

SLAMR provides these recommendations without requiring many example executions or detailed analysis of the semantics of the program. What I instrument allows it to predict the effect of different memory configurations rather than simply reflecting the available memory. It collects information corresponding to the abstractions provided by frameworks, so it can distinguish which memory usage is useful and account for when alternate storage was used. To understand the requirements of the underlying language runtime in this analytics stack, it also collects statistics about the program execution that can replayed into a dramatically simplified model of a garbage collector. Both of these models are built around the goal of providing a *conservative* bound, allowing users to use the resulting memory recommendations in confidence rather than inflate them to avoid the risk of memory exhaustion from errors in the estimates. I show through evaluation that SLAMR provides effective, consistently safe recommendations for a variety of analytics programs and does so with minimal measurement overhead.

Contents

Chapter 1

Introduction

Large-scale data processing once required specialized hardware and months of dedicated engineering effort. Now it is an everyday feature in business and research, made possible by widely available software tools and affordable public computing resources. Built around the design of colocating computation with data, a proliferation of 'frameworks' handle the challenges of making distributed computations reliable and scalable, letting users focus instead on specifying the computations to perform.

Originally developed for the very large data processing problems encountered by large Internet companies, these frameworks have expanded well beyond this niche. Users with all sorts of data too large to process comfortably on their personal machines use distributed data processing tools. Libraries built on top of such frameworks enable not only simple statistics collection, searching, and index building, but complex calculations, such as constructing models for recommendation systems. Thus, these frameworks have enabled activities previously only performed by the most sophisticated enterprises to be a commonplace occurrence.

Along with new capabilities comes new complexities. When analytics tools work imperfectly, users have trouble interpreting what is happening. After all, the purpose of data processing frameworks and libraries is to shield users from the details of the distributed computation. This separation, however, can be problematic for managing resource usage and performance where the abstractions are often incomplete. To solve these problems, it matters what all the purportedly hidden layers are doing. Ordinarily performance issues are not correctness problems, but resource allocation problems frequently are.

In this dissertation, I address a particularly important resource allocation problem: choosing memory allocations. This decision is often delegated to users by design. Users supply the framework some computing resources and need to decide how much to give it. As more users pay for computing resources piecemeal rather than having a purpose-built cluster to themselves, this is a decision they must make frequently. To help these users, I developed and evaluated a tool, SLAMR (Safe Lightweight Analytics Memory Recommender), that integrates with an Apache Spark analytics stack to assist users with their decisions. My tool peeks beneath the abstractions that make it difficult to infer resource requirements from both the high-level interfaces of analytics frameworks and by observing which resources those frameworks use.

I examine this problem of memory allocations in a context where it is particularly important: data analysis frameworks that focus on keeping data in memory. The advantage of keeping data in memory is better response times and throughput, especially for emerging interactive and iterative machine learning workloads. If sufficient memory is not available, users will not experience adequate performance benefits.

1.1 Analytic Framework Interfaces

Analytics frameworks focus on specifying transformations of data. In contrast to traditional supercomputing systems based on interfaces like MPI [56], users largely specify what data to produce rather than where to perform computations and how to move data between them. Frameworks handle those details. They provide an array of useful transformations and, often, the ability to create more by composing existing transformations. In implementing these transformations, a certain kind of flexibility is the goal— supporting arbitrary computations on arbitrary amounts of data — if one has enough machines with which to 'scale out' the computation.

This kind of interface was popularized by Google's MapReduce [25], the predecessor to the modern wave of systems which enable distributed computations with fine-grained fault-tolerance. MapReduce started by providing its namesake transformation — applying a map function to produce key-value pairs, then calling a reduce function with the values for each key. The MapReduce framework handled the details of scaling out the computation, and allowing user-supplied functions ensured that this interface was flexible. Successors to MapReduce provide a wider, more convenient suite of transformations and are less dependent on storing intermediate data to disks, though many of the core implementation strategies remain the same.

To automatically distribute computations, these MapReduce-derived frameworks track how data is partitioned across a cluster. To execute transformations, the framework generally runs the computation where the data already is. Whenever data needs to be moved, the framework coordinates that movement. By taking this responsibility, the framework can handle the load balancing and fine-grained fault tolerance necessary to work with reasonable efficiency on clusters of non-specialized hardware.

This automation ought to mean that once a user writes a program using one of these frameworks, it will just work. As is always the case in complex systems, this has often not been true in practice, especially when the concern is not just completion but reasonably efficient completion.

1.2 Deploying Analytics

There are a variety of ways for users to obtain computational resources. For example, they could have a cluster built for their applications. But, considerably more common is sharing that cluster with other users — other divisions of a company or members of the general public.

Such environments are not new; supercomputing installations have long had such sharing. But modern environments are more flexible in how resources are shared and what options they give users, with the cluster shared at a finer granularity than individual 'compute nodes'. Using them efficiently is more important as users are often billed (actually billed or paying 'funny money' within a company) in a fine-grained manner for what they request. Even if they are not, cluster operators want to ensure their cluster is used efficiently.

Along with more options for how users run programs, the increased ease of use of analytics frameworks has compressed the development cycle. In addition, analytics frameworks themselves have focused on increasing interactivity by decreasing deployment, response, and programming times, making them more suitable for exploratory analyses. Figuring out what resources to request is no longer hidden as one of many tasks in performing distributed data analysis. As less sophisticated users increasingly perform these large computations, determining how to run them efficiently has become more of a puzzle.

Analytics frameworks, despite their automation of distributing and load balancing computations, generally have not assisted users in finding a good configuration of resources for the computation. This is partly because this flexibility is rather new. Historically, most large-scale deployments of analytics were tied to where data was persistently stored. In the most important, highly-scaled use cases, they depended primarily on the speed of this I/O — the performance of some distributed filesystem and whether they would be able to colocate with that data.

For these applications, obtaining reasonable allocations of resources was mostly a matter of accommodating the temporary data of disk-to-disk tasks and selecting a sensible degree of parallelism. But newer frameworks have found benefits to keeping data in local, non-persistent storage — usually DRAM. For applications that do many passes over data, including many enabled by constantly improving support for complex and interactive analyses, this happily decouples them from the speed of some distributed storage systems and especially from the speed of spinning disks. To actually obtain these advantages, a user must request enough memory to store their working set.

Practically, users need to decide their resource request before they commence their computation. When data lives in a distributed filesystem, adjusting where computation takes place based on the available computing resources may be easy. This occurs, for example, with some cluster scheduling techniques for MapReduce-like workloads.But when a working set needs to be distributed across the machines assigned to the user, adjustments require moving that data.

1.3 User Behavior and Resource Exhaustion

So, how do users make resource requests to deploy analytics? To answer this, I examined a trace from a mixed workload on a Google cluster [73]. Comparing memory requests with memory usage revealed that users are largely wasteful, seeming to reserve twice the space required. This arises even when there should be incentive to make corrections given the high demand for and likely cost of the underlying compute resources. If these users make such errors, how could one expect users who are still developing their analysis and who have more modest resources to do better?

The user experience of resource exhaustion — especially memory exhaustion — suggests why. Because of the abstractions, users lack good intuition about how their program is actually executed. Still, they see effects of resource exhaustion: programs crash or run very slowly. In the event of a crash, a user at least understands the cause is resource exhaustion. But, especially with fine-grained fault tolerance, it may take a while for the error to be produced and to determine that its cause was allocating too few resources and not a bug in the user's code.

Further complicating users' understanding is that there is no single "memory capacity" to configure. The memory configuration is complex, involving many parameters. These include setting aside space for many different purposes — such as for keeping persistent data, storing values before they are stored persistently, performing aggregations, assigning space for garbage collector operations, etc. In addition to the difficulty of understanding the framework internals that drive the demands for these different allocations, users are charged with understanding how these settings should change as they change the number of workers (often another configuration parameter) offered to increase parallelism or decrease cost.

Frameworks will often diligently make best use of the resources they have configured. With less memory than the true working set size (for some task), the framework may complete computations by performing extra I/O, serialization, and deserialization. The result is that users may suffer from the effects of memory exhaustion and not know it — their program will run well enough, and they will not know it could be better. The framework will maintain its abstraction, fulfilling its promise that the user does not need to worry about how the computation is actually performed, but only so long as the user does not care sufficiently about the performance or cost of the computation.

1.4 Insights from Instrumentation

The same framework abstractions that separate users from their memory exhaustion problems offer a solution. Users already rely on the framework to distribute their large amounts of data and to be responsible for their memory usage. To provide this capability, the framework does much of the work necessary to consolidate information about memory requirements in tracking how data and computations are distributed.

I take advantage of this to produce a practical model of framework memory requirements, for a particular framework, Apache Spark. To this framework, I add the appropriate instrumentation to report the memory usage information that is readily available to it. Since the framework already needs to track data and often data sizes as part of its normal operation, this instrumentation only required minor changes to the framework. The complexity of these changes was further limited by relying on abstractions within Apache Spark itself rather than the more varied interfaces provided to end users. Based on the result of instrumentation, I can compute a "good" configuration based on a program's execution.

This configuration is not simply something greater than the high water mark of memory usage. Integrating my recommendation tool with the framework gives it access to information about data that is not kept in memory — about those decisions the framework makes to more slowly compute a result instead of giving up and failing due to lack of memory. Integrating with the framework also makes it possible to determine whether data stored in memory is actually used. Integrating with the framework makes it easy to separate what memory usage is part of fixed per-worker cost that will change if the number of workers allocated is changed.

1.5 Managed Runtimes for System Software

A memory configuration for an analytics framework is not the full story, as the framework is only a piece of the puzzle. Apache Spark — like a majority of open source systems software written in the past decade — is written in a managed, garbage-collected language. This is of great convenience to Spark's developers, at least avoiding reasoning about memory leaks and perhaps making memory management faster than if Spark had implemented it itself. It is also of great convenience to Spark's users — they can write their functions in a managed language and have Spark run them efficiently, without awkward and often performance-degrading 'gates' between languages.

But these language's runtimes have their own memory configuration problem. If one simply considers the amount of space required for the active objects of a garbage collected program and sets that as the maximum memory size, the program will crash every time. Garbage collection achieves reasonable performance by collecting garbage (unreachable objects) in large batches, meaning that there must be space for garbage to accumulate. And modern garbage collectors do not simply just have one big region of memory they scan; such simple techniques achieve very poor performance. Instead, modern garbage collectors split up their memory to achieve reasonable performance. The choices about how to do so are themselves more memory configuration parameters which impact a program's overall memory foot print.

The resources required for a garbage collector depends greatly on the application. Programs which are very 'friendly' to the garbage collector, for example, by managing a few very large allocations, may require essentially nothing beyond what one would assume from framework information. For other programs, the garbage collector may require substantial extra space to avoid continuously scanning memory for expired objects. Complicating this assessment, most garbage collectors rely on some heuristics to achieve good performance, whose effectiveness will vary between programs.

I address this variability by measuring program behavior to determine its garbage collection needs. I do this specifically in the context of in-memory analytics, using framework memory requirements to alleviate concerns about possible rare program behaviors. Using observations of program activity, I estimate the program's overall performance across a range of garbage collector configurations. Rather than focus on maximizing performance, I estimate an upper-bound on the garbage collector's compute-time overhead. Using this, I can choose a configuration where I can be confident the overhead will not be excessive.

1.6 Roadmap

The combination of framework and garbage collector configuration gives a complete recommendation for memory allocation for in-memory analytics. In this dissertation, I describe and evaluate SLAMR, a tool I have developed to provide these recommendations by observing the behavior of a user program.

I start by reviewing the history of analytics systems in Chapter 2, including the evolution from disk-based analytics systems to in-memory systems. I review, in particular, some of the evolution of probably the most successful in-memory analytics system, Spark. Then, in preparation for devising tools addressing analytics frameworks, I review the long history of configuration recommendation tools targeting a wide variety of systems with a wide variety of goals.

Given this review, I proceed to analyze the context for memory configuration recommendation in Chapter 3 by means of an example of an environment from Google. In this I show the loss of efficiency experienced in this large environment due to improper memory configurations. I also find evidence that cluster-scheduler approaches based around overcommitting are not very good solutions. The penalty when a user program fails because of insufficient memory is large. To compensate, users are likely to make very conservative requests and cluster schedulers are likely to avoid overcommitting.

I provide an overview of a solution to this challenge in Chapter 4. My solution is built around modeling memory usage requirements for frameworks and underlying language runtimes. In contrast to most prior 'autotuning' systems, the design of SLAMR and the recommendations it produces are focused on providing fast, consistent recommendations to users. These ensure that recommendations are practically deployable by users. I also describe what form memory requirements must take to be usable by users — configuring multiple components and assisting in decisions about the degree of parallelism to use. In the next two chapters, I describe these models.

In Chapter 5, I describe the part of SLAMR that models memory usage in an analytics framework, in particular, Apache Spark. SLAMR uses observations of user requirements from the examination of the cluster workload. This model is built on a simplifying heuristic

— allocating enough memory for everything eligible in the framework to be kept in memory. Based on adding a small amount of additional instrumentation, I record a trace of data accesses from a representative execution. I transform this record into a concrete configuration for Apache Spark for a given level of parallelism. I evaluate these recommendations on a variety of example programs.

Recommending the amount of space for the analytics framework is insufficient without also accounting for overheads from its runtime environment, such as from garbage collectors it uses. In Chapter 6, I address these garbage collection overheads, a notorious source of surprising failures. As with framework requirements, I focus on being conservative, in this case by estimating an upper bound of the actual overhead. This bound is based on a simplified model of a typical throughput-oriented garbage collector. I collect information about program behavior from garbage collector counters, making small changes to the effective garbage collector configurations to collect sufficient information from a single execution. Given this, I show that my approach produces useful estimates of maximum overheads for a variety of analytics programs.

In Chapter 7, I describe how to combine these models to produce recommendations in an appropriate form. The individual models by themselves, producing estimated overheads and minimum memory for a given degree of parallelism, do not immediately result in a configuration recommendation. To meet the requirements of users articulated in Chapter 4, I matched viable configurations against models of user 'utility' that mirror typical charging for computing resources. These utility estimates are biased towards smaller configurations, thereby producing a 'minimum' configuration. I show that, despite needing to consider tradeoffs between allocated memory and estimated computation time, suggested configurations are not very sensitive to computation time estimates because of the steep increase in overhead as allocated space decreases.

Finally, I discuss some future directions in Chapter 8. I describe how the instrumentation available in analytics frameworks was lacking for evaluating memory requirements. These are largely problems of visibility between layers that future framework authors should consider as they design their systems. I also describe how both my framework and garbage collector models could be extended to meet other memory recommendation goals. Rather than focus on minimizing instrumentation overhead and purely conservative recommendations, one can make more nuanced memory/time tradeoffs and choose from a wider range of configurations, generally at the cost of increased measurement overheads.

Chapter 2

Related and Prior Work

The increased availability of large datasets and the computing power to process them have created increasing demand for large-scale data processing. In response, researchers have developed tools like MapReduce and its many derivatives to assist this growing user base. These tools include abstractions that handle details of fault tolerance, distributing computation, and sharing computing resources. Such abstractions have a long history, dating to scientific computing workloads, where machines were purpose-built and rare, and the workloads were more bound by computational power than data size.

Contemporary data processing problems and computing environments are different. Fast development and data exploration are now more common goals, and the distinction between consumer and high-end computing hardware has narrowed. Beginning with the rise of MapReduce [25], modern computing frameworks have thus focused on utilizing commodity-like machines with increased automation to hide the additional complexity this requires.

Originally, MapReduce-based systems, in their pursuit of scale, focused on computing with data that would only fit on spinning disks. But, as users have seen value in distributed computation frameworks for less huge datasets and the decreased price of DRAM has created demand for in-main-memory computation. Thus frameworks like Spark are built around memory, providing better performance and interactivity.

MapReduce-like systems make many ‘invisible’ choices about how to distribute computations, how to perform communications and joins, and how to allocate resources between multiple uses. These choices are generally reflected in a proliferation of configuration parameters, which sophisticated users often find affect performance dramatically. This proliferation of settings is not new; even problems with less moving parts, like performing fast linear algebra, have similarly complex parameter spaces. Thus, many solutions have been proposed for automatically exploring these parameters, modeling the effects of changing these parameters, or creating control loops for adjusting them.

In this dissertation, I particularly focus my analysis on one class of parameters: memory allocation sizes. Prior work on adjusting the allocation of memory, which I will review later in this chapter, has largely focused on long-running interactive services where there is an easily observed, roughly continuous relationship between memory allocation. I will examine this problem in the context of in-memory analytics frameworks, where such changes in performance are less gradual and measured by the execution of a relatively long pipeline, not a short request. Consequently, my approach is more similar to non-control-loop based performance modeling techniques than much prior memory tuning work.

2.1 Roadmap

In this chapter, I review both the evolution of analytics systems and efforts to better tune their configurations for performance and resource usage. I also examine efforts to tune configuration and resource usage in other types of systems, usually to improve performance. I begin by reviewing the history of analytics systems, starting with MapReduce, the system that started the modern emergence of data-intensive computing. I describe how these systems lead to in-memory distributed computing frameworks, notably Apache Spark, which I use as my platform in the rest of this dissertation. I then look specifically at memory management in these frameworks. I start by reviewing how Apache Spark's approach to this problem has changed, evolving to increased instrumentation and explicit memory management.

My goal in understanding memory usage in this thesis is to help users set configurations related to memory management. Accordingly, I review prior work on configuration tuning in general. Most of this work focuses on optimizing performance (usually latency), but similar techniques have been used to manage memory footprints. I start by reviewing the largely offline, 'black box' autotuning approaches, followed by application-specific extensions of this that integrate abstract models of how the system operates. I pay special attention to prior work that has modeled caches, usually in the context of computer architecture, as I will apply this to modeling Apache Spark's caches.

After looking at offline configuration tuning techniques, I review online tuning techniques, which generally use a control loop approach to find the optimal configuration. I do not explore these techniques in depth, instead focusing on approaches which makes less changes to the application environment and which depend less on applications being in a 'steady state'. However, these systems are particularly interesting because they have been used for adjusting resource allocations similar to those I want to configure. Typically, these have targeted for long-running services with variable workload, where an offline approach simply would not make sense.

After reviewing more generic configuration management approaches, I look at work on tuning garbage collectors or changing how they work. In a system like Apache Spark — and many modern tools that would be part of an 'analytics stack' — these are ubiquitous, and their performance and memory overhead matters. Most related to my work in Chapter 6, I examine control-loop-based approaches that garbage collectors use to tune their configuration parameters at runtime. These models together with prior work on modeling cache performance will most inform the configuration models I develop in this dissertation.

2.2 Analytics Frameworks

Developing tools to enable users to easily analyze has been a major area of research and commercial development [99, 79, 51, 5, 6, 84, 63]. The increased prevalence of a large natural datasets, such as from the Internet, aided by the increased ease of data collection, sharing, and storage, has driven demand to scale-out data processing. Unlike the distributed computations of scientific simulations that have driven traditional supercomputers, the dominant tasks in these new data processing workloads are focused around moving data rather than raw computation. Simultaneously, a shift from custom-hardware supercomputers to more cost effective mass-produced hardware has reduced barriers to scalable computing but created increased complexity in software and more focus on scale over efficiency, to make use of these less consistent systems with less performant networks. To handle these complexities, data analytics *frameworks* arose, combinations libraries and runtimes to support writing distributed data analysis programs.

2.2.1 Disk-Based Analytics: MapReduce and Hadoop

The most high impact framework in this transition is MapReduce[25] and its (sometimes more general) reimplementations Hadoop[7] and Dryad [43]. Combined with distributed filesystems that colocated storage on compute nodes, MapReduce moved computation to data. In contrast to traditional database designs, it emphasized in-query fault tolerance and load balancing. Rather than relying on checkpoints or restarting transactions, MapReduce automatically recovers from worker failures with little extra work.

MapReduce's design enables out-of-core computation, forcing users to structure computations in a way that can effectively use large sequential I/Os. The original use cases for MapReduce — 'web scale' datasets — had no hope of fitting on fast random access storage. The MapReduce abstraction let the MapReduce framework focus on efficiently organizing the I/O to this to storage.

While MapReduce proved a valuable execution engine, users generally did not find it sufficiently pleasant to use as a programming model. This lead to a proliferation of abstractions that provided a higher-level programming language for pipelines of MapReduce operations. These included both abstractions that provided an imperative interface [63, 19, 22] with fairly direct translation to MapReduce operations and implementations of SQL-like declarative languages [84, 14]. These abstractions provide libraries of common MapReduce operations, like maximum-within-group and sorting, that are straightforward but tedious to express directly against the MapReduce execution engine, and manage schemas and intermediate storage.

2.2.2 In-Memory Analytics

As MapReduce became more popular, it became less important to compute with data using spinning disks or even with solid-state drives. Several changes drove this: first, large memory systems became considerably cheaper. Second, with the wide availability of MapReduce implementations, users without a dedicated cluster computing engineering team could easily do large-scale data processing. These users often had more modest datasets — too large to comfortably use one machine, but not so clearly requiring the fine-grained fault tolerance or disk-based nature of MapReduce.

Also with the increased availability of MapReduce, users were trying to perform calculations for which the MapReduce programming model was not designed. One example was data exploration; MapReduce's inability to analyze moderately sized datasets without the full overhead of reading from and writing to disk made it a poor match for data exploration. Nevertheless, users trudged through long response times with systems like Pig [63], since a full dataset was too large or too expensive to put in a traditional database and some of their exploration steps would require reading and transforming huger volumes of data than would fit in memory. Another issue was that though the abstractions over MapReduce made it easy to write complex joins, these joins required numerous MapReduce jobs, each rereading data from disk. But probably the most pressing need came from machine learning workloads. The core operation for many machine learning workloads required iterative refinement, such as making successive gradient descent steps. This translated, in the MapReduce model, to reloading the input many times from persistent storage.

These pressures lead to alternate designs that eschewed fine-grained fault-tolerance [70, 79] and/or persistent storage of intermediate results [99, 79, 49]. Probably the most successful alternative was Apache Spark [99]. Spark still targeted very large scale, so it did not remove fine-grained fault tolerance, but it allowed users to replace storage of intermediates results on disk with storage of intermediate results in memory.

2.2.3 In-Memory Programming and Execution Model

The programming model of Apache Spark, typical of the modern wave of data processing frameworks, is built around an abstract representation of distributed datasets. Spark calls this abstraction 'resilient distributed datasets' or 'RDDs'. Users of Spark write a *driver* program, which manipulates RDDs as if they are local variables. These computations can be mixed with the ordinary control-flow and operations on values stored locally in the driver program. Spark provides several transformations on RDDs, including both operations which produce new RDDs and operations which produce ordinary values. Notably, these transformations are often parameterized by user-specified functions, which allow systems like Spark to efficiently support a wide variety of computations.

For operations that produce new RDDs, rather than performing the transformation immediately, Spark generally stores the transformation — or for RDDs not produced from other RDDs, the input source. When the value of an RDD is actually needed, Spark performs the computation requested, mostly on *worker* machines. To perform this computation, Spark starts with the requested RDD and examines its dependencies back to input sources or values which are already stored. It then executes the computation of the dependencies, followed by the actual dataset. Each RDD is computed in partitions, whose size, in Spark, is set at creation time. Along the way, Spark stores computed partitions if requested by the user for that RDD by marking it as 'persisted' or 'cached'; otherwise, if the RDD is not marked as such, values are discarded after they are used by the next phase of the computation. Spark's fault tolerance strategy reuses this machinery. To compute a value that is lost due to a worker failure, Spark walks it dependencies until it finds a data source or persisted value, then computes starting after these existing values.

Spark supports RDD transformations that require many-to-many distributed communication, such as to implement a group by over a large dataset. To implement these, Spark provides an 'internal' shuffle and aggregation operation. When an RDD is derived from another RDD using a shuffle operation, Spark performs a *map task* for each partition of the input RDD, grouping values by their partition in the output RDD. These tasks write out data to be read later by corresponding *reduce tasks*, each of which produces one partition of the output RDD. Depending on the transformation, Spark may perform aggregation of values, such as sorting values within a partition by key or summing values with a particular key. These operations are supported by doing an in-memory aggregation.

In many practical cases for users of frameworks like Spark, the user of the RDD abstraction is not the end user, but some higher-level library. This includes many libraries distributed with Spark, which provide database-like interfaces or machine-learning primitives. Regardless, the central abstraction in systems like Spark is the ability to manipulate datasets at the driver in a similar manner to local operations and have these translated to distributed operations, performed in a fault-tolerant manner.

2.3 Machine Environments For In-Memory Analytics

Running in-memory computations requires that users obtain resources, and especially memory, on which to run their computations. Just as analytics frameworks have evolved, so have common environments for in-memory analytics. Where, originally, companies would build special-purpose clusters to run an analytics framework, usually Hadoop, this is now less common. Common cases for analytics — especially the lower response times of in-memory analytics — involve getting computing resources on demand from a shared pool.

The need for computing resources are often satisfied using *cloud computing* providers, who offer these resources for rent [9]. Users of such services benefit from the economies of scale in provisioning and maintaining cluster hardware. In addition, the users benefit from *elasticity* — they can pay for resources only when they need them, and obtaining, for example, 100 machines for one hour costs approximately the same as obtaining one machine for 100 hours. This flexibility is particularly relevant for analytics programs, where analytics frameworks can effectively use these additional machines to give answers more quickly.

Simply being able to obtain computing resources quickly, without buying hardware for a dedicated cluster, has been a major enabler of analytics and the move towards in-memory analytics. Without the ability to deploy computing resources quickly, the promises of easier application development and lower latencies from increased use of memory would ring hollow.

The most frequently used cloud computing services for analytics include providers like Amazon EC2[3] and its many competitors[35, 1, 57], which offer virtual machines in a variety of configurations. This is known as ‘infrastructure as a service’ (IaaS). Users of these environments are responsible for configuring each virtual machine, deciding how to assign its resources, and paying even if those resources are left unused. These same cloud computing providers also provide higher-level services, including supporting analytics-like queries. These include tools that automate deployment of an analytics stack on virtual machines of a customer’s choice, like Amazon Elastic MapReduce [5]. It also includes tools which hide the computing power used, like Google’s BigQuery [34], which provides an more limited SQL-like interface, but charges users by the data queried instead of exposing decisions about what machines to use.

When running an analytics program in an IaaS environment, users need to choose both how many virtual machines and which types of VMs to provision. In EC2’s case, they have a choice of 11 different memory to CPU ratios as of this writing, in addition to variation in their I/O capabilities. For any of these types of virtual machines, users can meet any aggregate memory or CPU power requirement by asking for enough machines, with different levels of cost-efficiency. EC2’s competitors often provide a similar array of choices.

Even users who do not run on public infrastructure often face similar tradeoffs. When users run analytics programs on an enterprise’s internal cluster, they are likely to share that cluster with other users using a scheduler similar to Google’s Borg [91] or Apache Mesos [39] or Apache Hadoop YARN [87]. Alternately, some institutions provide an environment that imitates an commerical IaaS cloud provider using software like OpenStack [64]. For all of these environments, even if the user is not somehow charged for resources like the public IaaS providers, they still need to decide on a configuration request for each allocated worker. Cluster schedulers generally will rely on this resource request to try to achieve high utilization of the cluster, allowing its potentially many users to share it effectively.

2.4 A History of Memory Management Inside Spark

As Spark has moved from a research prototype to a commercial system, it has evolved. Although the core abstractions provided by Spark have remained the same, the methods by which Spark manages its memory to implement these abstractions have changed substantially. Generally, Spark has done more and more explicit tracking of its memory usage, improving its performance and handling of rare situations. In this section, I review Spark’s evolution towards more and more explicit management of memory and the likely motivations behind this. Separately, in Chapter 5, I will look at Spark’s memory model from the perspective of modeling its memory requirements.

2.4.1 The Early Days: Weak References

Early versions of Spark used Java's '*soft references*' to do cache eviction and decide how long to keep intermediate results in memory instead of recomputing them. Spark stored data in memory as objects in the Java virtual machine on which it ran and relied on the virtual machine's garbage collector to decide when to evict cached data. Soft references indicate to this garbage collector that the reference should cleared, allowing the referenced data to be freed, when there is enough memory pressure. Unfortunately, the behavior of garbage collectors of popular Java implementations with soft references in practice is quite poor. The policy in OpenJDK, the most popular Java implementation for servers, generally clears soft references after a substantial amount of time has already been spent doing garbage collection, rather than doing so proactively. This is complicated by the garbage collector generally lacking good information about how to choose between multiple soft references to clear.

The performance of soft references is unacceptable due to this reluctance of the garbage collector to clear them and the unpredictable choice of which references to clear. The drop in performance from keeping data cached is non-obvious: the problem is that the frequency of garbage collections is proportional to the inverse of the free space available (see Chapter 6). When a program is producing temporary objects, there can be significant free space available after each garbage collection, therefore the logic which frees soft references may never be triggered. (OpenJDK has multiple heuristics to detect that it needs to free memory, including a threshold based on an excessive portion of program runtime being spent in GC, so there are limits to the scale of this effect.) However, the extra space taken up by the data referenced only through the soft reference can easily reduce the free space after each collection from, say, 4 GB to 1 GB, easily quadrupling the GC overhead.

2.4.2 Maturity: Explicit Accounting

Spark quickly abandoned this reliance on garbage collectors to free 'cache' memory. This required Spark to peek under the covers of its language runtime and understand how objects are stored. Consequently, Spark began estimating the size of cached objects. To use this sizes, Spark implemented an explicit replacement policy rather than relying on a garbage collector's choice. A pure least recently used (LRU) policy would produce the worst possible performance for application of particular interest to Spark. For example, a typical implementation of gradient descent, a common primitive for machine learning algorithms, in Spark would make passes in partition number order over the input data each iteration. With an LRU policy, Spark would never get any advantage to storing the values in memory. Instead Spark avoids evicting a partition from a dataset to make room for another partition from the same dataset.

Beyond explicitly managing its cache, Spark confronted other sources of potentially excessive memory usage. There were cases where memory outside of cached data would have non-trivial sizes and thus Spark users would surprisingly run out of memory. One source was

Spark's implementations of aggregation operations like group-bys and sorts. Originally, after dividing data into partitions, Spark would perform group-by-like operations for a partition entirely in memory. If partitions in the form necessary to perform these operations were large, this could cause memory exhaustion, even though Spark kept its storage for 'cached' data in check. To compensate, Spark switched to a strategy that supported disk-based merge similar to that of Hadoop MapReduce. To implement this, it started accounting for the memory used during these aggregation operations to decide when to fall back to disk. Another source of memory exhaustion was during the computation of a partition to be cached. Previously, Spark would only made eviction decisions after the replacement was already in memory, but this meant that extra space needed to be allocated 'outside' the cache for items that would be stored in the cache. In response, Spark started estimating the sizes of partitions as they were computed instead.

2.4.3 Move to C-like Memory Management

Spark has been evolving to rely less on language runtime memory management. The most notable artifact of this is Databrick's Project Tungsten[75], which is bringing Spark closer to the 'bare metal' — to explicitly choosing its memory representations and manipulating them with native machine code. This approach relies on higher-level abstractions over the Spark core. Rather than manipulating data as native objects in a high-level language runtime, users write code against something more like a database. This higher level of abstraction allows for specialized code generation and the management of data in forms unlike those supported naturally by the language runtime. In addition to the increased complexity of the implementation, a downside is lower generality — by focusing on operations which can be specialized to act on these optimized storage formats, it does not extend to integrating general user defined functions and non-Spark libraries that is, for many users, a key feature of Spark.

2.5 Performance Optimization Approaches

There are variety of approaches system designers have taken to optimize the performance of a system with a complex configuration space. One approach, particularly useful in cases where the performance of small, easily benchmarked piece of code, is to essentially try all possibilities. Other approaches are more systematic: combinations of experts and statistical techniques might build a performance model of the system. Using such performance model would also require trying several configurations, but only to observe some parameters of the model. Some work replaces this training phase with continuous refinement, turning the model into a control loop, particularly suitable for long-lived programs with slowly changing workloads.

2.5.1 Autotuning

In the high-performance computing space, *autotuning* is a common program optimizations technique where many variations of an important function are generated, and each is tested. Typically, an expert will write a program that will take a set of parameters are produce a candidate version of the function with optimizations chosen based on the parameters. Autotuning techniques have produced some of the fastest linear algebra [93, 15] and FFT [27] libraries across a wide range of platforms. Since there are usually many parameters to choose when optimizing mathematical routines, the autotuning search is very expensive, even with some heuristics to to limit for the search. Because of this, more recent proposals [29, 13] use a predictive model in order to explore the parameter space more efficiently than exhaustive search. Even with this improvement, autotuning searches are very resource intensive (often testing thousands or more configurations), since effect of the optimizations is too complex to model accurately with a small number of samples.

2.5.2 Performance Modeling

Many systems have built performance models of applications to configure them. A variety of application types have been targeted: scientific computing workloads (e.g. [71, 44, 11, 21, 60, 23, 26]), OLTP workloads (e.g. [58, 85, 97]), MapReduce-like workloads (e.g. [90, 48, 78, 38, 28, 98, 89]), web workloads ([47, 86]). Additionally, some work [78, 71, 96, 40] attempts to be mostly application-agnostic, observing and controlling application behavior at the VM level, though sometimes supplemented by some application-level progress information.

A performance model matches an incoming job and its configuration to some estimated performance. For a model to be useful, one must find attributes of incoming jobs that both predict performance and can be easily observed. When the performance model needs to work for new jobs and unused datasets, developing the model is therefore non-trivial. In addition, the model is likely to be limited to the particular type of model queries and perhaps to the particular type of workload. For applications with nice parallel structure, like MapReduce, running a sample of the parallel tasks in the job can be a good way to gather representative statistics [48, 98]that can predict performance. Some systems rely on some static analysis of the unknown programs, for example, they might use query plans [30] or general program analysis techniques [60] to produce an input to the performance model.

A majority of performance modeling approaches rely on fitting observed performance and workload parameters to a statistical model. The statistical model can either be a generally applicable regression model (i.e., trained from a set of "representative" workloads), an expert-chosen model (i.e., based on the expert's understanding of bottlenecks in the application), or a combination of the two. Even for the general regression models, some expert choices need to be made about what features are important enough to extract from the programs. The accuracy of a performance model depends on whether these expert's choices and any training workloads really capture the performance regime during the overall system's use.

Another category of performance models can be seen in PACE [60], which, for scientific computing applications, proposes to statically analyze applications to synthesize a trace of important hardware events before the program can be run on real input. These hardware events are fed into expert-built models of hardware performance. The accuracy of statically based approaches is limited because such static models cannot account for variations in program behavior based on input. There are also practical concerns about the static analysis: static analysis tools are imprecise and may have trouble reproducing even the non-input-dependent behavior of some programs, and many programs are not available in format suitable for static analysis tools. For example, programs written in multiple languages or include libraries for which source code is not available may be very challenging for static analysis tools.

Although performance modeling approaches have been successful in predicting the runtime of MapReduce-like programs across a range of configurations, these approaches do not readily generalize to in-memory analytics. Even for systems like FMEM [98], which has a primary goal of predicting memory usage, the prediction is for temporary usage that is largely independent of large-scale properties of the dataset being processed. Furthermore, these predictions focus on determining actual memory usage given a particular heap size constraint rather than determining whether a particular heap size is acceptable.

Performance modeling systems for MapReduce that are concerned with memory configuration have generally focused on modeling the behavior of short-lived individual tasks whose output begins and ends on disk. Consequently, side-effects for a misconfiguration are generally increased disk I/O for an already disk-based sort, which tends to degrade performance relatively smoothly. Where there are out-of-memory failures in these systems, they are easier to predict from executions over sampled data since the memory exhaustion tends to comes from temporary data and never from data persisting in memory between tasks. In contrast to these disk-based systems for which performance costs of using disk are normal, systems like Spark usually have a performance 'cliff' when the working data size exceeds the available memory and disk is used for the first time.

Where performance modeling systems that deal explicitly with in-memory analytics, like Ernest [89], a statistical performance modeling system for Spark, they have generally sidestepped this memory exhaustion issue. Ernest uses a regression model to learn the performance envelope of a particular application as the configuration changes. But the primary goal is to characterize how performance changes when the (typically machine-learning) application is within the good performance space to meet a particular time or cost target. The generalized linear regression model Ernest uses for an application is unlikely to have any examples of memory exhaustion in its training set.

2.5.2.1 Cache Modeling

Outside of large-scale data processing, one domain where performance models have been very necessary is hardware design. An area of particular interest for large-scale data processing are how hardware designers evaluated different cache designs. To do so, they needed to evaluate many cache designs given the access pattern of a representative benchmark of programs. Traditional techniques for cache simulation are already very expensive, as a simulated cache access is likely to be thousands of times slower than a real cache access. Adding the complexity of trying many cache designs to this would seem to make such analysis impractical.

This lead to 'stack algorithms' approaches [55, 12, 62, 83, 45], which can track the behavior of caches across a wide range of cache configurations with a single analysis pass. These approaches rely on the replacement policy having the property that the items in a cache of size N are also in a cache of size $N+1$. This allows the occupancy of many possible caches to be tracked in their namesake stack — with the least frequently evicted item inserted on top of the stack. Naive algorithms for stack analysis are quadratic — still better than a straightforward simulation at each of millions of possible cache parameters, but unacceptable. As detailed in [45], approaches based on splay trees can accelerate this analysis to amortized log-linear time. Even without the use of splay trees, [45] observes that temporal locality makes simpler techniques, such as organizing the stack into bins whose sizes are tracked, achieve similar performance.

2.5.3 Resource Management Control Loops

Most work focused on choosing configurations from performance models has assumed that a single configuration needed to be chosen. For resource allocation, this has traditionally been true since users needed to provision for peak. The rise of cloud computing, which has sometimes made reducing allocated resources practical and money-saving, has lead to several proposals to resize applications based on varying demands. These include commercial solutions [77, 4] and research prototypes [85, 78, 47, 17, 40, 37]. Typically, these systems target interactive workloads whose demand varies due to external factors. Their goal is to fit resource usage to this varying demand. Usually, the demand is matched by adjusting the number of (virtual) machines assigned, but occasionally more fine-grained mechanisms such as adjusting a virtual machines' CPU share are assumed (as in [78, 40, 37]).

One particular challenge for autoscaling systems is dealing with sudden spikes in demand. One approach, used by the SCADS Director [85], is to always have machines running and ready to deploy in case of a sudden demand spike and to use an open-loop predictor so that the needed capacity can be determined. Systems that are closed-loop (such as [102, 78]) cannot determine how much capacity is needed to satisfy a spike after existing resources are saturated, so they are careful to scale up resources quickly (at the cost of allocating resources that will not be used) and to converge to allocating enough resources to account for typical variance in application demands.

2.5.3.1 Memory control loops

Most control loop schemes that relied on resource usage monitoring adjust allocations based on the CPU usage since the workloads they controlled were usually CPU-bound. If they adjusted other resources, it was usually only incidentally, such as by adjusting a number of allocated virtual machines. A small number of systems (such as [40, 37]) have focused on controlling the memory usage to make better use of memory on a single node. Typically, they focus on inferring the memory-performance tradeoffs of long-running interactive services (like web serving or OLTP databases).

Memory usage tends to be more stable than CPU usage, but adjusting it is more difficult because memory may contain data that cannot be safely discarded. A classic mechanism for adjusting memory usage is swapping, but its performance impact is usually unacceptable with modern memory and disk speeds. Thus, more than determining the correct memory allocations, the major challenge for schemes that adjust memory usage is avoiding these performance problems by adjusting estimates to ensure headroom and choosing better mechanisms to adjust memory usage. Recent attempts to adjust memory usage usually apply pressure to existing memory management systems, such as page cache managers or garbage collectors. Memory is actually freed by the normal action of these systems in response to increased demand for memory; for example by evicting cold pages that are already written on disk or by increasing the frequency of garbage collector sweeps.

Since memory management is generally focused on sharing memory on a single machine, an alternative to directing coexisting memory managers is to use a shared cache, whose behavior can easily take into account all the users on the machine. The Transcendent Memory proposal for Linux [53] proposes using such a shared cache (implemented in a hypervisor) for as “second-chance” cache for the guest operating system caches or for application caches aware of the service.

2.6 Garbage Collector Tuning

Garbage collectors’ performance is a source of much frustration to programmers. Usually, the frustration is focused around garbage collector pauses. The earliest garbage collectors were based on a ‘stop-the-world’ mark and sweep approach. These garbage collector designs were particularly unfortunate for interactive applications, so many later garbage collector designs focus on concurrency. They trade some efficiency for doing more of the work of the garbage collector outside of noticeable pauses. The culmination of this trend are very low pause garbage collector targeting large heaps like Azul’s ‘C4’ garbage collector [10] or the Shenandoah garbage collector [20] planned for OpenJDK.

But while these collectors are asynchronous, they still require tuning for good performance. Concurrent collectors will fail to be concurrent when their concurrent collection work cannot keep up with allocations given the available space. Moreover, pause times are not the only concern. For applications like batch analytics, how much the garbage collector competes with the application’s computations for the CPU is significant. Using the older, non-concurrent collector designs reduces these overheads.

2.6.1 Manual Tuning Advice

Regardless of whether it is latency or throughput that matters most, tuning the garbage collector is very important to the performance of applications that run on managed runtimes. This has had lead to many attempts at providing heuristic tuning advice such as in [31, 42]. Typically, this tuning advice has been focused on long-running server applications. It is for these applications that time investment in tuning is most valuable and the latency cost of garbage collector pauses is worst. In addition, the default settings for garbage collectors has often been a throughput-focused one, so merely switching to a more latency-focused concurrent collector obtains a very large benefit.

Being focused on long-running services, tuning advice typically was based on assuming a steady-state workload. It generally mixed rules of thumb — such as allocating a particular portion of the available space to different garbage collector regions — and making adjustments based on garbage collector logs. The advice in [42] is to systematically make adjustments to each of several garbage collector parameters based on observing performance and sizes once the application is in a steady state. These adjustments are informed by knowledge of what should achieve best performance — such as making survivor spaces large enough to avoid having 'short-lived' objects enter tenured space. But, generally, these approaches are based on making small adjustments to individual garbage collector parameters and checking whether they improve performance.

2.6.2 Improving Application/GC integration

Much effort has been spent on trying to make garbage collectors themselves better suited toward common applications. In addition to making garbage collectors that happen to behave better given the allocation patterns experienced by typical programs, some research has focused on allowing programs to integrate with the garbage collector more directly.

One approach, shown in Broom [32], would allow applications to divide memory into regions based on how the application uses memory. This allows sets of objects, such as those making up a stored dataset in a system like Spark, to be effectively managed as a unit by the garbage collector. In existing systems, the garbage collector does not receive any hint that the lifetimes of these objects are related and must separate them from temporary data, or longer-lived data, by scanning it. Instead, the frameworks that know when these sets of related objects should become unreachable can tell the garbage collector about this explicitly, substantially improving its performance.

Other proposals have focused on coordinating not the arrangement of memory so much as when garbage collection activity happens. [52] tackles the difficulties of garbage collection latency across distributed system by controlling when garbage collections occur based on application needs. This avoids exaggerating the effect of pauses by letting them occur when the worker node experiencing the pause has a large number of pending requests from other nodes. In the case of throughput-oriented systems, this can be done by coordinating garbage collection pauses so workers do them simultaneously; in response-time oriented system, work can be routed away from workers as they perform garbage collection activities. Similarly, [82] proposes an API to make applications aware of where garbage collection events are happening so they can make these types of adjustments.

2.6.3 Runtime Adjustments

Beyond new garbage collector designs, another common approach for tuning garbage collectors is to adjust the space allocated to the garbage collector at runtime. This is particularly relevant when managing multiple garbage collected applications on a shared system or balancing garbage collected memory with operating system caches. [18, 16] examined control loop approaches for adjusting the amount of memory assigned to garbage collection to maximize total system throughput. These focus on systems where it is practical to adjust the amount of space allocated to the garbage collector at runtime, for example because the space can be taken from OS caches or other managed programs. Consequently, they act by observing the total extent of garbage collection activity relative to other memory pressures (page faults and other applications' garbage collection activity). From these observations, they make heap size adjustments up or down until the system stabilizes.

Another concern of garbage collectors is tuning internal parameters. [88] describes how the sizes of the various regions of the Hotspot generational collector can be adjusted to maximize performance. It uses a control loop-based approach that obtains a local estimate of the likely throughput changes that would result from changes to the sizes of various memory regions. Unlike systems that try to balance language runtime and external memory usage, it relies extensively on counters within the language runtime to estimate the approximate sizes and collection rates.

This explicit estimation of sizes and collection rates is based around assuming application behavior exists in a steady state and modeling the corresponding internal garbage collector behavior, the starting assumption I also use in Chapter 6. Unlike my work in Chapter 6, the approach in [88] is distinctly online. After adjustments are made to garbage collector parameters, it observes how the garbage collector behavior changed. Based on these changes and a performance model, it determines in which direction to make future adjustments in order to maximize total throughput. Being an online system, [88] is concerned with adjusting how memory is split between different managed regions rather than the total amount of memory available, which is relatively inflexible at runtime in most practical applications.

2.7 Summary

Numerous efforts have confronted the increasing complexity of large-scale computing platforms. As new abstractions have made large-scale data processing easier for users, they have many knobs that must be configured for best performance. Although experts may always adjust these parameters to squeeze the last bits of performance, researchers have tried to tame this complexity by automating this tuning. Typically, they construct performance models of the system to guide their choices. These models range from completely generic approaches based on exploring all possible configurations to manually constructed approaches based on a simulation of the underlying resource demands. Given these performance models, they either devise a control loop or a system to make static recommendations based on a training phase.

Of particular difficultly for these systems are the configuration knobs related to memory usage. In addition to being difficult to adjust at runtime because of the data movement costs, memory allocations cross layers. Applications use memory through a language runtime, providing the runtime incomplete information about its memory and performance requirements. Some efforts have focused on improving this barrier, especially in the distributed case, by providing interfaces for applications to indicate their preferences to garbage collectors and vice-versa. Similarly, other proposals have focused on improving communication between garbage collectors, since otherwise operating systems cannot profitably reclaim memory from a garbage collected heap.

In this dissertation, I configure memory allocations across many layers with a modeling-based approach. I use an analysis of the operation of in-memory analytics frameworks and their underlying language runtimes to guide data collection and recommendations. A difficulty of modeling memory usage this way is that allocating too little memory has very negative effects on performance — thrashing. Prior work that deals with memory usage handles this by having an online mechanism to adjust memory usage in response to these problems or by operating well away from the 'minimum' memory allocation. In contrast, I rely on the construction of my model to confidentiality avoid these problems.

Chapter 3

User Use of Shared Computing Environments

Effectively dividing the resources of shared computing facilities has been a major concern of system operators since the dawn of time-sharing systems. In modern commodity clusters, to obtain the advantages of consolidation, operators of shared infrastructure need to partition resources dynamically. Despite many proposals for new resource managers [46, 94, 100, 91] for these environments, there has been little visibility into the workloads these cluster managers face. To help better understand these workloads, Google released to the research community a substantial cluster usage dataset from one of their production clusters [95]. I analyzed this "trace" to understand the environment in which modern resource managers operate.

I was particularly interested in what information is available for cluster management systems to run batch programs, such as data analytics programs, alongside interactive workloads. Until recently, computing environments where this was possible were rare: the sharing mechanisms usually required coarse-grained division of resources, for example assigning whole compute nodes to tasks, and running different kinds of workloads together was rare. A common wisdom has been that there should be a great increase in efficiency because interactive workloads have a lot of spare capacity and therefore flexibility to fit batch workloads which can survive occasional failures due to overcommitment of resources. Important to such schedulers is the information available to predict resource usage. With good information, a scheduler should achieve high utilization even after taking account of lost cycles from failed attempts at overcommitting resources.

The viability of this approach may depend on the accuracy of resource requests supplied by applications. To avoid placing tasks only to have their work wasted because they do not fit, schedulers want to predict resource usage in advance. This is especially true for memory usage; programs with insufficient memory are not likely to simply run somewhat slower. Schedulers might predict resource usage more accurately by monitoring applications. But the scheduler's information, through generally application-agnostic monitoring, is limited, especially at the most important application-launch time. Furthermore, if the scheduler acts too aggressively on this information, it may surprise users who expect consistency in the resources their applications receive.

In analyzing this trace, I examined whether Google's scheduler had the information they needed — either directly from their users or through making predictions from application-agnostic monitoring. I particularly wanted to understand this about memory usage, since in-memory analytics places increased pressure on memory and it is more difficult to overcommit memory without relocating data. As far as the scheduler was lacking sufficient information, I wanted to understand what the causes of that gap were: When the scheduler was getting poor resource requests, what was likely motivating users to make these bad requests? If the scheduler tried to predict resource usage beyond resource requests, what problems did or would overcommitting memory resources be likely to cause?

My examination showed that the scheduler seemed to lack what it needed to predict resource usage to overcommit memory well. Resource requests was a fairly poor predictor. The best simple predictor of actual resource usage I found, instead, was recent resource usage. While promising, this mechanism gives no good answer for what a scheduler is supposed to do when placing a program initially. Even later, relying on it would make it hard to provide good guarantees to users who wish to avoid spurious task evictions and is problematic for programs whose resource usage might change over time.

Perhaps in response to these concerns, the Google scheduler was fairly deferential to resource requests. Still, its modest attempts at overcommitment resulted in high cost in terms of evictions to achieve only around 50% utilization as shown in Figure 3.1. The gap between resource requests and average usage accounted for most of this difference. The workload was very diverse in many ways: urgency, resource requirements, the gaps between resource requests and usage, how much resource usage varied within a job, etc. While this is precisely the diversity that should create opportunities to colocate programs with non-interfering resource demands, this diversity means that a simple examination of patterns or averages would not correct the poor resource requests. Instead, the Google scheduler rationally relied upon user resource requests largely as they were, leaving most resources unused.

In examining this trace, I first review the information collected in the Google trace, providing an overview of its size and terminology I will use to describe it. Then, I examine the mix of types of jobs, revealing the great variety of requests which is a challenge and opportunity for the scheduler: a challenge because the constant high load requiring very fast scheduler decisions and their lack of uniformity making simpler slot-based approaches untenable; an opportunity because there should be a wealth of small tasks to fill unused space on the cluster. I then examine how well the scheduler does and could do at taking advantage of this opportunity. I show that there is a large gap between what users request and require and investigate the sources of this discrepancy.

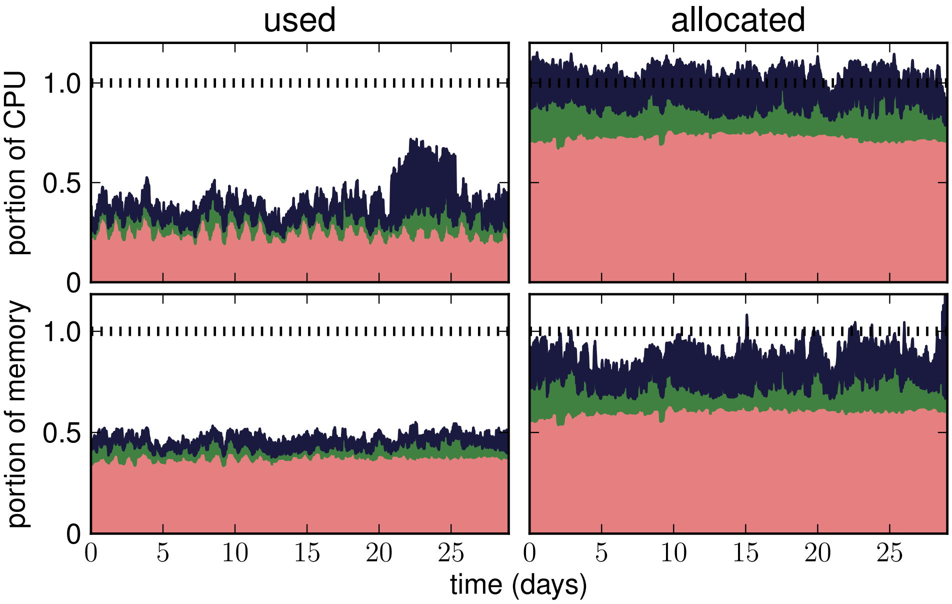


Figure 3.1: Moving hourly average of CPU (top) and memory (bottom) utilization (left) and resource requests (right). Stacked plot by priority range, highest priorities (production) on bottom (in red/lightest color), followed by the middle priorities (green), and gratis (blue/darkest color). The dashed line near the top of each plot shows the total capacity of the cluster.

3.1 The Environment of a Google Trace

The Google trace provides a month of detailed request and usage information for a collection of approximately 12K physical machines that are managed by a common cluster scheduler. Users submitted around 650 thousand *jobs*, each consisting of one or more *tasks*, which are programs to be assigned to and executed on machines. Tasks are not gang-scheduled, though they are typically meant to be run simultaneously and usually execute the software. Each task is specified with various attributes, which can include priority, resource request (an estimated maximum amount of RAM and CPU needed), and constraints (e.g., don't run on same machine as another task of this job or on a machine without an external IP address). The trace includes sufficient information about the machines to match these requests and constraints to machines, including changes in machine availability over time.

The cluster machines are not homogeneous; they consist of three different platforms (the trace providers distinguish them by indicating "the microarchitecture and chipset version" [74]) and a variety of memory to compute ratios. The configurations are shown in Table 3.1. Exact numbers of CPU cores and bytes of memory are unavailable; instead, CPU and memory size measurements are normalized to the configuration of the largest machines.

Number of machines	Platform	CPUs	Memory
6732	B	0.50	0.50
3863	B	0.50	0.25
1001	B	0.50	0.75
795	C	1.00	1.00
126	A	0.25	0.25
52	B	0.50	0.12
5	B	0.50	0.03
5	B	0.50	0.97
3	C	1.00	0.50
1	B	0.50	0.06

Table 3.1: Configurations of machines in the cluster. CPU and memory units are linearly scaled so that the maximum machine is 1. Machines may change configuration during the trace; we show their first configuration.

We will use these units throughout this paper. Most of the machines have half of the memory and half the CPU of the largest machines.

In addition to the CPU and memory capacity and microarchitecture of the machines, a substantial fraction of machine heterogeneity, from the scheduler's perspective, comes from "machine attributes". They are obfuscated ⟨key,value⟩ pairs, with a total of 67 unique machine attribute keys in the cell.

Unlike classic 'cloud computing' environments, users are not isolated from each other through the use of full virtual machines. Instead, users run directly on on the physical machine's operating system. Users are isolated from each other using the Linux kernel's cgroups [68] mechanisms. Unlike full virtualization, this means that users share filesystems and kernel services. This means that there is less overhead to run a user program than a traditional commodity computing environment permitting more fine-grained sharing. The cost is that there is less isolation between users' CPU, memory, and I/O demands.

3.2 Mix of Job Sizes, Frequencies

Unlike environments built for a single type of applications like web serving or scientific computing, the Google cluster ran a very diverse workload. There is a mix of long-running services and short-lived, intermittent batch programs. The promise of operating a shared cluster as Google does is precisely this sort of mix of workloads — the long-running services ought to have substantial flexibility because of the gap between their peak and normal demands. Less time-critical programs might fit in the 'spare' capacity, effectively eliminating capital costs for batch computations.

In addition to the multiple qualitative types of jobs, the trace also illustrates a very wide range of sizes of jobs. In terms of pretty much any dimension — the amount of resources devoted to each task, the number tasks in a job, the total resources devoted to a job, the durations of tasks and jobs — the trace has many orders of magnitude of variety. This variety poses a challenge to cluster schedulers, which need to be efficient to deal with a flood of small requests and still must handle bin-packing issues that arise from large, long-lived allocations.

3.2.1 Mix of Priorities

One signal of differing job types is the priority associated with the tasks. The trace uses twelve task priorities (numbered 0 to 11), which I will group into three sets: production (9–11), middle (2–8), and gratis (0–1). The trace providers indicate that latency-sensitive tasks (as marked by another task attribute) in the production priorities should not be "evicted due to over-allocation of machine resources" [74] and that users of tasks of gratis priorities are charged substantially less for their resources.

3.2.2 Large, Long-Running Jobs

The aggregate usage shows that the production priorities represent a different kind of workload than the others. As shown in Figure 3.2, production priorities account for more resource usage than all the other priorities and have the clearest daily patterns in CPU usage (with a peak-to-mean ratio of 1.3). Usage at the lowest priority shows little such pattern, and this remains true even if short-running jobs are excluded.

The presence of this daily pattern, combined with the guarantees given to production priorities suggests that it is dominated by user-facing interactive services. This is corroborated by looking at the mix of job lengths; as shown in Figure 3.3, production priorities include a majority of jobs that ran longer than a day even though only 7% of all jobs ran at production priority. In contrast, the middle priorities account for a majority of all jobs but only 7% of jobs that run longer than a day. Combined with less of a guarantee against eviction, this suggests that these priorities would be suitable for batch programs to drive the interactive services. The aggregate workload is more like that seen in MapReduce-like systems, with many short-running tasks, not triggered with any obvious periodicity.

These are clearly not perfect divisions of job purpose — each priority set appears to contain jobs that behave like user-facing services would and large numbers of short-lived batch-like jobs (based on durations and utilization patterns). The trace contains no obvious job or task attribute that distinguishes between types of jobs besides their actual resource usage and duration: even 'scheduling class', which the trace providers say represents how latency-sensitive a job is, does not separate short-duration jobs from long-duration jobs. Nevertheless, the qualitative difference in the aggregate workloads at the higher and lower priorities shows that the trace is both unlike batch workload traces (which lack the combination of diurnal patterns and very long-running jobs) and unlike interactive service (which lack large sets of short jobs with little pattern in demand).

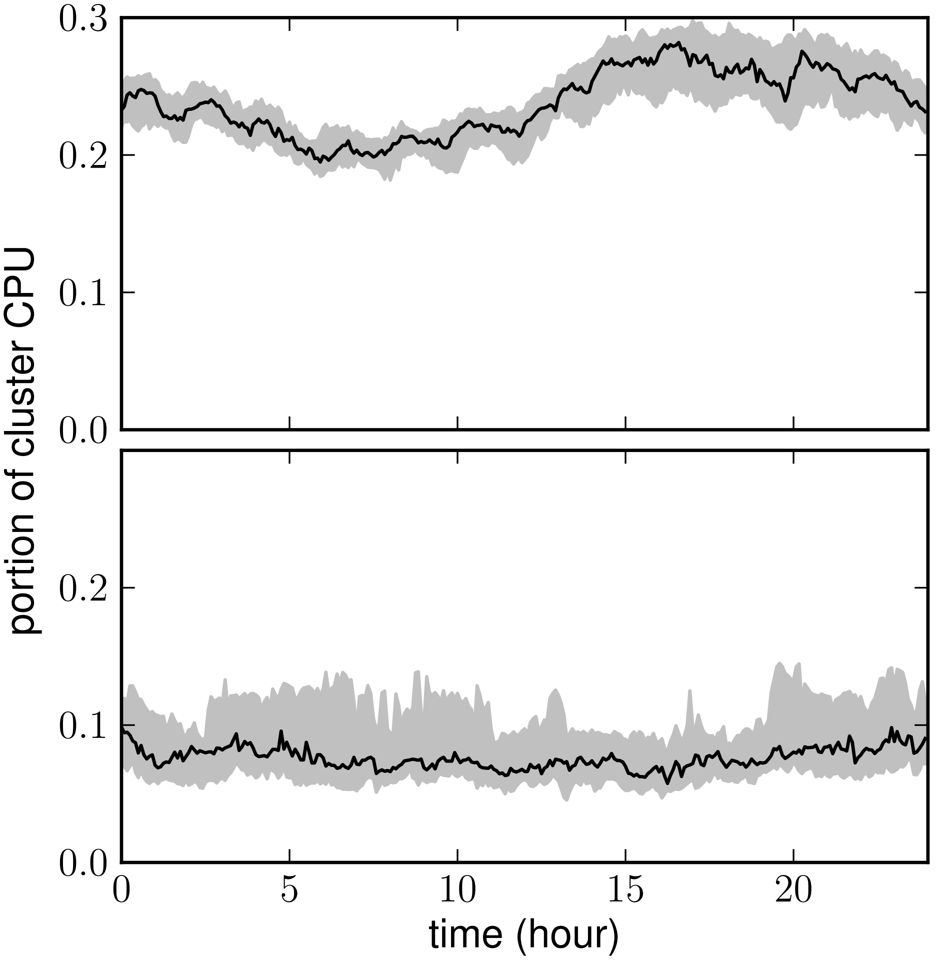


Figure 3.2: Normal production (top) and lower (bottom) priority CPU usage by hour of day. The dark line is the median and the grey band represents the quartiles.

3.2.3 Variety of Task Shapes

Each task has a resource request, which should indicate the amount of CPU and memory space the task will require. (The requests are intended to represent the submitter's predicted "maximum" usage for the task.) Both the amount of the resources requested and the amount actually used by tasks varies by several orders of magnitude; see Figures 3.4 and 3.5, respectively. These are not just outliers. Over 2000 jobs requested less than 0.0001 normalized units of memory per task, and over 8000 jobs requested more than 0.1 units of memory per task. Similarly, over 70000 jobs request less than 0.0001 units of CPU per task, and over 8000 request more than 0.1 units of CPU. Both tiny and large resource requesting jobs include hundreds of distinct users, so it is not likely that the particularly large or small requests were caused by the quirky demands of a single individual or service.

I believe that this variety in task "shapes" has not been seen in prior workloads, if only

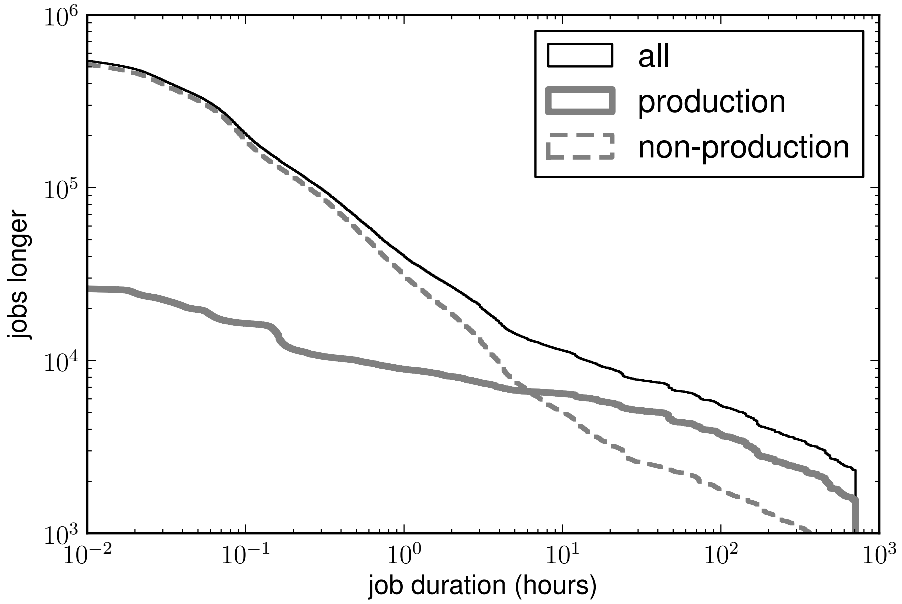


Figure 3.3: Log-log scale inverted CDF of job durations. Only the duration for which the job runs during the trace time period is known; thus, for example, we do not observe durations longer than around 700 hours. The thin, black line shows all jobs; the thick line shows production-priority jobs; and the dashed line shows non-production priority jobs.

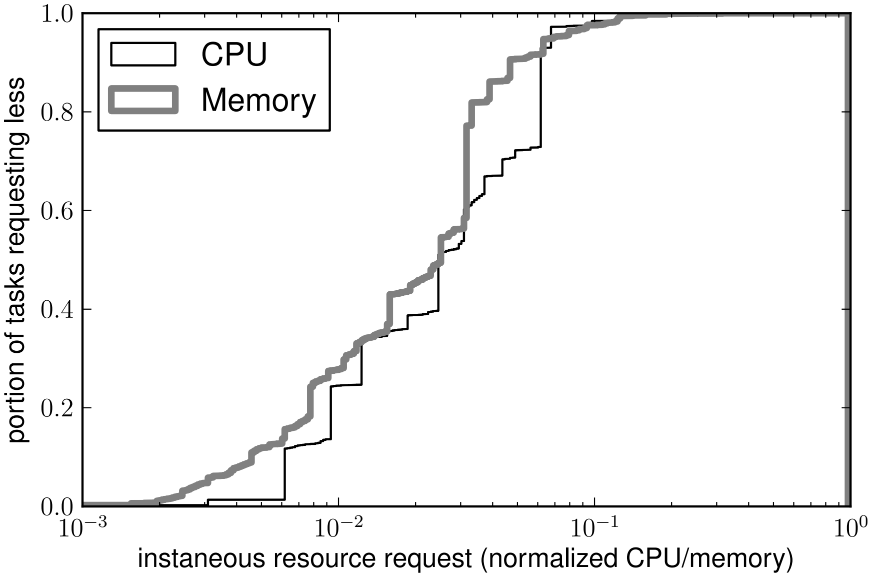


Figure 3.4: CDF of instantaneous task requested resources. (1 unit = max machine size.) These are the raw resource requests in the trace; they do not account for task duration.

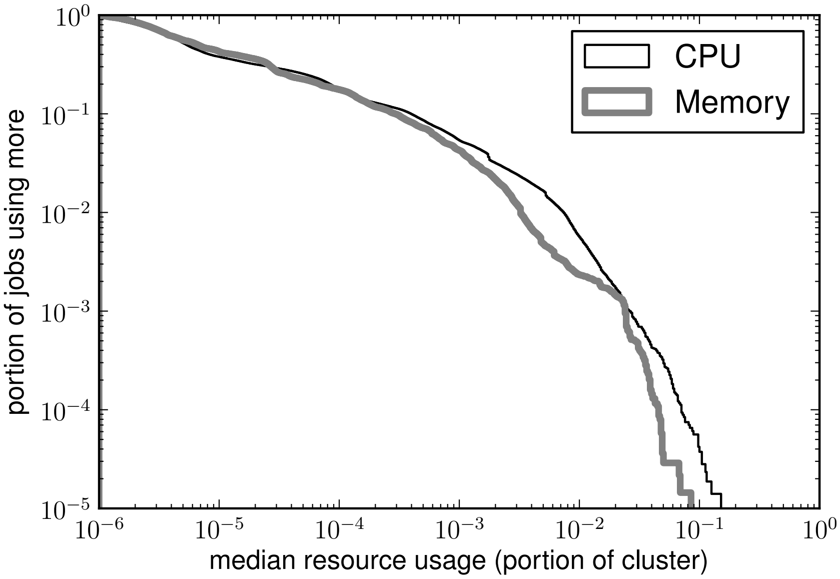


Figure 3.5: Log-log scale inverted CDF of instantaneous median job usage, accounting for both varying per-task usage and varying job task counts.

because most schedulers simply do not support this range of sizes. The smallest resource requests are likely so small that it would be difficult for any VM-based scheduler to run a VM using that little memory. (0.0001 units would be around 50MB if the largest machines in the cluster had 512GB of memory.) Also, any slot-based scheduler, which includes all HPC and Grid installations I am aware of, would be unlikely to have thousands of slots per commodity machine.

The ratio between CPU and memory requests also spans a large range. The memory and CPU request sizes are correlated, but weakly (linear regression $R^2 \approx 0.14$). A large number jobs request 0 units of CPU — presumably they require so little CPU they can depend on running in the 'left-over' CPU of a machine; it makes little sense to talk about the CPU:memory ratio without adjusting these. Rounding these requests to the next largest request size, the CPU:memory ratio spans about two orders of magnitude.

3.2.4 High Scheduler Load from Small Tasks

Since many tasks behave like long-running services, one might expect the scheduler not to have much work from new tasks. This is especially true since the trace providers indicate that MapReduce programs execute separate tasks for the worker and the masters, which server as execution containers for many MapReduce 'tasks'. So, at least some common sources of what would be fine-grained tasks are, in this trace, coarse-grained. However, Figure 3.6 shows that the scheduler must decide where (or whether) to place runnable tasks frequently.

In peak hours, the scheduler needs to make hundreds of task placement decisions per second. Even during quieter times, the average scheduling throughput is several tasks per second.

There are two reasons for the frequency of scheduling events. One is that there are many short-duration tasks being scheduled. Another is that tasks terminate and need to be rescheduled; I will call this *resubmission*. The task terminations preceding these resubmissions are labeled by the trace providers: they are either *failures* (software crashes of a task), *evictions* (task ceased to fit on the machine, due to competing workload, over-commitment of the machine, or hardware failures) or *kills* (underlying reason for a task's death is not available).

Resubmissions account for nearly half of task submissions to the scheduler. However, a principal source of resubmissions (14M of the 22M resubmission events) is tasks which repeatedly fail and are retried. Another major cause (4.5M events) of resubmissions is evictions; as discussed later, most of these evictions are attributable to machine configuration changes or other (higher priority) workload being started on the machine. For the remaining resubmissions (4.1M events), the task is marked as *killed*; these may logically represent software failures or evictions, or other reasons, such as tasks being restarted to change their configuration.

3.2.4.1 Crash-loops

Large spikes in the rate of task resubmissions seen in Figure 3.6 can be attributed to 'crash-loops'. These are cases where the tasks of a job fail deterministically shortly after starting, but they are restarted after these failures. Of the 14M task failures recorded in the trace, 10M are in three crash looping jobs, each of which has tens of thousands of distinct and repeatedly failing tasks. The durations of these three large jobs ranged from around 30 minutes to around 5 days.

Jobs with large numbers of tasks are not the only ones that appear to experience crash-loops. Approximately 2% of the memory-time requested comes from jobs that experience more than 10 failures per task over the course of the trace. Most of these failures occur in lower priority jobs (which are probably used for development), but there are some noticeable (but smaller in terms of task failure count) crash loops even at the production priorities.

3.2.5 Small jobs

Even though the scheduler runs large, long-lived parallel jobs, most of the jobs in the trace only request a small amount of a single machine's resources and only run for several minutes. 75% of jobs consist of only one task, half of the jobs run for less than 3 minutes, and most jobs request less than 5% of the average machine's resources. These small job submissions are frequent, ensuring that the cluster scheduler has new work to schedule nearly every minute of every day.

Job submissions are clustered together (Figure 3.7 shows job interarrival times), with around 40% of submissions recorded less than 10 milliseconds after the previous submission

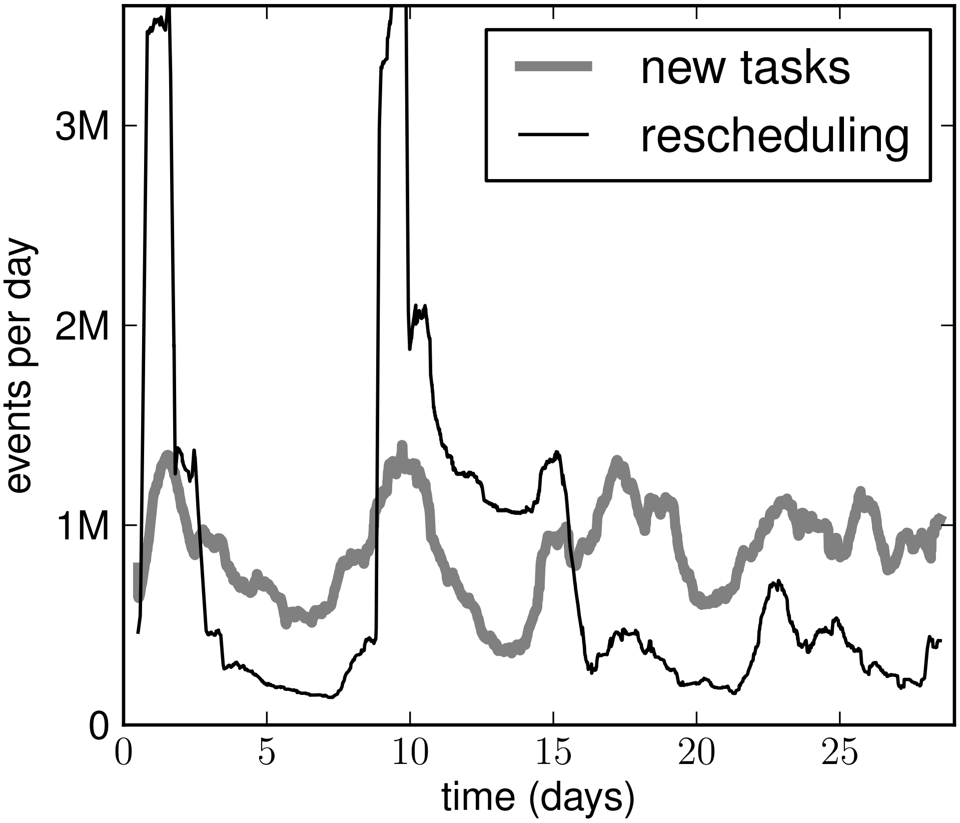


Figure 3.6: Moving average (over a day-long window) of task submission (a task becomes runnable) rates.

even though the median interarrival period is 900 ms. The tail of the distribution of interarrival times is power-law-like, though the maximum job interarrival period is only 11 minutes.

The prevalence of many very small interarrival periods suggest that some sets of jobs are part of the same logical program and intended to run together. For example, the trace providers indicate that MapReduce programs run with a separate 'master' and 'worker' jobs, which will presumably each have a different shape. Another likely cause is embarrassingly parallel programs being split into many distinct single-task jobs. (Users might specify many small jobs rather than one job with many tasks to avoid implying any co-scheduling requirement between the parallel tasks.) A combination of the two of these might explain the very large number of single-task jobs.

3.2.6 Evictions

Evictions are also a common cause of task rescheduling. There are 4.5M evictions recorded in the trace, more than the number of recorded software failures after excluding the largest

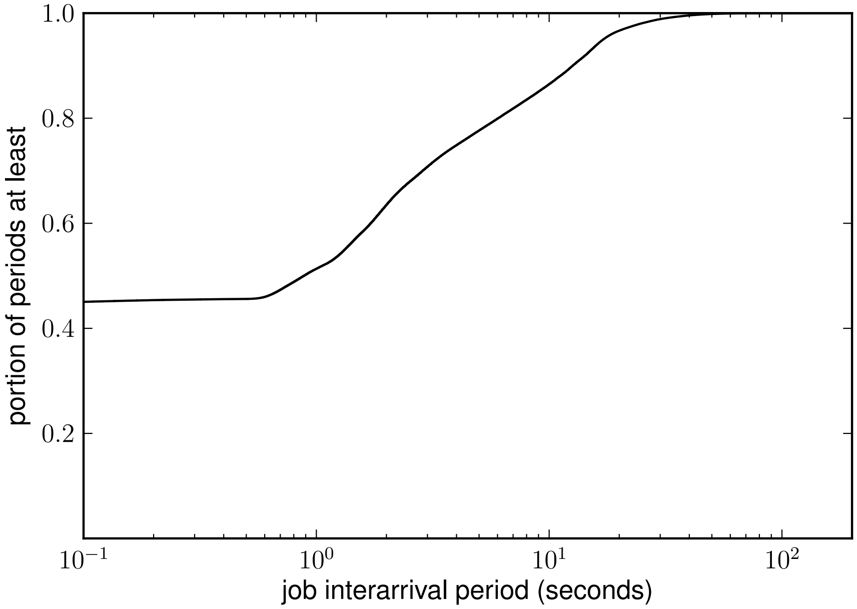


Figure 3.7: Linear-log plot of inverted CDF of interarrival periods between jobs.

crash-looping jobs. As would be expected, eviction rates are related to task priorities. The rate of evictions for production priority tasks is comparable to the rate of machine churn: between one per one hundred task days and one per fifteen task days, depending on how many unknown task terminations are due to evictions. Most of these evictions are near in time to a machine configuration record for the machine the task was evicted from, so we suspect most of these evictions are due to machine availability changes.

The rate of evictions at lower priorities varies by orders of magnitude, with some weekly pattern in the eviction rate. Gratis priority tasks average about at least 4 evictions per task-day, though almost none of these evictions occur on what appear to be weekends. Given this eviction rate, an average 100-task job running at a gratis priority would expect about one task to be lost every 15 minutes. These programs must tolerate a very high “failure” rate by the standards of a typical cloud computing provider or Hadoop cluster.

Almost all of these evictions occur within half a second of another task of the same or higher priority starting on the same machine. This indicates that most of these evictions are probably intended to free resources for those tasks. Since the evictions occur so soon after the higher priority task is scheduled, it is unlikely that many of these evictions are driven by resource usage monitoring. If the scheduler were measuring resource usage to determine when lower-priority tasks should be evicted, then one would expect many higher-priority tasks to have a ‘warm-up’ period of at least some seconds. During this period, the resources used by the lower-priority task would not yet be required, so the task would not be immediately evicted.

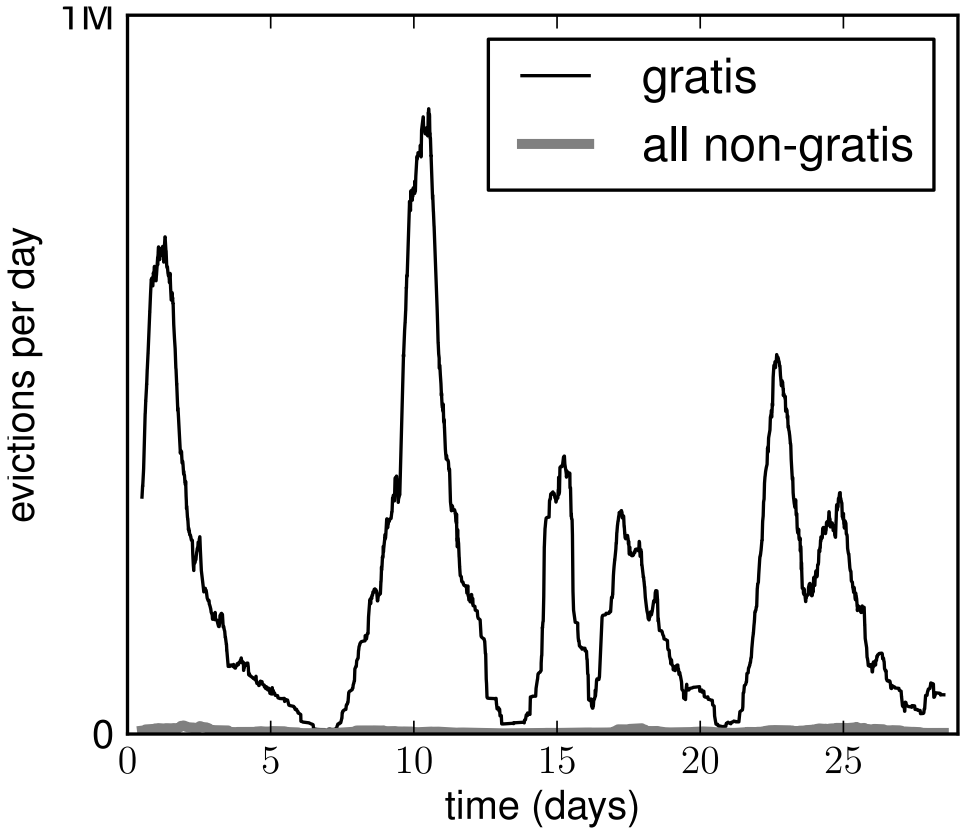


Figure 3.8: Moving average (over day-long window) of task eviction rates, broken by priority.

Given that the scheduler is evicting before the resources actually come into conflict, some evicted tasks could probably run substantially longer — potentially till completion, without any resource conflicts. This does not mean that Google's schedulers reliance on resource requests is necessarily irrational: it maybe necessary to ensure that resources are reliably available to high priority programs or may avoid delaying eviction until a time where it is even more expensive. Nevertheless, to estimate how often delaying eviction might help, I examined maximum machine usage after evictions events and compared these to the requested resources of evicted tasks. After around 30% of evictions, resources requested by the evicted tasks appear to remain free for an hour after the eviction, suggesting that these evictions were either unnecessary or were to make way for brief usage spikes we cannot detect in this monitoring data.

3.3 Allocations versus Usage

Given my interest in the accuracy of resource requests and Google's apparent reliance on them for scheduling and eviction decisions, I endeavoured to take advantage of having information about the resource requests themselves and the resource usage they ultimately resulted in. Ultimately, I wanted to understand how well these requests reflected usage and whether it would be easy to make them reflect usage.

3.3.1 Allocation And Resource Usage Data

Resource requests are relatively straightforward to quantify. Each task, at any given point in time, has a particular CPU and memory request. Measuring the total amount logically allocated amounts to counting, at any given instant, the sum of these requests. Resource usage information, in contrast, does not have a definite instantaneous number — and, indeed, cannot since CPU usage is about the time spent occupying a core. And, even for memory usage — which should have a definite instantaneous value — operating system and shared memory on the machines means that there are multiple options and potential distortions in measurements.

The trace providers include usage information for tasks in five-minute segments. At each five-minute boundary, when data is not missing, there is at least (and usually exactly) one usage record for each task which is running during that time period. Each record is marked with a start and end time. This usage record includes a number of types of utilization measurements gathered from Linux containers. Since they are obtained from the Linux kernel, memory usage measurements include some of the memory usage the kernel makes on behalf of the task (such as page cache); tasks are expected to request enough memory to include such kernel-managed memory they require. I will usually use utilization measurements that represent the *average* CPU and *average* memory utilization over the measurement period.

To compute the actual utilization, I divided the trace into the five-minute sampling periods; within period, for each task usage record available, we took the sum of the average CPU and memory usage weighted by the length of the measurement. I did not attempt to compensate for missing usage records (which the trace producers estimate accounts for no more than 1% of the records). The trace providers state that missing records may result from "the monitoring system or cluster [getting] overloaded" and from filtering out records "mislabeled due to a bug in the monitoring system" [74].

3.3.2 Lost Capacity

Figure 3.1 shows the utilization on the cluster over the 29 day trace period both in terms of the measured resource consumption (left side of figure) and 'allocations' (requested resources of running tasks; right side of figure). Based on allocations, the cluster is very heavily booked. Total resource allocation at almost any time account for more than 80% of the cluster's memory capacity and **more** than 100% of the cluster's CPU capacity. Overall usage is much lower: averaging over one-hour windows, memory usage does not exceed about 50% of the capacity of the cluster and CPU usage does not exceed about 60%.

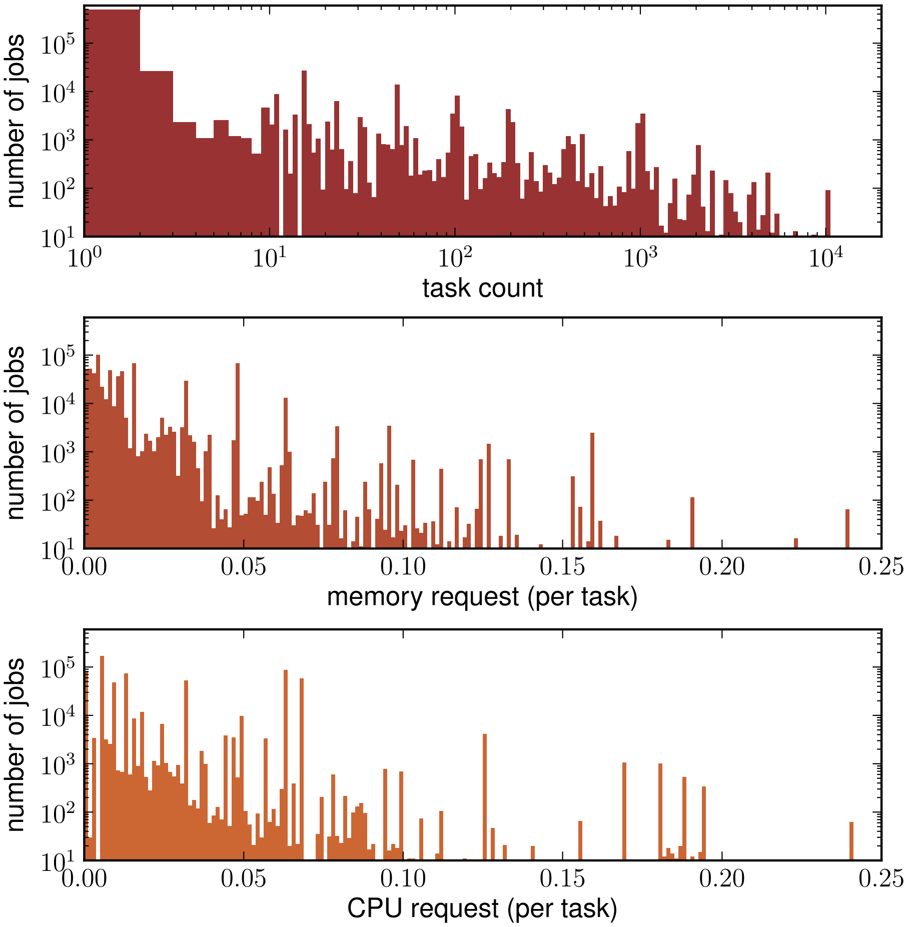


Figure 3.9: Histograms of job counts by task count (top), memory request size (middle) and CPU request size (bottom). Note the log-scale on each y-axis and the log-scale on the top plot's x-axis. We speculate that many of the periodic peaks in task counts and resource request sizes represent humans choosing round numbers. Memory and CPU units are the same as Table 3.1. Due to the choice of x-axis limits, not all jobs appear on these plots.

A natural conclusion is that something is wrong with resource requests. One could speculate that this problem was merely one of a lack of user incentives or resource monitoring tools. Perhaps the application-agnostic information available to the cluster scheduler could fix these requests. If so, it would seem that this gap between allocations and usage would be irrelevant.

3.3.3 Non-automation

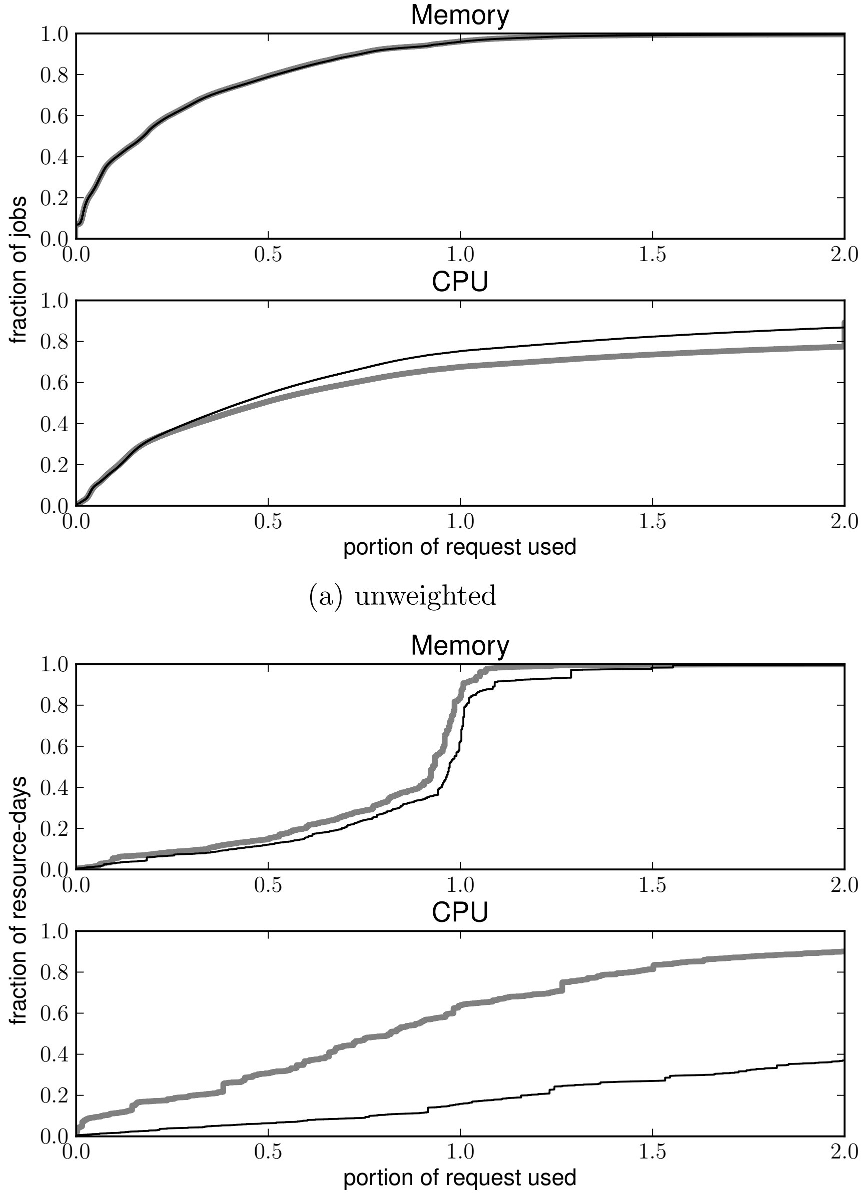
Resource requests appear to be specified manually, which may explain why they do not correspond to actual usage. One sign of manual request specification is the uneven distribution of resource request sizes, shown in Figure 3.9. When users specify parameters, they tend to choose round numbers like 16, 100, 500, 1000, and 1024. This pattern can clearly be seen in the number of tasks selected for jobs in this trace; it is not plausible that the multiples of powers of ten are the result of a technical choice. We cannot directly identify any similar round numbers in the CPU and memory requests because the raw values have been rescaled, but the distribution shows similar periodic peaks, which might represent, e.g., multiples of 100 megabytes of memory or some fraction of a core. For memory requests, it is unlikely that these round numbers accurately reflect requirements. For CPU requests, whole numbers of CPUs would accurately reflect the CPU a disproportionate number of tasks would use [72], but it seems unlikely that the smallest 'bands' (at around 1/80th of a machine) represent a whole number of cores on a 2011 commodity machine.

Requests in this trace are supposed to indicate the "maximum amount . . . a task is permitted to use" [74]. Large gaps between aggregate usage and aggregate allocation, therefore, do not necessarily indicate that the requests are inaccurate. If a task ever required those CPU and memory resources for even a second of its execution, then the request would be accurate, regardless of its average consumption. Thus, resource requests could be thought of as reflecting the maximum *anticipated* utilization of CPU and memory for the requesting task.

Without the ability to rerun programs and understand their performance requirements, we cannot find the "correct" resource request for applications, i.e., a request that reflects the maximum resources that the tasks may need. I extrapolate from the actual usage observed: the maximum or some high (e.g. 99th) percentile of the actual usage samples should be a good estimate of an appropriate resource request. Since the tasks in a job are usually identical, I assumed that each resource request must be suitable for each task within a job. To avoid being sensitive to any outlier tasks, I took the 99th percentile of these estimates for each task as the extrapolated resource request for all tasks of the jobs.

The differences between the resource request we extrapolated and the actual request is not what one would infer from the aggregate usage and requests shown in Figure 3.1. One might infer that jobs generally requested twice as much as they used from the aggregate figures. But considering the high percentile usage as an estimate of the actual resource request, the typical request 'accuracy' is very different.

As shown in Figure 3.10b, jobs accounting for about 60% of the memory allocated fall within 10% of my estimate of their appropriate request. The remaining jobs over-estimate their memory usage. Memory requests rarely under-estimate their jobs' memory utilization by a large factor, probably because tasks are terminated if their memory request is exceeded by too much. Memory overestimates amount to about 20% of the total memory allocation while the total difference between the memory usage and allocation is about 50% of the memory allocation. The remaining 30% can be accounted for by the difference between



(a) unweighted

(b) weighted

Figure 3.10: 3.10a shows the CDF of the maximum five-minute usage sample relative to the resource request for the corresponding job. The thin line represents the largest five-minute sample within each job. The thick line discards outlier tasks within a job; for each job, it shows the 99th percentile of the largest five-minute usage measurements for each task. The top graphs shows memory usage; the bottom graphs show CPU usage. 3.10b shows the same CDFs as 3.10a, but with each job weighted by its per-task request times the number of task-days its tasks run.

high-percentile memory usage and average memory usage within each job, discussed in the next section.

CPU usage is not as constrained by the request as memory usage. Tasks used both much less and much more than CPU than they requested. To determine whether the CPU requests are usually too high or too low, it is useful to weight each job by the size of its per-task resource request multiplied by the number of task-days for which its tasks run. The difference between the request with this weighting is shown in Figure 3.10b; the resulting CDF reflects how much each unit of allocation is an over- or under-estimate of its usage. As shown in Figure 3.10b, with this weighting, the 'middle' job accurately estimates its CPU usage: about half the weight is in jobs for which 99% of the tasks have no CPU usage sample greater than the CPU request, and half in jobs with samples strictly greater.

3.3.4 Consistency of Usage

Resource usage is not the same between tasks in a job or over time within a job. As long as requests or predictions need to represent maximum actual usage, this variation will hamper the efficiency of any scheduler no matter how accurate the requests or predictions are. In this trace, this variation seems responsible for more of the inability of resource requests to predict usage than the requests being obviously poorly set by users. This is perhaps unsurprising for CPU utilization, where, perhaps, a program with a high maximum CPU usage would perform acceptably if capped at some smaller size, but unexpectedly it is also true for memory usage.

Conceivably, if users understood their resource usage well enough and the scheduler interface were rich enough, one might imagine that users would identify these outliers explicitly. The system from which the trace was extracted allows users to change their resource requests over time and across tasks in a job, but this feature is rarely used: jobs accounting for around 4% of memory allocated adjusted their CPU request in this fashion, and jobs accounting for another 3% updated their memory request.

Based on the usage measurements in the trace, the maximum of the usage samples are much larger than average usage — both between tasks in a job and within single long-running tasks over time. Even though memory usage is very stable over time, these outliers exists even for memory utilization: differences between the maximum usage of tasks within jobs account for about 20% of the total memory-hours requested (roughly the same as the apparent inaccuracies of each job's requests). Differences between the maximum usage for a task and its typical usage account for another 18%. (Because maximum memory usage can slightly exceed the memory allocation, the sum of these percentages can exceed the aggregate difference between requested and used memory-hours.)

These large gaps are rare; for example, most of the difference is the difference between the maximum and 99th percentile measurements (even for memory usage). We suspect that some of these outliers may be measurement error (e.g., page cache memory for common system files being accounted to only one program) or spurious (extra data being cached by the kernel that would be released at no performance cost under memory pressure). However,

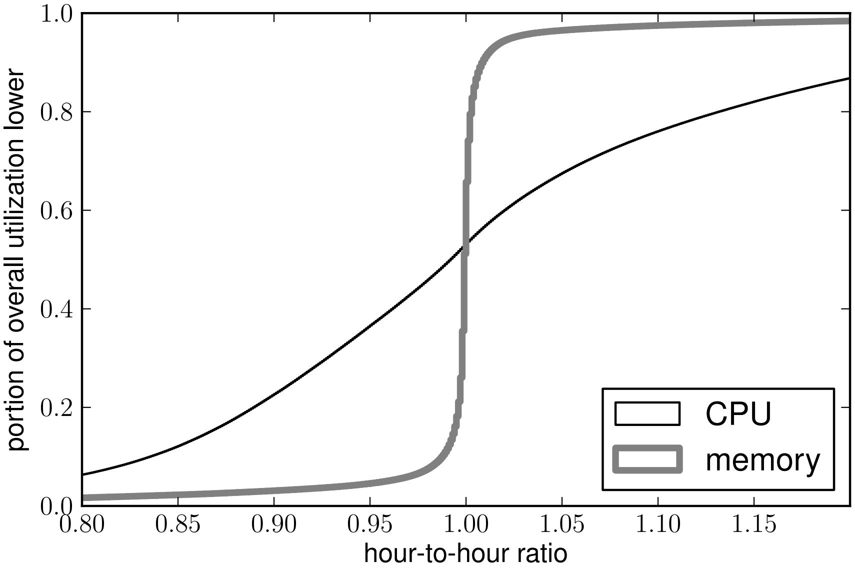


Figure 3.11: CDF of changes in average task utilization between two consecutive hours, weighted by task duration. Tasks which do not run in consecutive hours are excluded.

similar gaps exist for CPU usage where variability is less surprising. The effect does not appear to arise solely from startup effects, as even very long-running jobs experience these outliers.

Even though these gaps are rare, they have substantial implications for schedulers. For many applications, if any one task in a job can experience a usage spike then schedulers or users might (perhaps correctly) assume that all tasks will eventually. Schedulers that actually set aside resources for this spike will necessarily limit their system utilization they can achieve. Instead, to achieve high utilization in the face of usage spikes, schedulers should not set aside resources, but have resources that can, in the rare cases where necessary, be made available by, for example, stopping or migrating fine-grained or low-priority workload.

3.3.5 Stability of Actual Usage

When tasks run for several hours, their resource usage is generally stable, as can be seen in Figure 3.11. Task memory usage changes very little once most tasks are running. Memory usage data is based on physical memory usage, so this stability is not simply a consequence of measuring the available address space and not actual memory pressure.

Because there are many small tasks, a large relative change in CPU or memory usage of an individual task may not translate to large changes in the overall ability to fit new tasks on a machine. Taking advantage of this, a simple strategy for determining if tasks fit

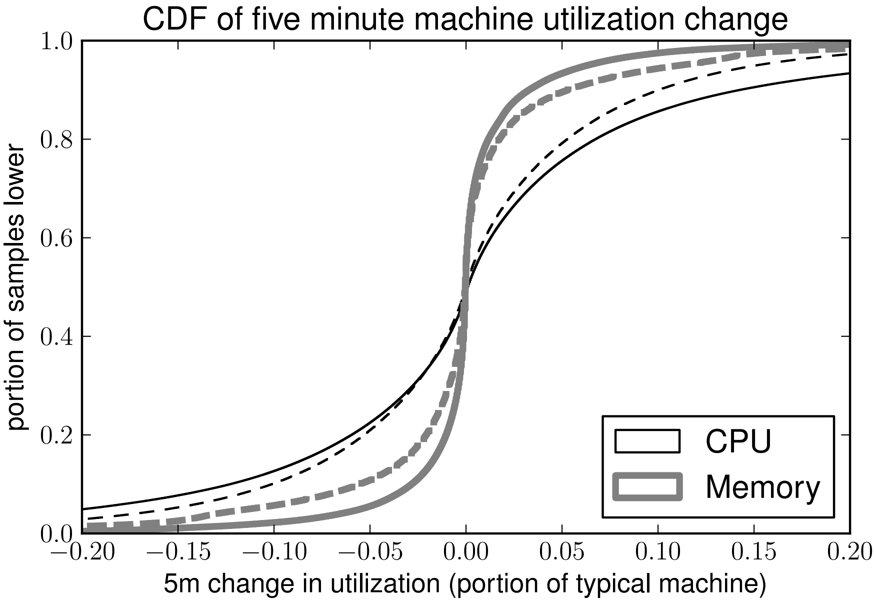


Figure 3.12: CDF of changes in average machine utilization between two consecutive five minute sampling periods. Solid lines exclude tasks which start or stop during one of the five minute sampling periods.

on a machine is to examine what resources are currently free on each machine and predict that those resources will remain free. I examined how well this would perform by examining how much machine usage changes. On timescales of minutes, this usually predicts machine utilization well as can be seen in Figure 3.12. Since most tasks only run for minutes, this amount of prediction is all that is likely required to place most tasks effectively. Longer running tasks are more problematic, but the relatively stable memory usage of long tasks as a whole suggests that the long-term memory usage of long-running tasks is stable. Given this, a scheduler should be able to colocate these tasks to efficiently use memory by monitoring resource usage over time and relocating tasks.

3.3.5.1 Repeated Jobs

A major problem with relying on the stability of usage is that it does not give much information about jobs that are not yet running. One hope is that most of the jobs could easily be identified as repeated jobs and historical information could be used. The trace providers provided some information about this in the form of a “logical job name” which the trace providers indicated would usually remain the same across multiple executions of a program.

The number of jobs per logical job name followed a Zipf-like distribution; most names appeared exactly once while the most common job name appeared more than 22k times.

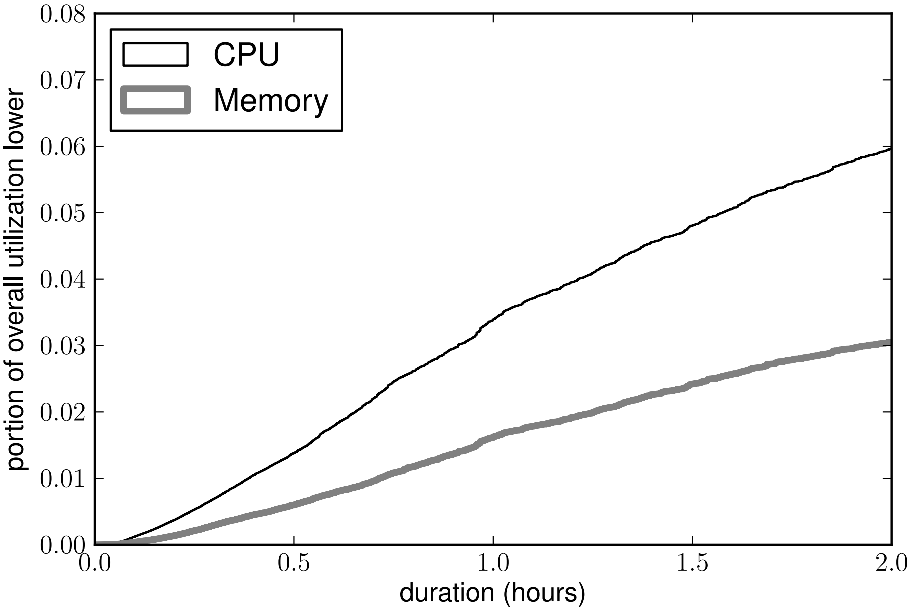


Figure 3.13: CDF of utilization by task duration. Note that tasks running for less than two hours account for less than 10% of utilization by any measure.

There were signs of jobs that were repeated on daily or weekly schedules, which deviated substantially form the Zipf-like distribution. Overall, however, the frequently repeated jobs naturally tended to be smaller ones and the long-tail of infrequently repeated jobs suggests that examining this “logical job name” marker will not predict much of the resource usage.

3.3.5.2 Short jobs

One apparent obstacle to forecasting resource availability from previous resource usage is the frequency with which tasks start and stop. Fortunately, though there are a large number of tasks starting and stopping, these short tasks do not contribute significantly to usage. This is why, as seen in Figure 3.12, ignoring the many tasks which start or stop within five minutes does not have a very large effect. Figure 3.13 indicates that jobs shorter than two hours account for less than 10% of the overall utilization (even though they represent more than 95% of the jobs). Hence, the scheduler may safely ignore short-running jobs when forecasting cluster utilization. Although these short-running jobs likely often represent repeatedly running jobs whose usage could be predicted from history, but the utility of such prediction would be small to simply looking at the history of currently-running jobs.

Even though most jobs are very short, it is not rare for users to run long jobs. 615 of the 925 users of the cluster submit at least one job which runs for more than a day, and 310 do so outside the gratis priorities.

3.4 Summary of Observations

My examination of Google cluster trace revealed many surprises. The workload is very challenging for schedulers; it has a very high number of scheduling events. There is a great variety of task sizes, and thus most tasks amount to little of the usage. However, the existence of large, long-lived tasks mean that the scheduler is likely to face bin-packing challenges that would require more careful scheduling mechanisms, particularly to efficiently use less flexible resources like memory.

From this variety of workload, and in particular, the mix of long-running services which traditionally have huge gaps between peak and mean usage and batch programs which can handle occasional eviction, I assumed that Google would achieve very high utilization. This was not the case. Based on the total allocation of the cluster averaging a bit below 100%, Google's scheduler largely respected the resource requests of their users. In spite of this, the scheduler's small attempts at overcommitting resources came at a great cost in terms of evictions of lower-priority tasks, perhaps the scheduler's best option for resources like memory where it cannot let a low priority task run temporarily slower like it can for compute resources. Still, the result was 'wasting' about 50% of the cluster's actual resources.

Looking at the information available to the scheduler to predict resource usage this was not a surprise. In general, even the notion that a single number indicates resource usage — certainly the most natural thing to ask for when one allocates the resources — did not match reality. Maximum empirical resource usage varied over time and among presumably interchangeable tasks within a job, at least when one was looking for high percentiles of the resource usage. Presumably, some of this effect might be opportunistic usage — such as caches that might go away at little performance cost if less memory was available, but we cannot determine this from the trace.

Ultimately, the best predictor of resource usage was past usage. Aggregate resource usage on each machine was largely stable, so it seemed the simplest way of obtaining a useful estimate for scheduling would be to assume free resources would be similar in the next few minutes.

Resource requests, in contrast, were remarkably uninformative. In some senses, they were the best information the scheduler in the trace had, since they represented a 'contract' with the user concerning their maximum resource usage. Thus, the Google scheduler apparently extensively relied on these requests. Not surprising given anecdotal stories about how clusters are used, resource requests appear to be largely manual and largely guessed. For memory, where tasks would not get substantially more than they requested, these guesses seemed to consistently overstate the task's requirements, though not by consistent amounts, perhaps because underestimates would fail. The apparent stability of memory usage suggested that these misestimates might be fixed if users had more information; though there was variability, these programs do not have inherently chaotic resource utilization.

3.5 Motivating a Memory Usage Tool

The Google cluster trace shows the promise of cluster consolidation — the ability to achieve high utilization and therefore efficiency by colocating diverse workloads. It also reveals some special challenges posed to schedulers in managing memory:

- Memory requests are typically overestimates. Relatively few tasks substantially underestimated their required memory — probably because tasks would often fail if they sufficiently exceeded their memory request.

- Overcommitting resources, especially memory, comes at a real cost in terms of wasted work. The cluster scheduler in this trace made only modest attempts to overcommit cluster resources and still disrupted lower priority workload significantly. A more aggressive scheduler would amplify these effects, giving lower priority workloads an environment equivalent to a very flaky machine. This difficulty suggests that although memory usage may be accurately predicted by past usage, taking advantage of these observations is difficult. Predictions based on history will inevitably sometimes be wrong, and strategies based on preemption for dealing with them are likely to make a cluster appear unreliable to lower-priority users.

- Users do not spend time tuning resource requests. Resource requests, including memory requests, appear to be dominated by round numbers rather than following a distribution that would suggest actual measurement of requirements. This suggests that efforts to improve memory requests without increasing user workloads need to start by assuming zero workload. One likely reason is that the increased costs — surely significant given the number of machines in this trace — of inaccurate requests are outweighed by the inconvenience of failure. To address this issue, as I develop performance models in the following chapters, I will focus on not penalizing users for tuning their memory requests and on minimizing the extra time and effort required to obtain good memory recommendations.

- Schedulers that support a diverse workload like Google's are limited in how they can make sense of application resource usage. For both compute and memory resources, there is uncertainty about whether resource utilization was required for correct performance, and there was often a potential that applications would take up excess resources or make do with more limited resources. Although these observations likely had a limited effect overall on my analysis of typical usage stability in Section 3.3.5, they are troubling when making predictions for individual tasks. Are there outliers in memory and compute usage because the usage is necessary for adequate performance or simply because space was available for some operating system cache or compute-power was available for some batch computation? This uncertainty motivates my collection of application-level knowledge about resource usage in future chapters over using operating system-level measurements.

Based on these observations, I developed a tool, presented in the next chapters, for recommending memory configurations for in-memory batch programs. These are programs where the memory usage is particularly important.

Chapter 4

Memory Modeling Overview

Memory configuration is a frequent source of frustration. This comes from gross errors — problems that trigger failures, or effective failures, due to insufficient resources. While these are large failures in terms of their effects, they are often caused by small errors. The failing configurations may be similar to working configurations, just crossing some working set size boundary. Traditional tuning approaches often are poor at combatting this — while they can certainly observe this boundary, their best approach to pinpointing it is to experience failures repeatedly. While the cases where a user will run the program so many times that this is a minor cost benefit most from tuning, they are also the easiest. Being able to try many, many configurations makes finding a good configuration among them simple. Instead, I serve users who do not wish to disrupt their workflow to find good configurations.

I avoid disrupting the user workflow in two ways. First, I prioritize making sure the recommendations do not cause the users to experience failures, by suggesting a clear upper-bound on the memory required to perform the computation effectively. Second, I try to make the instrumentation as non-invasive as possible, occurring as a user refines their data analysis task, rather than requiring them to dedicate training time to it. To serve these needs, I developed a tool, SLAMR (Safe Lightweight Analytics Memory Recommender), that takes measurements from runs the users perform normally, and gives them recommendations as they launch slightly revised versions of their analyses.

To enable this, I construct an explicit model of how memory is used in these systems. For the Apache Spark analytics stack I targeted, I divide this model into two submodels — one for the framework and one for the underlying language runtime. To design each of these models, I first examined what makes a configuration recommendation useful. It is not simply providing the maximum average performance. Users often face a harsh choice where a poor recommendation will result in memory exhaustion. Users are willing to waste memory to avoid this problem and would be unlikely to use a tool that would make these problems more frequent.

In addition to the complexity of providing configurations with low risks, there is complexity in the configurations themselves. The configuration is not as one number, but settings for many interacting components. In the case of Spark, these components include the

Spark framework itself, its underlying language runtime, and its underlying operating system. These interactions between these components make observing memory configurations more difficult: for example, excess memory in one component can substitute for insufficient memory in another.

Based on these challenges, I will outline the key heuristic that guides SLAMR: keeping the computation in memory. Relying on this heuristic makes it easy to meet the goal of consistently producing non-failing configurations without detailed profiling. It also simplifies modeling of cross-component interactions.

I can combine a Spark framework model — oriented around this heuristic — with a garbage collector model to enable 'continuous training' approach. I start from any working run and provide users better and better configurations. By assuming data must stay in memory, I can practically provide good recommendations with much less training information. And, naturally, these recommendations convincingly keep the user out of danger. Because my model makes rules about where data is kept instead of simply making statistical inferences from observed performance, users do not need to fear that is simply missing observations where the performance breaks down.

4.1 Roadmap

In this chapter, I review the motivations behind and overall structure of my configuration tool. First, I examine the configuration experience for users. Users face a sharp memory/performance tradeoff, combined with difficulty in understanding and configuring memory usage because of the layered nature of modern distributed systems.

Given these problems, I describe the design of SLAMR and how it tries to address them for the Apache Spark analytics stack. Figure 4.1 provides an overview of the components of SLAMR, along with where each component is covered in detail. In this chapter, I will describe the motivations that are common to the two models described in future chapters — one for the Spark framework itself and one for its underlying language runtime. Both of these are designed based on the sharp memory/performance tradeoff users face and around avoiding issues from the interaction between configuration at different layers, but in different ways.

In the case of the framework model, I focus on keeping the computation entirely in memory to avoid falling behind the curve. This focus also ensures that my observations are not dependent on the behavior of other layers of the system. For the language runtime model, I am not as easily able to avoid interference with my measurements from the behavior of other layers of the system or to identify a concrete point where everything is in memory. I will outline how I imagine users combining these two models, and how I match the submodel's overhead estimates to find a configuration just beyond a sharp memory/performance tradeoff. This will serve to provide context for my description of these models in the next two chapters. Later, in Chapter 7, I will return in more depth to the problem of combining the two submodels and evaluate the combined results.

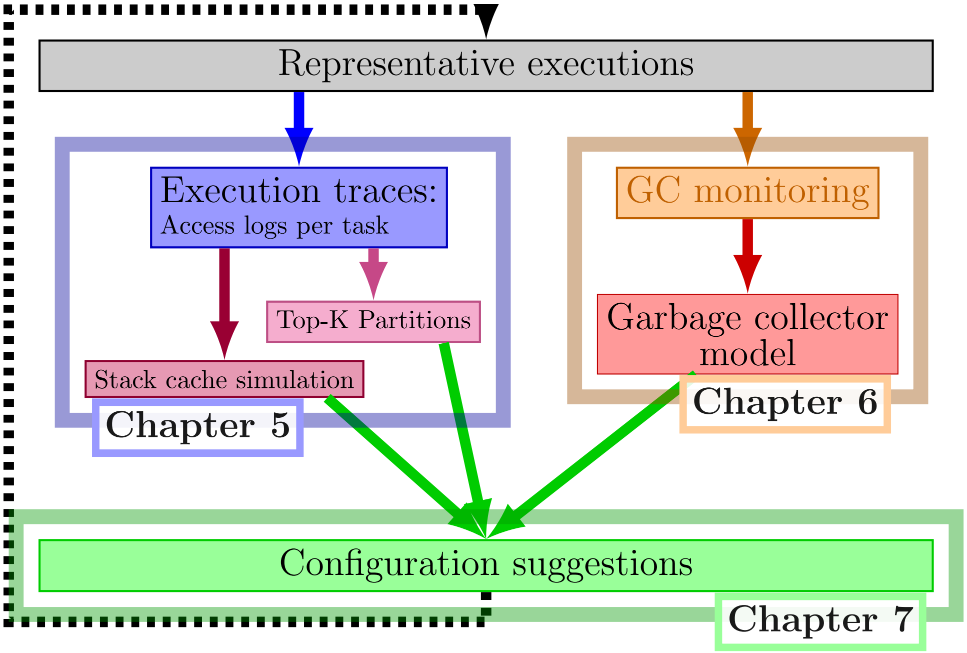


Figure 4.1: Block diagram of SLAMR's pipeline of instrumentation to configurations, along with the chapters in which each part is described.

4.2 Targeted Configurations

Generally, there is a tradeoff curve for memory allocations as shown in Figure 4.2: Small allocations fail entirely — or the program is so sluggish it may as well have failed. Then, there is often a region of steep decline in runtime where the program is thrashing and relieving a little memory pressure can drastically relieve this thrashing. After exiting this thrashing region, the program's working set size is satisfied, so adding memory provides little improvement to program's runtime. How close this qualitative view of memory allocation tradeoffs is to actual performance will vary, as shown in my evaluations in later chapters, but generally, it will guide my configuration design.

My goal is to help users avoid the surprises. Rather than optimizing the expected performance, the user should have reasonable confidence that their configurations are beyond the thrashing region and in the region of diminishing returns. Preferably, the configuration will not be very far from the transition point; it will be just beyond the 'knee'. Based on my observations from the Google trace, I believe that users often stay well beyond the knee — by allocating much more memory than they need to. Since these configurations provide roughly equivalent performance, my goal is to provide the same experience at lower cost.

To do this without disrupting the user workflow to collect a large amount of data, I will make conservative recommendations. How conservative will depend on the program in question — there will be some programs for which we will allocate extra memory to avoid extra I/O and computation that is, not, in fact expensive. But the alternative would to

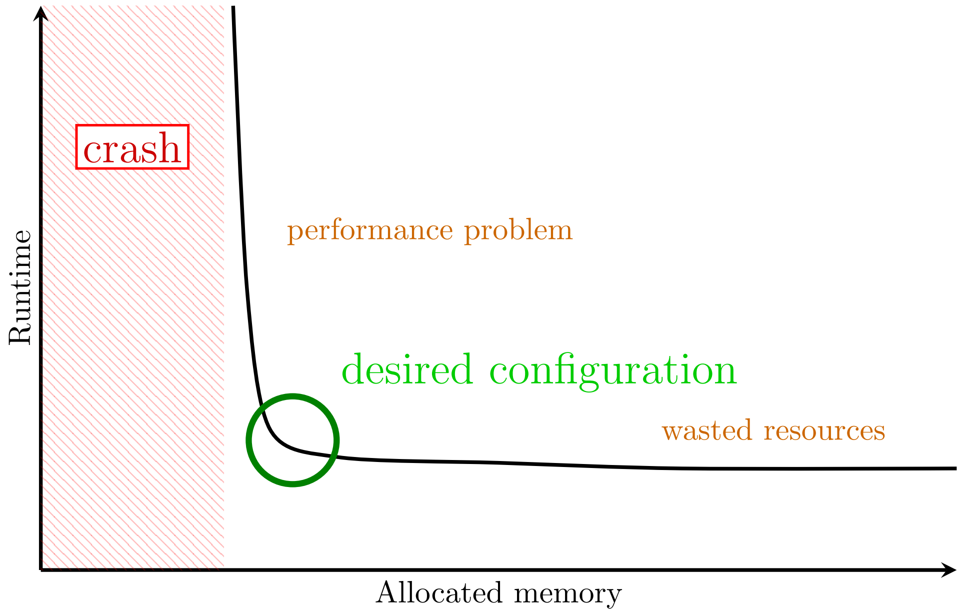


Figure 4.2: The conceptual tradeoffs between memory allocation and runtime faced by users.

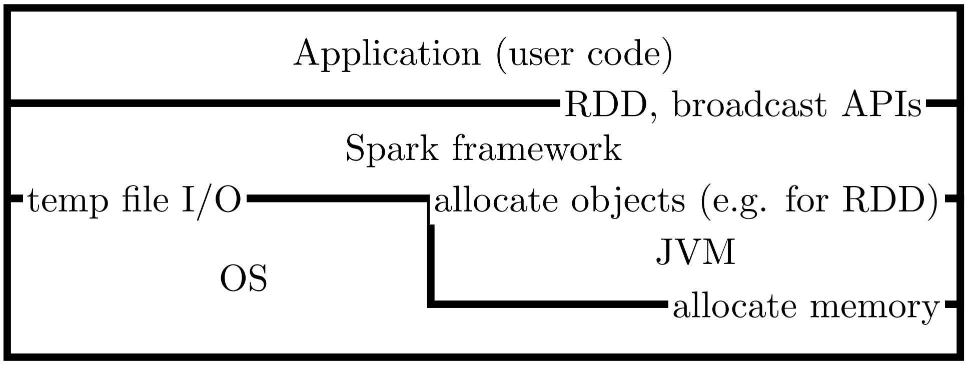


Figure 4.3: The various layers through which a Spark program uses memory. A component sits on top of the other components it uses memory resources through. Notations at boundaries represent the kind of memory consumption that occurs.

collect more data to demonstrate that the alternative was feasible.

4.3 Multiple Components

Figure 4.2 suggests that a user who configures a program for an analytics framework simply chooses a number — how much memory to allocate. This is often how configuration options are presented, but high quality configuration is not so simple. In-memory analytics systems like Spark are composed of multiple, interacting components.

Each of these many components generally manages its memory separately. In many cases, the size allocated to a particular use is fixed at configuration time, and models must configure each size separately. In the cases where users easily configured Spark by setting a single number, they were simply using a default portion of the total for each component; this default may not be optimal, and any memory usage modeling must understand this division of memory. Some memory regions are configured implicitly — they will use all the space available that is not used by other regions. These must still be configured, but implicitly, by imposing constraints on all the configurations.

Broadly, in Spark, there are three components SLAMR is concerned with: the Spark framework itself, the language runtime, and the underlying operating system. The layering and relevant points of interaction for these components are shown in Figure 4.3. There are three parts whose memory usage I model: the Spark framework based on application-specified datasets, the language runtime based on the framework and application's memory allocations, and the operating system based on cached temporary files produced by Spark.

4.3.1 Framework

Analytics frameworks like Spark are primarily responsible for managing data and data movement. As depicted in Figure 4.3, a user specifies RDDs (datasets), including the transformations that derive new RDDs from others, and, sometimes, data to broadcast to all tasks. The framework implements policies that translate these requests into memory usage requested from the language runtime in the form of objects or the operating system local in the form of temporary files. To decide when to use memory, frameworks like Spark often track the total amount of space used and available. Some tracking is required even in data analysis systems which do not attempt to keep data in memory persistently; for example, Hadoop MapReduce manages sorting buffers in this way.

In Spark, this tracking primarily involves the management of two regions: 'storage', representing the memory managed by a component called the block manager, and 'execution', primarily representing memory used during aggregation and many-to-many shuffles. The block manager is responsible for most sources of persistent memory usage — cached and broadcasted data requested by users. The block manager is also responsible for any data temporary stored in the local filesystem. This occurs both when Spark believes its memory is exhausted and to support many-to-many data transfers. Execution memory consists primarily of memory used by Spark's out-of-core sorting support. I describe how I instrument and model both these kinds of memory usage in Chapter 5.

4.3.2 Language Runtime

Like much modern systems software, Spark and Spark applications are written using automatic memory management. Most of Spark's in-memory storage consists of ordinary objects allocated in this managed language. These allocations correspond both to the datasets that users requested Spark store and to temporary data Spark produces during the transformation a user specifies to produce an RDD. From the perspective of the language runtime, the large memory regions managed by Spark are no different from temporary objects allocated to support computations except in their lifetimes and sizes. Spark's efforts to improve the performance of this automatic memory management have largely taken the form of using object layouts that are more friendly for automatic memory management, perhaps by not using managed memory at all.

Although managed memory makes programming applications easier, it makes understanding memory requirements more challenging. Frameworks do not have precise visibility into its memory usage — even of the regions it purportedly manages — and the underlying language runtime has overheads. Most notably, it will not behave correctly without space allocated for unreachable objects to accumulate before they are actually freed by a garbage collection pass. Since the garbage collector lacks ways of dealing with insufficient memory like dropping cached data or placing data on the local filesystem, its response to memory exhaustion is the least user-friendly — it triggers out-of-memory errors or loops of expensive garbage collection passes. Because of this, it is a major source of frustration for users.

4.3.3 OS Caches

Frameworks like Spark makes some use of the local filesystem. This may occur when data eligible to be stored in the language runtime memory does not fit, but frameworks often use the local filesystem even when memory is plentiful. The most notable case for Spark is data to be sent to other workers on the local filesystem, as described in Section 5.1.4. As a result, Spark performance might implicitly depend on how effective the operating system is at caching those local filesystem accesses. In the best case — and presumably the case for the best-looking Spark benchmarks — the accesses either will be served from cache in DRAM or will involve large sequential read and writes that are fast enough compared to the corresponding computation to hide I/O costs.

4.4 Interdependencies

Multiple layers introduce complexity not only in configuring memory requirements but also in measuring memory requirements. How much memory is 'free' and might be deallocated has different answers depending on the layer of abstraction one uses. Memory used for a stale application-level cache may be logically free, but as far as the operating system and language runtime is concerned, it is used and cannot be reclaimed except at great cost. Similarly, files cached by the operating system may be occupied and necessary for performance, but as far as the application is concerned that memory is available.

These effects make simple approaches to memory modeling — based on observing the 'high water mark' — ineffective. Although there are efforts to unify memory management between the various layers, overall, there is no one place which gives a complete picture. Counting what memory was used at some layer and the corresponding performance will tell an incomplete story, missing memory that could be made useful if more was available or overcounting extra allocations that were not helpful. Regardless of whether memory usage at some layer is helpful, it is likely to affect the behavior and requirements on layers.

One way these effects manifest is that free space in the operating system's filesystem caches may dampen the effect of an analytics framework having insufficient. In Spark's case, unable to cache data, it will either recompute the data — paying the cost of reading it from some filesystem again — or store the data on the local filesystem and read it again. With sufficient filesystem caches, this impact of this may be negligible, but using that observation to decrease the total memory allocation would expose the actual cost of I/O. This phenomenon can make it difficult to empirically observe memory requirements — to have confidence that some memory/time tradeoff is acceptable, one would need to take extra effort to ensure that one observed its maximum extent.

Spark's self-management of memory also imposes difficult to predict demands on the language runtime. The pressure that a framework like Spark places on the language runtime will depend greatly on how much space it thinks it has available to cache data. Spark will tend to fill the available space with cached data, which the language runtime must retain and scan as it manages memory. But the overhead may not disappear as Spark caches less data. The alternative to reading data from cache may be recomputing it or deserializing it from (possibly cached) persistent storage. This process may require the creation of temporary objects creating a different kind of pressure on the language runtime's garbage collector. Thus, lowering the memory kept by Spark could easily increase the garbage collection overhead.

Generally, between all of these layers, each type of memory can partially substitute for the others. Either there is a direct substitution — like reading from filesystem caches instead of from process memory — or an indirect exchange of performance — like slightly increased performance from additional page cache space available after decreasing language runtime memory.

4.5 Criteria for Choosing Configurations

When making configuration recommendations, my primary goal is to provide a *upper-bound* on the required memory. Accuracy is not the first priority; I assume that users prefer confidence that the result is never going to result in memory exhaustion or thrashing over obtaining the time-optimal configuration. To do this, the central heuristic I will rely is keeping the computation essentially in memory — preserving the user's intent in specifying what to compute in memory and ensuring that memory exhaustion will not occur.

In the next two chapters, I will construct models of how a data analysis framework, namely Spark, uses memory and how an underlying language runtime uses memory. If in constructing these models, I relied on simple black-box measurements of performance and resource usage, I would risk being mislead by memory in one layer substituting for another. By aiming for keeping computations in memory, I sidestep these issues. For example, I do not worry about implicitly counting on reading from filesystem caches instead of spinning disks; instead, I assume that caches are required.

By addressing memory/time tradeoffs in that way, I avoid sensitivity to those cross-layer effects where one type of memory substitutes for another. For example, by avoiding the tradeoff entirely, I cannot accidentally mismeasure a performance tradeoff by measuring the performance of reading from filesystem caches instead of from a spinning disk.

I want predictions to account for possible but perhaps unlikely circumstances. It is these circumstances that will make a user ignore configuration recommendations and add some extra space 'just in case'. For example, in Spark, the order of computation might change based on timing differences or because of machine failures. Rather than only observing or estimating the typical order, I produce a recommendation which should be acceptable in spite of reordering. Similarly, for language runtime constraints, I want to somehow reflect any predicted framework demands on memory, even if I never observe them in the garbage collector traces. By taking this approach, I can give users confidence in the safety of my recommendations, so they do not feel the need to add an additional 'safety margin' to them.

4.5.1 Computing in Memory

To choose good configurations quickly, I make a heuristic assumption: that users want to keep their computation in memory. This is the original purpose of Spark, and through Spark's API, users and their libraries state their assumptions about what should stay in memory in their programs. This is clearly a compromise: Spark has real support for doing computations out-of-memory.

Spark can make CPU time/memory usage tradeoffs. When insufficient memory is available, Spark and similar frameworks fall back to other mechanisms — using persistent storage or extra computation. These mechanisms come from Spark's roots: one of the main advantages of MapReduce [25] at the time was efficiently performing disk-based computations. Spark's internals mirror its design, and Spark's performance in this mode has improved as, for example, it has been tuned for the sort benchmark [76]. Nevertheless, the nature of the tradeoff is application-dependent as our evaluation illustrate. Some programs which take many times longer when Spark performs its recomputation-based workaround for even a small amount of space, and some which experience barely measurable differences in performance.

While it possible to model these tradeoffs, this would likely require obtaining many measurements, especially to control for variation in the costs of I/O, serialization, filesystem caching, etc. I do not want to interfere with the user workflow by requiring the user to run programs many times. I especially do not want to require users to run their programs in configurations that may run the program very slowly, which is likely if disk I/O is forced. Instead, I aim for predictability: I do not second-guess when users have requested that Spark do computations in memory.

4.5.2 Time/space tradeoffs

Although generally, I choose the memory side of time/memory tradeoffs, I do need to consider some time/space tradeoffs. Keeping 'everything in memory' is not directly applicable to garbage collectors. Unless there is enough memory to store all allocations made during the entire long-lived program, some CPU overhead is necessary. So, instead, I obtain a lower bound on that overhead from a performance model. Using this lower bound, my goal is to avoid thrashing behavior where garbage collectors are triggered an excessive number of times. The performance tradeoff experienced by garbage collectors tends to exhibit the 'cliff' followed by marginal improvements shown in Figure 4.2, so the region of high overheads has a similar intuitive effect to keeping 'everything in memory'.

Based on the charging model of cloud computing platforms like Amazon EC2, I estimate that user costs are proportional to the amount of memory allocated times the time it is used. In cloud computing environments, users pay for virtual machines by the time unit. For memory-constrained users, these prices are determined largely by their memory capacity. My recommended configuration will aims to minimize the cost. Since I use upper-bounds on runtime, this will make sure that configurations where thrashing, and its high time overhead, will be considered too costly and be avoided. In addition to considering an estimate of the total cost, I will avoid suggesting very large configurations to avoid garbage collector overhead too aggressively by assuming there is some overhead to increasing the number of workers rather than perfect parallel scaling.

4.6 Training Procedure

The models in the next chapters rely on observing a program's behavior from example runs. Not just any execution of the program will do. Most importantly, the program must actually complete successfully. An execution that does not complete due to memory exhaustion will only indicate that not enough memory was available. Thus, a user of my system needs some way to get the program to work in the first place.

The typical mechanism users use — and that I use for my evaluations — is to start by looking at the size of input data. Since doing in-memory analytics requires keeping the data in memory and having extra additional space for derived temporary data, a small multiple of the input data size is a likely choice. Once there is a complete execution, there is still some question about whether the behavior I observe for one execution is consistent, or whether it will be relevant to small range of possible configurations. I construct my models of framework and garbage collector behavior to limit the bias created by observing executions with 'too little' or 'too much' memory.

This is not entirely effective for my garbage collector model. There is a limited range of configurations that it can predict from a given observed run. As I will show in Chapter 6, I can infer the behavior across a wide range from a single observations. But there are notable cases where my observations will be substantially wrong: when the 'correct' configuration is far away (by around an order of magnitude) from the observed configuration or the observed configuration lacks information due to having much more memory than required. In both cases, I construct my model to suggest a substantially better configuration, but not one particularly close to the likely final result. After observing the garbage collection statistics from the refined configuration, my model can more accurately predict the results in the interesting region. So, after some more iterations (depending on the degree of the initial error), the configuration will converge, correcting for the initial lack of data.

4.6.1 Framework Model Sensitivity

Because of the strong relationship between framework observations and the semantics of the data processing, the framework model I develop will generally reflect relevant configurations. I can separate the memory accesses necessary to compute the results from any extra memory accesses my model ignores, like recomputation. In this way, my framework predictions are largely independent of the observed executions.

One remaining source of variation is that Spark's partitioning and exact order of execution may be dependent on the available memory. I will avoid dependencies on the order of execution by trying to accommodate the worst-case execution order. This is likely necessary to reason about what might happen when failure recovery causes some computations to be performed again in a different order anyways. For partition sizes, Spark chooses them mostly based on the input and not the available partition size. For cases where they can vary because of the partitioning, I can estimate the effects of repartitioning as described in Section 5.6.5. Generally, changes in partitioning only have a significant effect if the partitions are very large, which is usually not the best choice — users would be better off increasing the partition count regardless of the number of cores in use.

4.6.2 Garbage Collector Model Sensitivity

My model of garbage collector behavior is more sensitive; it relies more on the workload actually placed on the language runtime. This workload may change based on how the framework computations are distributed and, of course, how much framework stores in memory. I do make adjustments based on framework information — such as knowing the maximum amount of data Spark tracks that may be in memory. But, overall, my tool lacks the information to separate workload placed on the garbage collector by recomputation, storage of extra data in the cache, etc. that will not exist in the targeted configuration.

Conceptually, it might be possible to identify recomputation-related demands on the garbage collector, but since Spark, like essentially all data analysis frameworks is multi-threaded, recomputation will overlap with 'normal' computation. In addition, Spark streams computed values to the next phase of computation — rather than store all of the values in a partition first — so recomputation may be tightly interleaved with other computation in the same thread. This design aids in support of out-of-core computations, so it is likely to be common among analytics frameworks.

4.6.3 Combined Training

Ultimately, I want to enable iterative configuration refinement. From the first working run, the user obtains configuration guess, but then future runs are still made with my low-overhead instrumentation enabled. Once framework-level configuration is chosen, subsequent runs should represent the true garbage collector workload. However, in addition to being dependent on load on the garbage collector, my observations of garbage collector activity are only valid for a range of sizes around the observed configuration. Fortunately, empirically, this range is large, and, again, I can rely on the best estimate from within the range observed to allow the next run to observe the relevant region. My instrumentation is lightweight enough that performing more instrumented runs is not a major issue — the loss of performance should be small compared to the reduced cost of the configuration.

4.7 Conclusion

Motivated by the challenges faced by users in configuring memory leads to a different type of configuration optimization system. The goal is not the optimum configuration above all else. But, instead, I recognize that for the sort iterative computations typical of in-memory analytics, users are often outside the region where 'swapping' is a reasonable performance option. Thus, my tool SLAMR is designed to stay out of this bad performance space, quickly, and confidently, rather than spend more data collection effort to navigating time/space tradeoffs. This avoids the problem of needing to observe risky time/space tradeoffs — tradeoffs often 'hidden' by caches at other layers of the system.

SLAMR is composed of two pieces. First, one that observes activity at the framework layer, avoiding dependencies on the amount available memory, as described in Chapter 5. This model identifies a configuration that is definitely beyond the 'knee' of the user's tradeoff curve and usually just beyond it — where all cacheable data is cached. Separately, it measures the load placed on the language runtime's garbage collector by the framework as described in Chapter 6. Rather than being completely independent, garbage collector predictions are constrained using framework memory requirements.

The result is a unified 'memory advisor'. The user runs their Spark program with low-overhead instrumentation always on. As they refine their program, they get suggested configurations, allowing them to refine their configuration — saving space or rescuing them from mysteriously low performance. While these configurations will be more conservative, the user will have dramatically improved on their guesses without facing the fear of going too low and waiting forever.

Chapter 5

Measuring Memory Usage in Spark

Analytics frameworks provide numerous abstractions to help developers and library writers organize their computations. Since the framework manages the distribution of computations and their fault tolerance, effectively all computation must be mediated by these abstractions if an application uses the framework. They provide the mechanisms to move the computation between machines, to store data persistently during the computation, etc. The tools analytics frameworks provide to do this are much of what motivates their use — using their implementations enables distribution and fault tolerance.

Having visibility into the entire computations, these same abstractions can also provide diagnostics about resource requirements. In this chapter, I will describe how I produce such diagnostics using the example of Apache Spark [99]. Combined with a model of language runtime of the analytics stack upon which Spark depends, described in the following chapter, this will be how my tool SLAMR produces a configuration recommendation for the user.

To estimate framework memory requirements, SLAMR records how the framework's data processing abstractions are used in a representative execution of an analytics program. This effectively produces a summary of the program. Using the framework's abstractions makes this summary simpler, naturally grouping related data and computations. It also preserves semantic information in sufficient detail that one can infer behavior with different resource availability.

Analytics frameworks commonly make decisions about whether to keep data in memory or fallback to using disk or recomputation. These decisions are based on available resources themselves. Thus, simply examining aggregate memory resources that are used will often reflect the framework's decisions about what to keep in memory more than actual memory requirements. To make predictions, I focus on collecting information that predicts framework decisions rather than reflecting them.

A framework's decisions about whether to keep data in memory also create their own configuration complexities. When frameworks need to decide how much space to use for temporary data over persistent storage, they usually rely on configurations or defaults. Regardless of whether one is changing these configuration parameters, a configuration recommendation must not just ensure that there is enough space, but that the framework decisions

Memory type	Framework config.	OS config.	Persistent?	Distributed?
RDD storage	'storage'	JVM heap	Y	Y
RDDs being computed	'unroll', 'storage'	JVM heap	N	N
broadcasts	'storage'	JVM heap	Y	N
Aggregation buffers	'execution'/'shuffle'	JVM heap	N	N
Map outputs	(none)	page cache	Y	N

Table 5.1: Types of memory usage in Spark program, the corresponding configuration parameters that must include enough space for this memory region, whether this memory is persistent between tasks or temporary for a single task, and whether this type of memory is split across nodes or must be stored on each node.

will use that space.

Especially in the academic literature, frameworks typically advertise an overarching abstraction — like Apache Spark's 'resilient distributed datasets'. It would be reasonable, then, to guess that one would only need to account for how this abstraction would correspond to memory usage. But, though this abstraction may dominate the resource usage of typical programs, there are other, less common ways for programs to use memory. Programs where these sources of memory usage are important are precisely the sort of non-intuitive behavior, memory usage not coming from obvious sources, that configuration recommendations should deal with.

With these concerns in mind, in this chapter, I construct a model of memory requirements for a Spark program. I begin by examining the various ways in which Spark's abstractions translate into memory usage. For each of these kinds of memory usage, I identify what configuration parameters control the framework's decisions that I will need to configure. I then describe how I modify the framework's abstractions to record the information I need to identify the load a program puts on each of these types of storage. I then review strategies, depending on the kind of memory usage, for translating records of usage into configuration requirements, handling cache-like memory regions and the distribution of data. Finally, I perform a sensitivity analysis to show how effective these configurations are at satisfying a variety of program's memory requirements.

These recommendations about framework memory requirements will be combined by SLAMR with a model of garbage collector requirements and the user's available machine configurations, as described in later chapters of this dissertation. The predictions of the model permit SLAMR to confidently satisfy framework memory requirements for configurations very different from the ones on which measurements are taken.

5.1 Spark Memory Taxonomy

I briefly describe Spark's programming and execution model in Section 2.2.3. As Spark executes a program in this way, it has a variety of ways in which it translates an application's use of its abstractions into memory usage, as shown in Table 5.1. These various types of memory vary between persistent data distributed across a cluster and temporary memory demands. This includes memory demands that Spark manages and does fallback for, like its important caching, and ones that fallen by the wayside such as map outputs that are stored in the workers' local filesystems. To make effective one must account for all significant regions of memory usage. Fortunately, for applications that make effective or natural use of analytics framework these are limited by the interfaces an that framework provides to process data.

Spark is built on having a single logical representation of persistent data — what it calls resilient distributed datasets (RDDs). These are the abstraction over which Spark's users manipulate datasets that are distributed across many worker nodes and through whose lineage Spark achieves its fault tolerance properties. Thus, understanding the memory usage of a Spark program is substantially about understanding how large and how active the RDDs are. But, there are other sources of memory requirements managed by Spark. Since they are sometimes (but not usually) significant, making conservative estimates requires taking them into account.

For the purpose of counting memory requirements, one can divide Spark's memory usage into two broad categories: memory that persists between tasks, most notably RDDs, and memory used while computing a task, most notably related to partition sizes. Each of these categories of memory usage will be divided and shared across multiple, separately configured regions. To produce configurations, my tool SLAMR aggregates several measurements of each kind based on how they map to these memory regions.

Memory that persists between tasks in Spark is largely managed like a cache. When space is available, Spark or the underlying OS keeps data around just in case a computation requires it. As space is exhausted, some less recently used item will be discarded. If that item is required again, it is reconstructed — either by computing it again or reading it from slower storage. SLAMR's recommendation for this memory is focused on avoiding that 'cache miss' cost.

Memory used to compute a task is more straightforward to measure, but a more common source of surprise. This memory is generally proportional to the amount of data processed by each task. In most programs I evaluate, these execution requirements are relatively modest. The most likely reasons for these requirements to become large is choosing an atypically small number of partitions or having very unbalanced partitions. As part of my goal of making conservative recommendations, I use measurements of the distribution of partition sizes to account for the worst possible case of unbalanced requirements. Thus, recommendations will generally be constructed by assuming the 'bad luck' of a single worker will get all the largest partitions at once.

For most programs I examined, the effects of this temporary storage was minor, less than 10% of the memory of the program. Where it is most likely to become a problem is when the program should be configured to use a different number of partitions. In this case, one can estimate the savings from changing the partitioning without sacrificing the conservativeness of the estimate as I will discuss in Section 5.5. There are programs where large partitions remain even after increasing the number of partitions arbitrarily — likely because there is some large group which cannot be split. Naturally, I did not find these among example programs I examined because such programs will inherently scale poorly.

5.1.1 Cached RDDs

RDDs are Spark's explicit representation of large datasets. Like most MapReduce-derived tools, these datasets are distributed largely invisibly to users. Users manipulate the data by applying operations to produce new distributed datasets rather than modifying data in place. Along with specifying these transformations, Spark allows users to mark which datasets that they believe are important enough to allow to persist in memory. Other datasets are only computed when needed, then discarded. It is these datasets that account for the bulk of Spark's memory requirements.

When a user marks an RDD to be persisted, Spark computes its partitions differently. By default, Spark only computes on demand, with each element passed to the next step of the pipeline as soon it is computed (except where group-by-like operation are required). For persisted RDDs, Spark checks for a cached version — including on disk or remote workers. If it does not find one, it starts computing the partition into a local array. If there is enough space available, potentially after evicting old partitions, this array will remain cached after the computation completes.

5.1.1.1 Accounting for Cache Space During Computation

In early versions of Spark, cacheable partitions were stored in an in-memory array unconditionally, and only then were evictions decisions made. At this point, memory would be required for both any dependencies of the computation (if cached) and the newly cached partition. Then, even if the newly computed partition would not be persisted, it would remain in memory until it was consumed by its downstream computation. Only after that, would memory usage consistently return to the amount allocated for caching.

In later versions of Spark, support for 'unrolling' cacheable RDD computations was added. In this scheme, Spark accounts for the space used by an RDD as it is computed. It computes a cacheable RDD into an array, like before, but it samples the sizes of the elements as it is being computed to maintain a running estimate of its total size. As it updates this estimate, Spark starts evicting RDDs from its local cache. If this eviction would not free up enough space, then Spark aborts the computation. In this case, the already-computed values are consumed from the local array, but the remainder of the partition is computed on demand.

Although this policy ensures that memory used for cached RDDs stays within the allocated space, it still has negative effects with large partitions. Without restrictions, a large partition would evict all other partitions on a worker as it was computed. But this partition ultimately fail to be persisted because it was too big. To avoid this effect, Spark limits the amount of space it will evict for a in-process partition. This results in different non-intuitive behavior: A user would likely assume that they can have one partition per worker core, with the total size of these partitions and their dependencies being somewhat less than the amount of dedicated storage. Instead, the maximum size of partitions that can be computed simultaneously is effectively limited by the amount of space dedicated to unrolling, divided by each of the cores computing these partitions. With the default setting as of Spark 1.4, if the partition and its dependency have equal sizes, this behavior is likely, depending on timing.

5.1.1.2 Eviction Policy

Spark's eviction policy for cached RDDs is approximately least recently used (LRU). There are, however, several notable ways in which Spark diverts from a pure least recently used replacement policy. One is that caching decisions are local even though the RDD cache is global. Partitions are only cached on workers where they are used, and Spark never makes explicit attempts to move data between nodes to correct for load misbalance

Another divergence from the least recently used policy is to accommodate the common practice of scanning a whole RDDs over and over again. This can result in worst-case behavior for LRU: each newly scanned partition would have been the least recently accessed and so already evicted. To avoid this problem, Spark does not evict an RDD for another RDD from the same partition, which results in a smoother decline in performance when there is insufficient memory and the working set is contained in a single RDD.

Spark also permits users to take greater control of the eviction policy by manually evicting RDDs. Spark calls this 'unpersisting'. While usually this is just an optimization, it is notably necessary to prevent disk space exhaustion if an RDD is marked to be persisted to disk if there is insufficient memory.

5.1.2 Broadcast Data

Spark includes a feature to distribute some piece of data to potentially every task. These are called 'broadcasts'. Normally, items broadcast are typically quite small. The most common use of broadcasts in Spark programs is internally to distribute small amounts of data to all tasks within a stage, such as user-defined functions common to many tasks, the locations of input data, etc. These are typically of small size. But the broadcast interface is also exposed externally to Spark's application developers.

The most externally prominent use case for broadcasts is for some iterative machine-learning tasks. There, a common task is distributed a current parameter set to be applied to all the cached data. In this case, the parameter set is likely to be of negligible size, but it could be quite large for some machine learning algorithms. Nevertheless, this use of large broadcasts does not fundamentally prevent scaling, since the parameter set — generally the size of the output model — is likely to have a fixed maximum size. As long as that fixed maximum size does not exceed the physical memory of each worker, the computation can use as many workers as wanted.

Some Spark programs use broadcasts in ways that do not scale so well. For example, Spark includes as a tutorial program, such an example, a *naive* implementation of Alternating Least Squares (ALS) [101]. ALS is a matrix completion algorithm; it takes a matrix M and finds a fit to those entries $XY^T \approx M$ where X and Y are 'skinny' matrices. Spark's naive implementation uses the framework's abstractions to distribute the computation but not the storage. It creates a broadcast for the each iteration's estimates for X and Y and for the actual values of M, requiring the framework to replicate those matrices on each worker. Such a program is not what one would recommend using, but nevertheless, we have a goal of making correct memory recommendations for programs that have made this decision.

Spark broadcasts are stored persistently by the driver program. For correctness, it is necessary for a broadcast to remain available unless it can be proved that an RDD that references the broadcast will not be recomputed. A reference to a broadcast can be obtained as a JVM reference and passed in a closure for any task. With this design, Spark would need to do introspection on the code supplied to tasks to identify which broadcasts are still available or to identify which broadcasts will be required before running a task. Given this, for Spark to reliably remove broadcasts, it requires something like the unpersist mechanism for RDDs, where users identify what broadcasts are still required.

When broadcasts are required by a worker, they are distributed to the worker programs via one of several possible mechanisms. The default is a BitTorrent-like mechanism, where workers will exchange chunks of data with themselves. After the broadcast is received by a node, it is stored in locally in the cache — the same cache used for RDD partitions, and subject to replacement under the same least-recently used policy. They, however, are not subject to 'unrolling' behavior of RDD partitions since they are usually not lists that can be partially computed. Thus, like the older behavior for RDD partitions, a large broadcast will consistently cause out-of-memory errors.

5.1.3 Aggregation Data Structures

Spark supports group-by and sort operations using a strategy similar that of to Hadoop MapReduce. Like MapReduce, it implements a many-to-many 'shuffle' communication pattern from a set of 'map' tasks to a set of 'reduce' tasks. Values are assigned to a particular reduce task based on keys supplied by map tasks. To do this, at each mapper, values are partitioned by the destination shuffle task, which can be computed by each value's key. The mapper tasks group values and their keys by their destination reducer, consolidating multiple values with the same key and sometimes sorting each group by key. The user (of the mostly internal Spark shuffle API) supplies an optional *combiner* function, which specifies how to consolidates multiple values for a particular key as the values are grouped by key at each of the mappers. Finally, these values are written to the local filesystem in contiguous runs for each shuffle task to request remotely. The shuffle task performs remote reads of each of these runs. If requested, it groups these values by key, applying the 'combiner' function as it does. Generally, the per-task grouping operations will use storage local to the shuffle task rather than relying on the remote reads to be synchronized enough to directly do an N-way merge.

Spark operations that use this support for aggregation can be an important source of memory pressure and memory/time tradeoffs. As Spark does these aggregation tasks for task, it can create in-memory data structures containing all the data in a partition, even if the partition will not be persistently cached. Like for cached datasets, Spark maintains a running estimate of the sizes of this data structure. When the estimated size would exceed available memory, by default, Spark writes the current state of this data structure to disk and clears the in-memory version. Then, when all input has been consumed, Spark does an N-way merge of the last in-memory data structure and each of the versions written to disk.

The costs of doing the out-of-memory operation are quite variable. A major source of overhead in this operation is not the disk I/O, but serialization and deserialization costs, which are clearly application dependent. Measuring these costs in isolation would be fairly straightforward, but, as they occur naturally, they are overlapped with the disk I/O and heavily interleaved with useful computations. Thus, it is difficult to identify their computations to overall overheads. Besides serialization, the major potential cost is that of the disk I/O itself, which is tricky to measure because it will be overlapped with useful computation and because it may benefit from OS and hardware caching.

5.1.4 Shuffle Storage

When Spark uses a many-to-many communication pattern, there is separation in time between then the inputs to the communication (the 'map outputs') are generated and when they are consumed. Consequently, Spark must store these map outputs. The default policy that Spark has chosen is to write the map outputs to the local filesystem of the worker where they were produced. The primary benefit of this choice over keeping the outputs in local memory is ensuring that in the absence of node failures that recomputing a partition will never require rerunning an entire stage. But this could be achieved by writing the data to disk on eviction instead.

Whatever the merits of writing shuffle data to the filesystem, the practical result is that whether shuffled data stays in memory depends on whether the filesystem keeps it in memory. For some Spark programs that use shuffles to achieve maximum performance, the filesystem caches must be large enough to hold the shuffled data. To achieve this, one needs to explicitly avoid allocating memory to the Spark application. Otherwise, the memory the OS would use for filesystem caches will likely be used for objects waiting to be garbage collected or for cached data that will not be accessed again.

5.1.5 Partition Sizing

An important factor in Spark's memory usage is the sizes of partitions it uses. This affects the size of temporary data while something is being computed and during aggregation. When partition are large, this temporary data can dominate memory requirements. Using very small partition sizes, in contrast, would cause task overheads to dominate runtime. In Section 5.5, I will evaluate the ability to estimate the effects of changing partitioning on memory requirements, though Spark lacks a generic configuration parameter to determine how many partitions to use to simply recommend setting.

Spark has several methods for choosing partition counts. In some cases, the user chooses. For example, the PageRank example program (used in our evaluation) has a 'number of edge partitions' command-line parameter, which is eventually supplied to various lower level Spark operators. Without such guidance, Spark tries to match the partitioning of its input; for example, when reading from HDFS (the Hadoop Distributed File System [80]), it will typically have at most one partition per HDFS block. Spark also has a default parallelism setting it uses as a minimum and when it cannot infer partitioning from the data source. For execution environments where the number of available cores is easily inferred, Spark configures this setting based on the total number of cores available.

5.1.6 Summary

Most of the memory managed by typical Apache Spark program represents cached partitions for its marquee abstraction, the resilient distributed dataset (RDD). But Spark has other ways to store values in memory, both to store temporary data to support computation and support alternative abstractions that are useful for some computations. This is not a failing of the RDD abstraction, but a result of efforts to efficiently support a wide range of distributed computations that one would expect from any similar framework.

These alternate forms of memory usage are important to analysis of memory usage. They can have large effects on memory requirements, and in particular, how memory requirements scale as the degree of parallelism changes. This split between types of memory usage is also exposed in Spark's configuration settings — a split between storage and execution memory. Some of the possibly memory usage is not explicitly configured, the framework performance may depend on 'unused' free space being available for file system caching.

Similarly, how memory usage is split within Spark changes data collection. As a practical matter, instrumenting each type of memory usage will often involves touching or using logs from different parts of the framework's codebase. After data collection, analysis must take into which configuration parameters are relevant to each kind of memory and whether that data has the lifetime of a computation or persists indefinitely.

5.2 Instrumenting Sizes

For each of the ways that Apache Spark's abstractions can use significant amounts of memory, SLAMR needs to record related program activity. It does not only need to record the sizes used, but it also needs to have enough information to put that in context. For persistent data, this includes when each part is accessed again, to reconcile with the policy for when it might be dropped from memory. For data which is divided into many partitions, this includes

Memory type	Reads	Writes	Partition sizes	Size if not stored	Relative timing
RDD	Added	Existing	Existing	Added	Added
broadcasts	Added	Added	Added	Existing	Added
aggregation buffers	Existing	Existing	Added	Added	—
map outputs	Existing	Existing	Existing (partial)	Existing	Added

Table 5.2: Table of types of data storage abstractions in Apache Spark and the instrumentation that was available and which I added to extract my recommendations. The ‘reads’ and ’writes’ column represent whether records of task execution indicated which and how much of that type of memory was available or that needed to be added. The ‘partition sizes’ column indicates whether one could infer the size of individual partitions from the access log, if this was relevant. The ‘size if not stored’ column indicates whether that partition size was also available if Apache Spark’s policy was not to ever store the corresponding value in memory. The ‘relative timing’ column indicates whether information about the order of the access relative to other accesses was available, which is necessary to account for recomputation or for cache-like accesses.

not only the aggregate size, but the size of each part to understand how it is distributed and what extra resources are needed while it is being processed.

Much of this information was already collected by Apache Spark as shown in Table 5.2. These records are created largely to assist with performance analysis and to provide feedback to users through a browser-based monitoring interface. Most notably missing from existing logs is information about the relative order of accesses. With this, my tool can identify recomputation of values that would be otherwise kept in memory. Such recomputation should not contribute to memory recommendations that already allocate enough memory to avoid it. Spark did not already record access order partly because different abstractions recorded logged their separately, so there was no relative ordering between different types of data.

Also commonly missing were indications of when data was read and the sizes of data when data is not stored in memory. Prior versions of Apache Spark also failed to record the sizes of some types of data, such as hashtables created for aggregation, when it was stored in memory. This lack of instrumentation reflects the purposes of Spark’s size measurements: the recording that does exist lets Spark staying within configured memory thresholds rather than to provide information for monitoring.

5.2.1 Spark’s Own Instrumentation

Spark’s memory management requires the sizes of the objects it stores. Typically, these objects are stored as native Java (or Scala, which compiles to Java bytecode) objects. Java intentionally does not expose how objects are laid out in memory. The objects managed by Spark often contain references to other objects that are logically part of the same unit. For example, a Scala object that ‘contains’ a string does so by reference, and that string in turn likely contains a reference to an array of actual character data. To effectively restrict the memory used by such objects, Spark needs to devote some effort to size estimation.

Spark's size estimation has two components. First, given a particular object, based on the language runtime in use, it needs to estimate the size of that object's header, fields, and any additional padding required to keep fields aligned according to the runtime's policy. Then it needs to account for the space used by objects referenced by this object. For both of these tasks, Spark relies on Java's reflection support, which it allows it enumerate the fields of objects at runtime and access private values. Since object references may contain cycles, Spark maintains a list of previously visited subobjects as it performs the size estimation task for one logical object.

If Spark examined all the objects referred to by those stored in cache, it would often be unreasonably slow. For example, if a Spark user creates an RDD partition with ten million short strings, Spark's size estimation would require examining each of these strings and storing them in the 'visited' data structure — in addition to the actual work to compute those strings. To avoid this overhead, Spark takes a random sample of those objects and scales up the resulting estimate. Typically, this produces good results, but one can construct cases where it would not. For example, in the case of the ten million strings, if one string was much larger than the others, Spark would have a low probability of sampling it and thus its estimate would be biased low.

Spark's size estimation strategy operates on an object at a time. However, objects that Spark counts separately can share storage. For example, one might have an array of objects with a string field and produce from this array containing only those strings. When reserving space for both of these arrays in, for example, its cache of RDD partitions, Spark will count these strings twice.

This overestimation is a rational design for Spark. It ensures that when Spark evicts a partition that the total measured sizes of the remaining partitions is an upper-bound on the space still required. Also, the sharing is fragile. For example, if Spark serializes and deserializes a partition, such as to send it over a network, shared strings and the like are likely to disappear.

Whatever the biases of Spark's size estimation, it is important to know the sizes Spark estimates. If one more accurately estimates the sizes of data than Spark and supplies this size to Spark as a configuration parameter, one would experience poor behavior. Spark would evict and recompute partitions based on its inflated estimates, The benefits being more accurate than Spark would be in configuring how much memory the language runtime under it is allocated.

5.2.2 Measuring Unrealized Sizes

While Spark accounts for what it stores in memory, once it decides not to store something, it generally does not measure its in-memory size. Though Spark's runtime only needs to know that the object exceeds its threshold for storage, SLAMR's recommendations require a good estimate of the total size. Thus, I modified Spark to estimate sizes in cases where it declines to keep some data in memory. Since these objects will never be present in memory in the instrumented run, this requires a slightly different size estimation strategy.

Generally, Spark uses a pattern where operations internally are simply given an iterator — an interface to retrieve the next value from a partition rather than directly accessing stored values. When a partition is not stored locally, Spark returns an iterator that computes values on demand rather than from an in-memory data structure. I modified Spark to wrap these iterators to collect information about the data being computed. The wrapped iterator samples the size of items as they are generated, measuring the size of elements at exponentially increasing intervals. SLAMR uses these samples, combined with the true count of the number of items generated, to maintain a running estimate of the total size generated.

There are a couple places in Spark where I found it necessary to insert this size-estimation shim: when unrolling fails and in shuffle aggregation at the reduce tasks. In both these cases, the size estimate obtained is an estimate of the elements of a hypothetical array or hashtable, which must be adjusted slightly to account for the additional overhead of the array or hashtable.

5.3 Counting Cache-Like Data Structures

Usually, most of the data in a Spark program is stored in various persistent caches. This includes all the types of memory marked as ‘persistent’ in Table 5.1. There are two caches SLAMR models. First, the ‘BlockManager’ on each worker node, where RDD partitions and broadcasts are stored and whose size is configured by the ‘storage’ configuration parameter. Second, the worker operating system’s page cache, where temporary files used in the shuffle may be cached by the underlying operating system and whose size is configured by whatever memory is left ‘unused’ on the worker machine. In both cases, the caches are managed with a policy which is LRU-like.

In the case of the BlockManager, SLAMR first simplifies the analysis by treating it as two caches: one for the RDD partitions and one for the broadcast data. I believe little is lost from analyzing these separately since programs that use large broadcasts are likely to use it alongside any RDDs. SLAMR model each of these caches as a single LRU cache rather than separate caches for each worker. For the broadcast data, that broadcasts are likely to be actively used by all tasks in a stage and so treat the broadcasts as if duplicated across all workers. For cached RDDs, SLAMR make adjustments for the distribution of the cache across nodes separately, along with analysis of the size of each partition, as described in Section 5.3.3.

I consider whether assuming LRU instead of modeling Spark’s strategy of not evicting items for others from the same RDD causes serious inaccuracies in recommendations. Since SLAMR makes recommendations that are intended to avoid cache misses entirely, it does not recommend points where part of an active RDD fits in memory. Even if I were to extend it to make more naunced recommendations, the interesting points are likely to be either to include all or none of each RDD in the cache, choosing which RDDs to based on the least-recently-used policy, so Spark's deviation from LRU will also be of no consequence.

This simplification permits much simpler analysis techniques. In particular, I will rely on 'stack analysis' techniques that permit analyses of cache behavior for multiple cache sizes provided that items evicted with cache size $N + X$ are also evicted with cache size N. This is not generally true of Spark's modified LRU policy.

5.3.1 Log Replay and LRU stack

SLAMR's cache analysis mechanism is based logs of accesses — both reads and writes. I modify Spark's event to provide this information for each task where it is not logged already as described in Section 5.3.2. For its analysis, SLAMR constructs a single access trace from lists of what is accessed by each task. A concern with constructing this single access trace is changes in the execution order resulting from differences in the degree of parallelism. The additional overlap between many tasks executing simultaneously is likely to increase memory requirements: more of a next stage of the computation will be computed before all of the last stage can be discarded.

SLAMR compensates for this overlap fairly directly: it requires extra space according to size of partitions being computed (which, incidentally, will be the space required for 'unrolling', which it separately needs to ensure is large enough). An alternate strategy would be to synthesize a unified access log with each task's accesses, repeated multiple times. The resulting estimation would effectively require that the accessed data be able to remain in the cache between the times the task's accesses is repeated, achieving a similar effect. To simplify, I merely replay the task accesses in some order. Rather than try to estimate the actual interleaving of accesses, I replay the accesses in the order that the tasks are started.

SLAMR replays this log and keep track of cached items in order by the minimum size of cache that would be allowed to hold them as in [45]. On each read, we count the number of bytes above the item in cache. This tells us the total size of a cache that would be required to keep the item cached. On each read and write, we move an item to the top the stack, representing that with the LRU policy, it will be in a cache of the smallest possible size. As my tool performs this analysis, it constructs a list of the cache sizes required for each item to be cache hit (ignoring compulsory misses). Its recommendation is the maximum of these points.

This analysis can also yield a rough estimate of the amount of extra work that would be required with less memory. By recording the size of each missed read and the corresponding required cache size, one obtains an estimate of the amount of extra work for each possible cache size. This could be easily presented to the user to suggest other possible sizes. However, the size of the missed reads is a very crude predictor of the actual miss cost as will be shown in Section 5.6.4.1, so additional work would be needed to turn these measurements into consistent recommendations for users.

5.3.1.1 Replay Performance and Optimizations

Done naively, though this stack analysis is more efficient than a full cache simulation at every possible cache size, replaying an access trace would be quite CPU-intensive. The fundamental problem is that the algorithm as described above is quadratic in the maximum depth of the stack: it requires scanning the stack repeatedly to count the size of items above it. Since the maximum depth is the number of partitions used in a program, which is easily many tens of thousands, this is unacceptable.

To avoid this, [45] observes that a splay-tree based approach can track these sizes with amortized log-linear time instead. Interestingly, results reported in [45] note that simpler approaches, which do have quadratic worst cases, achieve similar or better performance over traces exhibiting good temporal locality. SLAMR uses a simple optimization to the naive approach that takes advantage of locality in the typical access patterns of a Spark program. To speed up computations of the depth of an item, it maintains a small cache of items and their known depths. Whenever an item is moved within the stack, we updated this cache using that items old and new depth. To find an item's depth, it scans the linked list until it encounters an item in the small cache. Spark typically accesses items in groups and in consistent orders, such as when iteration through the partitions of RDDs, so when an item is accessed, knowing the depth of the item above it in the stack is likely to be useful for a near-future access, perhaps the next access.

Another potential cause of CPU-intensive log analysis is that logs can contain many entries for tiny items that are accessed frequently. The most notable example of this is small broadcasts, including the code for tasks. SLAMR filters out very small broadcasts (e.g., less than 32 kilobytes), since their effects should be negligible. Performance of stack analysis is a major problem in the architecture literature primarily because the resulting stacks are very large — typically consisting of some ordering of many millions of cache lines. I avoid this problem in Spark because each entry will be large, since it corresponds to partitions that are large enough to make the overhead of launching a task for each of these partitions manageable.

There is one notable item in Spark that does not have a small number of pieces per task — shuffle outputs and inputs. In a thousand partition shuffle, each 'map' task has 1000 different outputs which are accessed 'separately' by reduce tasks. As the number of partitions increases, the number of different shards of the shuffle data we would need to consider to precisely match reads and writes increases quadratically, quickly becoming unacceptable. This is common since Spark's default partitioning strategy often mirrors the partitioning of the source data rather than lowering partition counts closer to the available degree of parallelism. Thus, there is often one partition for every several hundred megabytes of data, easily resulting in hundreds of partitions with only tens of cores.

To avoid tracking many pieces of shuffles, my tool consolidates items that are part of the same logical shuffle. Thus, each shuffle is treated as one data item that is read or written many times, and whose size is the total size of the shuffle. Generally shuffles are accessed together in one 'wave'. Thus, the sizes of parts of the shuffle should have negligible effect on our size estimate. This simplification may miss effects where two shuffles overlap, so part of the first shuffle can be discarded while the second shuffle runs. But this type of behavior would change as the number of parallel tasks can change, so is not useful to rely upon for making configuration recommendations.

I could also make similar simplifications for consolidating RDD partitions into one 'RDD' entity. This generally will have no effect on our recommendation for similar reasons and drastically decreases the amount of items we need to track. However, for the analyses of the programs in this dissertation, this was not necessary.

After implementing these optimizations, I found that main constraint on the speed of the log analysis is the relatively slow parsing of Spark's log files. For most programs, this was the case even before the optimizations — they did not have enough tracked items for the quadratic effects to dominate. The slow speed of parsing is not a fundamental problem, but merely because Spark's event log is written in a text-based JSON format and the log parsing code it provides is not optimized for speed.

5.3.2 Additional Access Log Instrumentation

Our replay strategy requires an access log for each task. The information that Spark records by default is, unfortunately, not quite sufficient. Spark records what changes each task makes to the block manager overall. Unfortunately, this includes both too little and too much information. It includes changes that result from eviction of partitions, which my tool needs to simulate under a different hypothetical cache size. It does not include a record of reads and does not include any record when Spark's unrolling logic declines to store a partition in cache.

Also, unlike more traditional caching systems, when an item is not present in cache in Spark, it is often recomputed from other items in the cache. This means that some of the accesses that occur in a task would not occur if the cache were more effective. If the analysis fails to account for this, then it would falsely assume the dependencies needed to remain cached for longer than necessary.

I modified Spark to record an *ordered* access pattern to RDD blocks within a task. This includes all attempted reads and writes, regardless of whether they are successful. Because it includes information about unsuccessful reads and writes, the resulting access log allows recomputation to be consistently eliminated: every access starting after an unsuccessful read until a following write (successful or not) of the same partition is part of the recomputation. In my analysis, I preprocess the log to omit this recomputation.

To make this access log complete, it was necessary to not only record accesses to RDDs, but also to include accesses to shuffle data and broadcasts. Recording these in an access log was necessary to correctly eliminate such accesses if they are triggered by recomputation. Since shuffles and broadcasts were effectively managed by different subsystems of Spark, the existing instrumentation did not consolidate them well with RDD accesses.

In the case of broadcasts, Spark had little recording of creation or accesses to broadcasts. In particular, Spark's logs did not indicate whether a task required a broadcasted item. I added instrumentation of both creation of broadcasts, which generally occurs outside a task, and of reads of broadcasts, ensuring they were recorded in the access log for the current task.

For shuffles, Spark records the amount of activity in aggregation, but it did not provide a convenient way of correlating the map and reduce tasks involved in the same shuffle operation. To compensate for this, I record accesses to shuffle inputs and outputs in the per-task access log along with the internal identifier for the shuffle operation they belong to. To keep the size of the access log from exploding, I make only one entry for each shuffle operation per task along with total on-disk size of task's input or output.

5.3.3 Naively Dividing Caches Among Nodes

Spark distributes data across multiple workers, but lacks any explicit policy for load balancing the data between the nodes. Like MapReduce, it relies on locality constraints, starting with any information from the (possibly distributed) filesystem of its input. Then, data by default resides where it is computed on, and data is preferentially computed where its dependencies reside. To an extent, Spark is likely to correct for load misbalance: workers with cached data will be used first by Spark. Consequently, if parts of RDDs are not cached, the first workers available to compute them will tend to be workers with less data cached. Since computing a cacheable partition causes it to be cached locally, these workers with less cache will tend to accumulate more cached data. Nevertheless, Spark lacks any firm guarantees or goals about how data is load balanced.

For my tool's recommendations, I assume that Spark's load balancing works well. So, if there are X bytes to cache, each of N nodes ends up caching around X/N bytes. I allow for some node to be off from this size by the largest partition size P, so I require each worker to not more than $X/N + P$ bytes. A distribution of partitions that satisfies this constraint is achievable with a simple greedy assignment. If one assigns partitions to to each node greedily until at most $X/N + P$ bytes is used on each node, then either there are no more partitions to place or each node must have at least $X/N + P - P = X/N$ bytes used since no partition of size at most P fits on any. But each node having X/N bytes used would imply all partitions are placed.

Spark does not explicitly achieve this greedy allocation. Its policy of running backup tasks will, however, converge to this allocation, at some additional cost. If Spark evicts or does not store data on some node, it will recompute the portion of the data which did not fit where it was originally computed — eventually, on one of the nodes where there is enough room. Since nodes with less data stored are more likely to be free for computation after tasks with location preferences are placed, Spark's schedule is biased towards nodes which will have room without explicitly considering node capacities. Thus, the additional cost should be proportional to the difference balancing the computation for equal compute times and balancing the computation for equal storage size. This tradeoff is unlike the phenomenon that motivates the general cache sizing policy, where future evictions will prevent persistent storage of all the data that is accessed repeatedly.

Among my example programs running with multiple workers, assuming the minimum $X/N + P$ variation empirically resulted in essentially no recomputations in my sensitivity analyses. This is likely because my example programs, like most analytics programs, did not have a significant variation in their computation cost per byte, and the maximum partition sizes P were not very small relative to the node size so as to give little slack to the achieved schedule. When this is not true, I would expect some cost from recomputation instead that could be avoided by allocating extra space to accommodate the difference between load balancing the computation and load balancing the storage.

It is worth exploring what an appropriate correction would be when there was significant variation in compute time per byte and users prefer to load balance the computation instead of the storage. To do this, I did a simple simulation to estimate a more typical amount of variability. I modeled task durations log-normally distributed with a mean of 1 unit of time and a standard deviation of .5 or .25 or .1 units of time. Given this, I simulated 100 machines, each running one task at a time and varied the number of queued tasks. I performed 1000 runs of the simulation for number of queued tasks and recorded 99th percentile the maximum number of tasks run by any simulated machine in these runs. The results are shown in table 5.3.

In addition to showing the necessary adjustment, this analyses reveals that adjustments would have negligible effect if one assumes that the task computation times per byte have a standard deviation as high as 10%, even accounting for the 99^{th} percentile worst allocation over 100 nodes.

This analysis is intended to estimate the worst effects of this load misbalancing. I used a large number of simulated machines and report the worst misbalance, not accounting for how backup tasks would correct for this or how these effects would be hidden by other sources of inter-machine performance variability. I assume that partition sizes are constant and computation times per partition still vary by this factor. In contrast, most reports of large skew in partition computation times are accompanied by skew in their sizes, which will correct for the misbalance in storage. Probably most importantly, I model machines as running one task at a time, creating worse 'head-of-line' blocking than would occur when multiple tasks run on a single shared-memory machine.

Even the pessimistic analysis shows that if one assumes there is relatively little skew in task durations per byte of data, then there is little benefit from additional correction after the maximum partition size factor. This is likely to be the case in most analytics programs, including the ones in my evaluation. Otherwise, if there is going to be significant skew in task runtimes per byte of data, one can compensate for this by adding a factor proportional to the amount of skew to reflect the typical variability in task runtime.

5.4 Measurements to Recommendations

The various measurements of memory requirements do not map directly to configuration parameters. The ultimate goal is to produce minimum values for several configuration pa-

	duration $\sim \ln N(1, .5)$		duration $\sim \ln N(1, .25)$		duration $\sim \ln N(1, .1)$	
mean #	max #	% of (μ+1)	max #	% of (μ+1)	max #	% of (μ+1)
2	4	133%	3	100%	2	100%
4	7	140%	6	120%	5	100%
6	10	142%	8	114%	7	100%
8	13	144%	10	111%	9	100%
10	16	145%	13	118%	11	100%
12	18	138%	15	115%	13	100%
50	64	125%	57	111%	53	100%
100	118	116%	109	107%	104	103%
1000	1059	106%	1030	103%	1012	101%

Table 5.3: This table shows simulation results of the load misbalance resulting from running simulated tasks across 100 simulated machines with varying distributions of task durations. In each experiment, all tasks are initially queued, tasks start immediately after a task finishes, and there is no use of backup tasks or similar mechanisms. For each set of parameters, the 99th percentile of the maximum number of tasks run on any simulated machine is shown as well as what portion of one plus the mean number of tasks this is. The latter number indicates the additional correction factor that would be required after adjusting for the largest partition size.

rameters for each worker, which correspond to memory usage as described in Table 5.1:

- ‘storage’ size — the size of the region used by the BlockManager to store cached RDD partitions and broadcasts;
- ‘unroll’ size — the size permitted to be used for RDD partitions being computed but not yet stored persistently as described in Section 5.1.1.1;
- ‘execution’ (shuffle and aggregation) size — the size permitted to be used to perform group-by and aggregation operations;
- total JVM size — the size allocated to each worker JVM, which must include all Spark usage that manifests as Java/Scala objects;
- total space for OS page cache — space that the worker nodes’ operating systems can use to cache temporary space

The first three parameters are configured directly with Spark. In Spark 1.6 and later, these unroll and shuffle sizes are combined into the execution size. The minimum total JVM size is a restriction that we will apply through our garbage collector overhead analysis discussed in chapter 6.

My analysis of Spark’s memory usage produces these per-worker minimums given the number of workers W, the number of tasks per worker P. As discussed in chapter 7, these parameters are likely chosen by repeating the analysis for a variety of possible worker and task counts based on some utility metric. Most simply, a fixed number of tasks P can be selected and the number of workers increased until the minimum total JVM size plus the total page cache space suggested is less than the size of the underlying worker machines.

To compute the storage size, my tool adds up the sizes required to avoid cache misses under the stack algorithm model for the total broadcast space and the total storage space, modified as described in section 5.3.3 to account for it being split across W workers. To this, it add the P largest partitions observed in any access log for any task, which is also the minimum value for the unroll size. This size will often be much larger than the actual size observed in the program since the P largest RDD partitions will be corresponding partitions of related datasets which will not normally be computed simultaneously. Nevertheless, SLAMR uses the top P sizes so that in the event of a failure and recomputation data should still end up cached and if each partition is composed of a very small number of large records, there will be enough memory to store them. Given typically small partition sizes, there would be little benefit to doing more sophisticated analysis to take the top P sizes of unrelated partitions.

To compute the shuffle size, my tool adds up the sizes of the P largest aggregation input and outputs I add up both inputs and outputs because the worst case is a task both reads from a shuffle and writes to a shuffle (e.g. a multi-step group-by). Like for RDD partitions, I do not try to compensate for related partitions being responsible for the top P aggregation hashtables.

To compute the space for page cache, I add space required to avoid cache misses for the shuffle storage data from the access logs for the tasks. When doing this analysis, as a simplification, I only consider the total size of the entire shuffle, since my instrumentation discards the sizes passed from input 'map' task i to output 'reduce' task j for space and overhead reasons anyways.

5.5 Repartitioning

A likely cause of unexpectedly large memory requirements for Spark is the use of large partitions. As described in Section 5.1.5, Spark has several ways of choosing partition sizes, which can result in a variety of actual partition sizes and counts depending on the environment. When these heuristics or explicit user choices lead to Spark program using a very small number of partitions, this can result in much larger memory requirements than a somewhat larger number of partitions.

In the extreme case, where the number of partitions is equal to the degree of parallelism, a program can require roughly twice as much space as small partitions: when each partition is being computed, both it and its same-sized dependent partition would need to be in memory simultaneously. This side-effect of Spark's immutability-based computation likely belies the intuition of many programmers who would set the number of partitions equal to the number of cores to minimize overhead.

Because of the relationship of partition counts to performance — less partitions may be faster and more partitions slower — and the non-trivial costs of repartitioning, I cannot easily compute the 'ideal' partition count based on memory utilization information. Nevertheless, I try to roughly predict the effect of changing the partitioning in the program's memory requirements, so that users can understand the likely effect on memory consumption and weigh that against their guesses about the performance impacts. Whenever the recommended space for unroll and shuffle memory is large, one can suggest to the user that increasing the partition count is likely to be useful. A rough prediction can close the loop by giving a concrete estimate of how much to change the partitioning by.

To make simple predictions of the effects of repartition, I adjust the observed sizes that are proportional to a partition size — the actual size of RDD partitions in memory and the sizes of a partition in aggregation. I naively multiply these sizes by this factor and perform the rest of our configuration production procedure.

These quick repartitioning estimates have some risk — when scaling up to more partitions, this will always assume the largest partition gets smaller even though this is untrue for some datasets. Some more detailed instrumentation could help with this, though as noted earlier, one is also likely to experience issues with Spark's measurement of partition sizes if there is a lot of skew of the record sizes.

5.6 Evaluation

I evaluated our tool's recommendations on our suite of example programs described below. For programs that could fit comfortably on one virtual machine, I choose an input size that would do so for the purpose of avoiding effects from multiple machines impacting sensitivity analyses.

I choose a single configuration of Amazon EC2 Virtual Machine for each program. Usually, I would use one of the M4 (1 core:4GB memory ratio) virtual machine configurations, but for a handful of programs that would not execute fast enough or seemed to require more memory per core, I instead used the C4 (1 core:2 GB ratio) or R3 (1 core:8GB ratio) virtual machines. For programs would not fit comfortably on one virtual machine, I chose a type of virtual machine size that would require 2–4 workers for the base configuration. The number of workers used was the minimum that would fit the minimum recommended sizes. As is typical of public cloud virtual machines on the market as of this writing, these virtual machines had storage backed by SSDs, which was used for Spark's temporary files and for non-generated input files.

To produce initial size estimates, I used a single execution based on the size of the input data provided or a simple estimate of the size of generated data. When possible, this initial run is at one times the size of the input, but in some cases a larger size is required for a run to complete successfully. In these cases, I tried twice the input size, then four times, then six, and so on, until one completed in less than ninety minutes. For each program, I set a

Program	Worker Type	Base Size
ADAM Sort	multiple r3.xlarge (4 cores, 30.5 GB)	2x input
ALS	multiple r3.xlarge (4 cores, 30.5 GB)	8x input
Carat pipeline	multiple r3.2xlarge (8 cores, 61 GB)	10x input
ADAM Transform	d2.xlarge (4 cores, 30.5 GB)	1x input
GLM (500k points)	m4.2xlarge (8 core, 32 GB)	1x input
PageRank	c4.4xlarge (16 core, 30 GB)	12x input
KMeans (100k points)	m4.4xlarge (16 core, 64GB)	1x input
Naive Bayes (100k points)	m4.2xlarge (8 cores, 32 GB)	1x input

Table 5.4: Example programs, their input sizes, and the configuration used to run them.

time limit for all runs, which was at least twice the amount of time it took for the initial sample execution.

To evaluate recommendations, I performed sensitivity analysis on each of the parameters. I only performed this analysis for programs which had a non-trivial minimum for that configuration parameter. For the other programs, I checked that varying each parameter from around 0.1 GB up to 0.5 GB did not result in any significant change in performance. When possible, I also obtain a more direct measurement of whether Spark is recomputing data or using the local filesystem. This show some cases when recommendations comply with the heuristic of keeping the computation in memory even when that does not substantially help performance.

5.6.1 Example Programs

The example programs we used for evaluation primarily represent machine-learning workloads, typical of those for which Spark was originally designed. I use a mix of synthetic benchmarks and those using real datasets, and, for comparison, the notion that memory requirements can be predicted from input sizes. Table 5.4 summarizes the example programs, the configuration used for their initial instrumented run, and the size of virtual machines used to run them. These programs represent a range of Spark workloads, including iterative machine-learning tasks and single-pass data transformations, working with data easily represented as dense matrices and data represented as social-network-like graphs.

I use several examples from Databricks's spark-perf benchmark suite [24], which has good coverage of the machine learning algorithms included with Spark, but uses synthetic input data. I ran these programs with the parameters the benchmark would use for some scale factor, but vary that scale factor because the amount of actual input and computation required for each varies significantly within the benchmark. I also collected a small number of more realistic examples. I have an old version of the Carat[61] analysis pipeline along with around 5GB of compressed observations. I also ran Spark's included GraphX [33] PageRank [67] example against the LiveJournal graph [81] and Spark's included (MLLib-

Program	Input Size	Shuffle	Storage	Unroll	Non-JVM
ADAM Sort	12.5 GiB	6.9 GiB	0.0 GiB	0.0 GiB	0.0 GiB
ADAM transform	17.0 GiB	0.1 GiB	0.0 GiB	0.0 GiB	0.0 GiB
Carat (small)	0.6 GiB	0.5 GiB	3.3 GiB	0.5 GiB	1.3 GiB
Carat (larger)	3.6 GiB	1.9 GiB	26.9 GiB	0.6 GiB	34.1 GiB
EMLDA (250k)	0.9 GiB*	0.1 GiB	1.2 GiB	0.0 GiB	1.2 GiB
GLM (500k)	3.7 GiB*	0.1 GiB	15.4 GiB	0.3 GiB	0.0 GiB
PageRank (128 parts)	1.1 GiB	0.1 GiB	9.6 GiB	0.2 GiB	3.2 GiB
PageRank (64 parts)	1.1 GiB	0.1 GiB	8.3 GiB	0.9 GiB	2.0 GiB
KMeans (128 parts)	3.7 GiB*	0.1 GiB	7.9 GiB	0.3 GiB	0.0 GiB
ALS on Netflix (one node; 25 dims)	2.8 GiB	3.6 GiB	11.4 GiB	0.3 GiB	129.4 GiB
Naive Bayes (100k)	3.7 GiB*	0.1 GiB	8.3 GiB	0.3 GiB	0.0 GiB

Table 5.5: Values of recommendations for the configuration of a single 4-core worker for various example programs. Input sizes marked with an asterisk represent programs which generate their input rather than reading it from a file; the size given is the storage required based on the number of 4-byte integers or floating point numbers involved.

based) alternating least squares (ALS) example against the Netflix prize [59] dataset, which I reformatted to match the Movielens dump import format Spark expects.

In addition to machine-learning focused benchmarks, I ran the ADAM [54] genomics suite on an file from the 1000 genomes program [92]. I used two tasks from it, a file format transformation and sorting the genome reads.

To demonstrate some pathological cases which do not occur in my more natural example programs, I constructed some synthetic examples. In particular, my other example programs tended not to demonstrate large requirements or effects from variation in shuffle storage. I constructed some synthetic examples that had very high recomputation costs and very high serialization costs to exploit these.

5.6.2 Overall recommendations

Table 5.5 shows my tool's recommendations for our example programs for a single 4-task worker. These recommendations are obtained from analyzing a successful run which had more memory available than all in-JVM recommendations. This illustrates that our recommendations are not easily substituted by a simple inference from the input size. This is not very surprising as the format in which data is stored is generally not the one used on disk or a flat array of values.

In addition to differing in their relationship between the logical data size and memory requirements, these programs differ in what types of memory they demand. This is an expected result of the variety of types of applications I evaluate. Many of these programs seem to be make essentially no user of Spark's shuffle support, and so require essentially no

Program	Base	+ Event Log	+ Spark sizes
EM-LDA	168 s ± 2 s	169 s ± 2 s	171 s ± 2 s
ALS	801 s ± 11 s	799 s ± 7 s	799 s ± 8 s
ADAM sort	4563 s ± 87 s	4537 s ± 84 s	4563 s ± 0 s
Carat	2419 s ± 13 s	2382 s ± 4 s	2400 s ± 31 s
GLM	83 s ± 2 s	85 s ± 0 s	85 s ± 4 s

Table 5.6: Execution times with and without additional instrumentation at a configuration close to the recommended configuration.

memory be allocated for that. Other programs, like ADAM genome sort, make little use of Spark's traditional RDD storage, but have very large requirements from memory related to shuffle.

5.6.3 Overhead

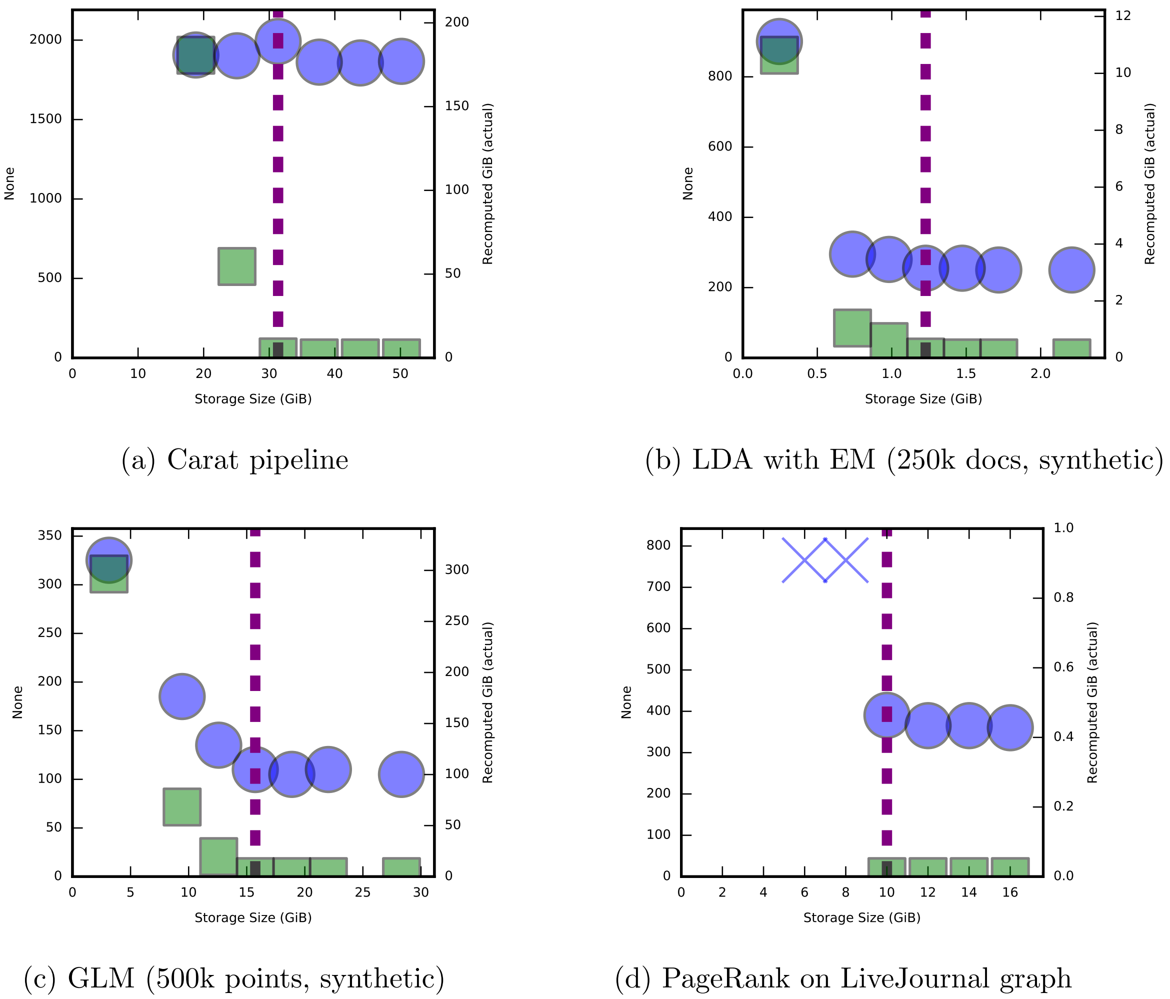
A natural concern is the cost of the instrumentation SLAMR requires. To measure this, I compared several example programs with all instrumentation disabled, with Spark's existing event logging enabled, and with my additional instrumentation enabled. To avoid startup effects, before each measurement a warmup run not included in the results was done prime caches.

Table 5.6 shows the results. The times show the average and standard deviation from three runs on the same set of virtual machines for this set of programs. For most programs, the additional overhead appears to be insignificant compared to other sources of small variations in program runtime.

5.6.4 Sensitivity Analysis

To evaluate my tool's recommendations, I recorded program performance as each configuration parameter varied around the recommend minimum. If the minimum was correct, then I expect to see negligible changes in performance from increasing the parameter. But below the recommended minimum, I expect to see a dramatic increase in runtime. The location of this increase indicates how much my recommendations lose due to being overly conservative.

To do these analyses, I ran the program with each configuration changed in increments of at most 20% of its suggested minimum, all other Spark parameters at their recommended minimums, sufficient JVM space to ensure good garbage collector performance, and enough non-JVM space to satisfy the minimum. Adjustments to configuration parameters that should increase managed memory usage are accompanied by a corresponding change to the JVM parameters. For each run, I report the runtime and, where available, a more direct measurement of how often Spark falls back to slower storage. In some cases, misconfigurations cause the program exceeded a timeout, which was at least twice the runtime of the run at the successful configuration.



(a) Carat pipeline

(b) LDA with EM (250k docs, synthetic)

(c) GLM (500k points, synthetic)

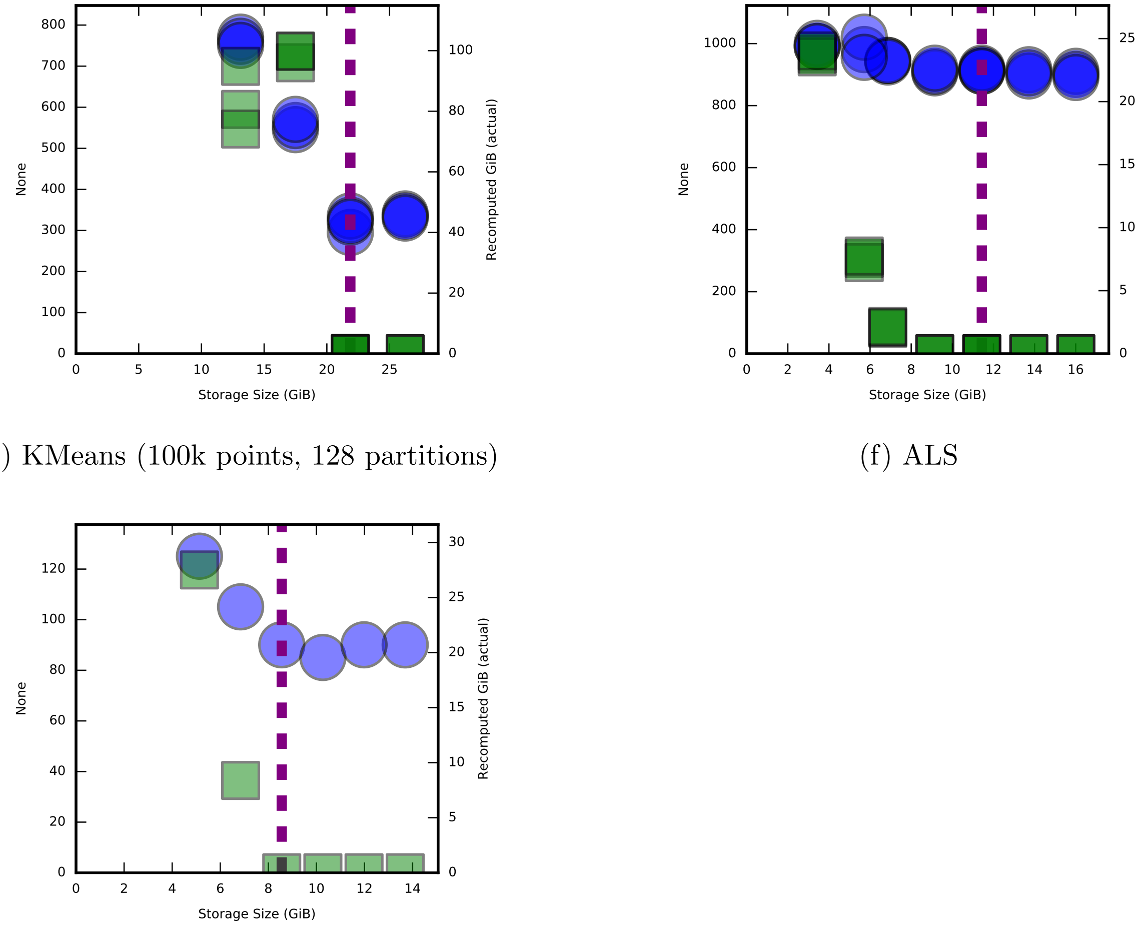
(d) PageRank on LiveJournal graph

Figure 5.1: Spark storage size versus recomputed bytes (squared) and runtime (circles). X's represent failed or timed out runs. The vertical line represents the recommended minimum configuration.

5.6.4.1 Storage Size

Figure 5.1 shows the variation in both runtime (circles) and number of bytes recomputed (squares) as the storage size configuration parameter is changed for several of our example programs. The horizontal line indicates my tool's minimum recommendation, and Xs indicate runs which exceeded our timeouts or crashed. To compute the number of bytes recomputed, I rely on the analyzing the access logs from each task. The number of bytes recomputed is the total size of writes to every RDD partition excluding the first write for each partition.

For all of these programs, the minimum memory is clearly large enough to avoid both

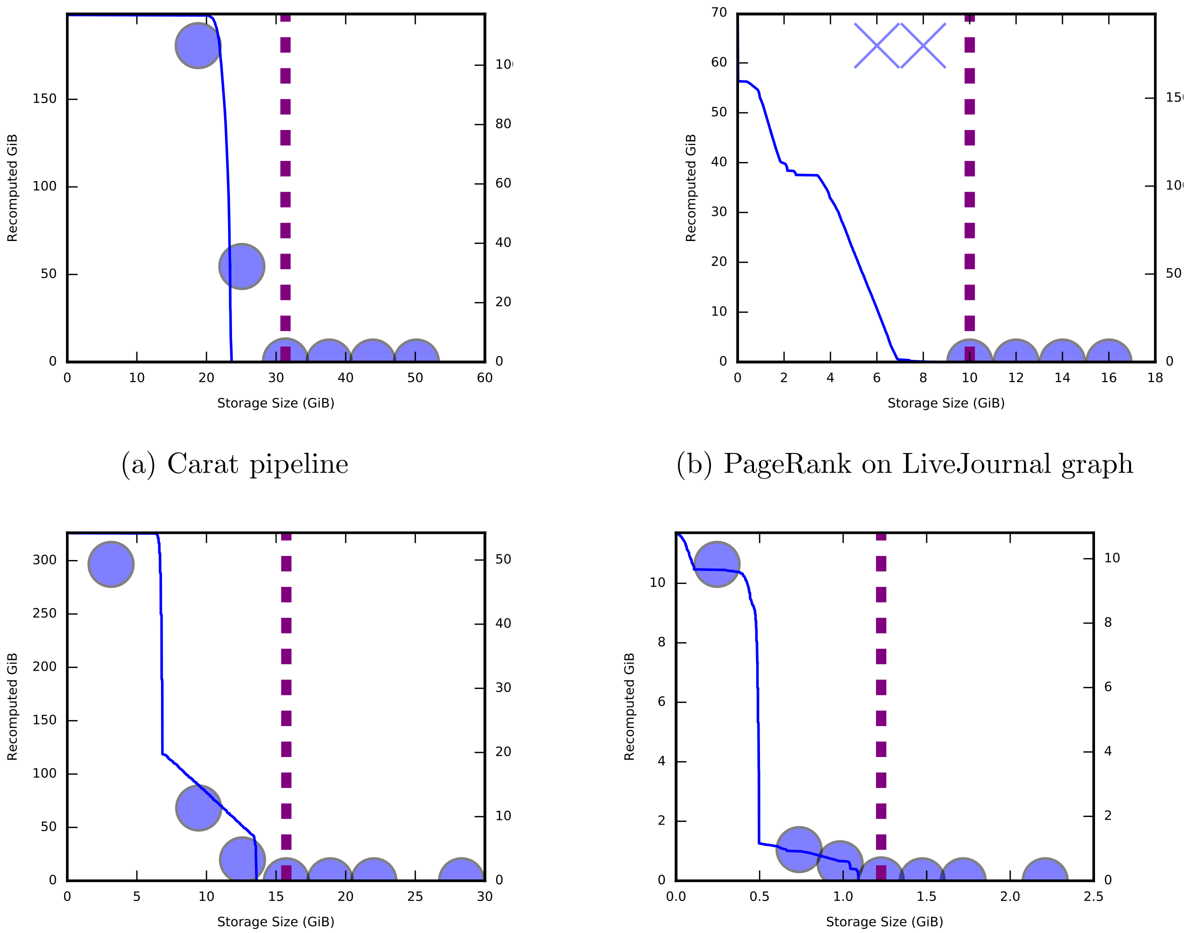


(e) KMeans (100k points, 128 partitions)

(f) ALS

(g) Naive Bayes (100k points)

Figure 5.1 (con't): Spark storage size versus recomputed bytes (squared) and runtime (circles). X's represent failed or timed out runs. The vertical line represents the recommended minimum configuration.



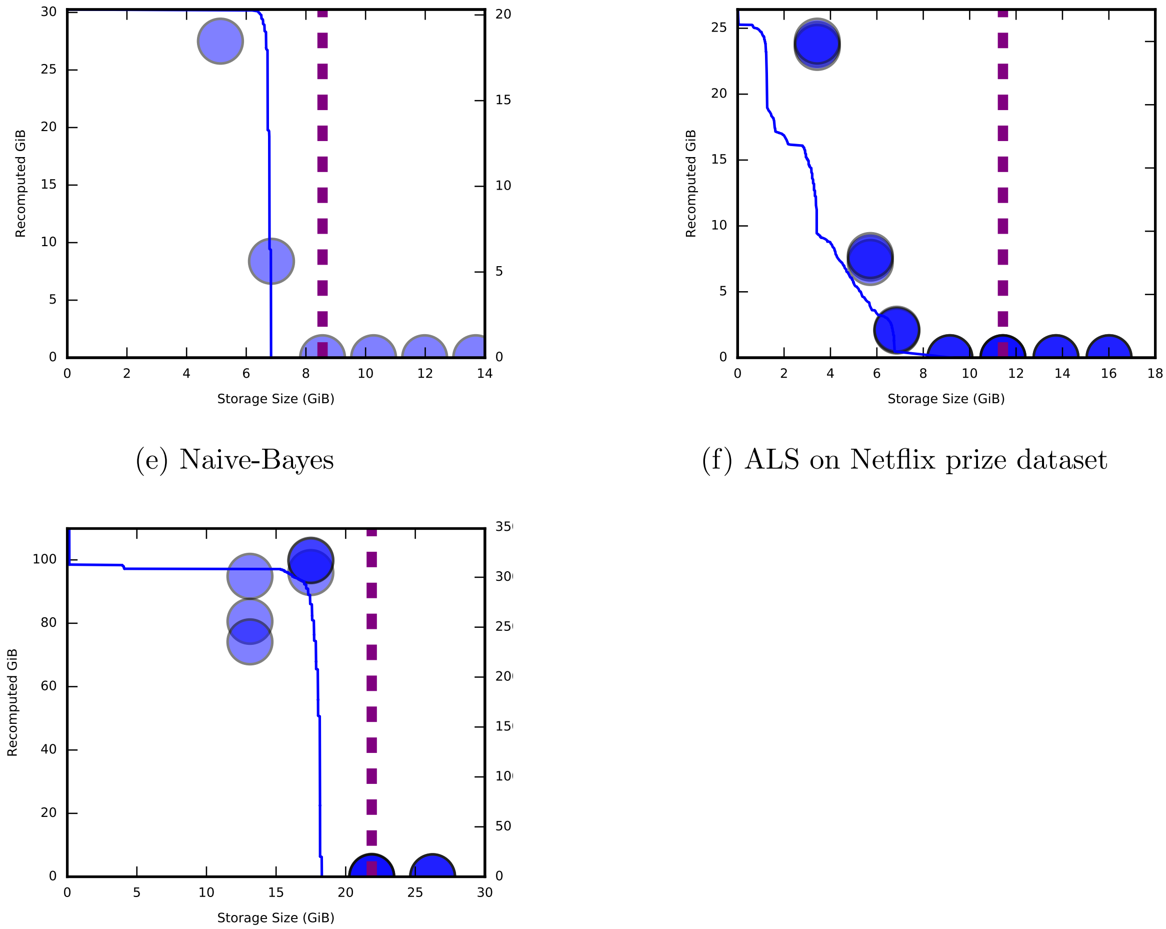
(a) Carat pipeline

(b) PageRank on LiveJournal graph

(c) GLM (500k points, synthetic)

(d) LDA (250k documents, sythetic data)

Figure 5.2: Storage size versus actual recomputed bytes and predicted cache miss bytes for RDD partitions. X's represent runs which failed or exceeded a timeout. The vertical line represents the recommended minmum storage size configuration.



(e) Naive-Bayes

(f) ALS on Netflix prize dataset

(g) KMeans (100k points, 128 partitions)

Figure 5.2 (con't): Storage size versus actual recomputed bytes and predicted cache miss bytes for RDD partitions. X's represent runs which failed or exceeded a timeout. The vertical line represents the recommended minmum storage size configuration.

recomputation and the possibility that more would substantially decrease runtime. For most programs, 80% of the recommended shows easily measurable degradation in performance. There are two exceptions: for the ALS example, the performance degradation is negligible until around 70% of the recommendation, and for the Carat pipeline, there appears to be no performance degradation whatsoever.

The stack algorithm based model makes some predictions about the amount of recomputation required, which are shown alongside the measured amount of recomputation in Figure 5.2. These estimates can give insights into the causes of the anamolous lack of loss in performance. In the ALS example, the stack-based model predicts relatively little recomputation for the sizes where relatively little performance impact or recomputation is observed. But this is not a universal predictor of performance; for example, for the PageRank example, any recomputation apparently has enough dependent computations to make runtime increase drastically.

5.6.4.2 Aggregation Space

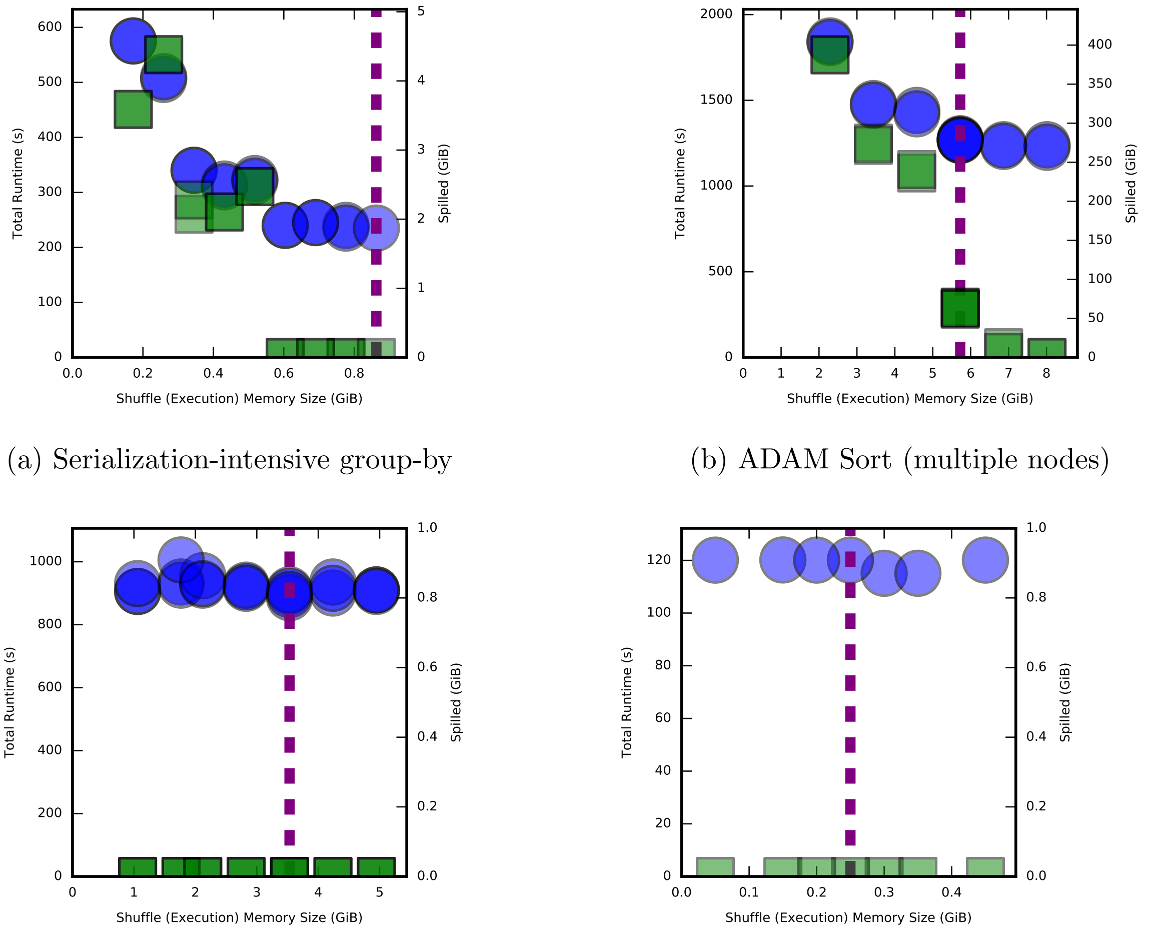
Figure 5.3 shows the impacts of varying space allocated for aggregations (within the tasks on either side of many-to-many shuffles) for the small number of programs which required non-trivial space for this and a synthetic example. For all the non-synthetic examples, the performance impact of the shuffle requiring spilling was modest. For example, supplying less than half the required space to the ADAM sort example increased the runtime by less than 50%. One likely cause for this is that the storage I spilled to was fast — backed by SSDs and probably cached by the local OS — so this result may not hold in all environments.

Spark records the amount of data that is 'spilled' to the local filesystem because there was insufficient space for the aggregation to its event log. These measures show that my tool's minimums do predict a size that avoids almost all spilling, though there are some cases where a small amount of spilling occurs. I suspect the cause of this is that Spark reserves space based on the anticipated size of its internal hashtable as continues to collect input rather than the total size that I estimate. Thus, full accuracy may require correcting slightly for Spark requiring more space.

I also created a synthetic example to illustrate one negative effect of spilling that occurs regardless of the costs of I/O: serialization. I wrote a Spark program that did an group-by using deliberately expensive to serialize key and value types. These keys and values represented small objects that simply contained a integer value, so most of the effort in serializing and deserializing them was overhead. The results are shown in 5.3a, where one can see the marked increased runtime from the high serialization costs even though it is certain that all the data is remains in the local OS's page cache.

5.6.4.3 Non-JVM Space

Figure 5.4 shows the effects of varying the available non-JVM space for the shuffle-intensive ALS example, to allow the OS cache map outputs from many-to-many 'shuffles'. Unlike



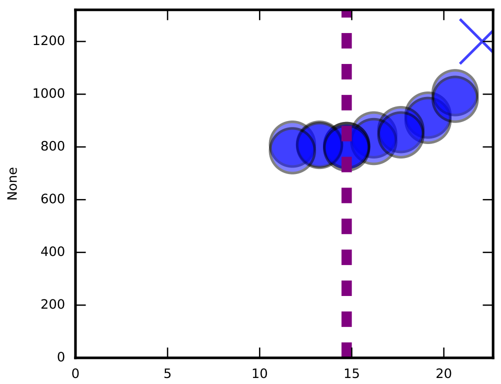
(a) Serialization-intensive group-by

(b) ADAM Sort (multiple nodes)

(c) ALS

(d) PageRank (16 partitions)

Figure 5.3: Spark shuffle size versus bytes spilled to disk (squares) and runtime (circles) on selected examples. The vertical lines shows our inferred recommendation.



(a) ALS (multiple VMs)

Figure 5.4: Effect of filling excess space on performance. Xs indicate runs which exceeded a timeout. Vertical line represents our tools' recommendation based on page cache we estimate is required to keep shuffles in memory.

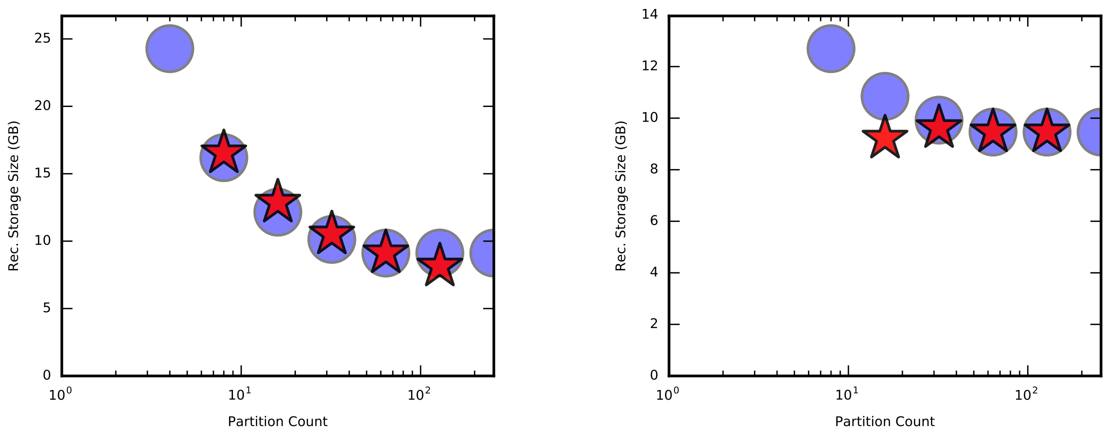
other configuration parameters, this value is set implicitly by not allocating space to a Spark worker. To manipulate the amount of free space without changing the amount of garbage collector overhead, I filled excess memory on the VM so it would not be available to the OS page cache or the Spark. The x-axis on the graphs show the amount of extra space reserved.

The graph shows a gradual degradation in performance presumably due to fallback to reading from the virtual machine's SSD. At sufficiently large sizes, the memory pressure appears to cut into the memory available to the JVM or other essentially system services, causing the Spark program to perform unacceptably. For other programs (with more modest use of map outputs), I did not observe significant changes in runtime from depriving them of space to cache map outputs.

5.6.5 Repartitioning effects

To evaluate the rough predictions of the effects of repartitioning, I examined the behavior of tow programs which gave convenient direct control over their primary partition count — the PageRank and KMeans example. Figure 5.5 requirements my tool predicted by naively scaling sizes from as described in section 5.3.3 from the measurements from a 64 partition and the recommendations produced from runs with different partition counts. For the KMeans benchmark, predictions are very close and generally consistently high.

For the PageRank example, with natural graph data, the apparent memory requirements appear to surprisingly decrease as the partition count is decreased. I believe this is an artifact of how Spark measures the size of partitions. As discussed in section 5.2.1, Spark overcounts data, like strings, which is shared between multiple RDD partitions. The PageRank example



(a) KMeans (100k points)

(b) LJPR on GraphX

Figure 5.5: Recommended size estimate for varying partition count configuration based on runs with 64 partitions (circles), and actual recommendations based on runs at given partition counts (stars).

.

Program	**Input Size**	**Shuffle**	**Storage**	**Unroll**	**Non-JVM**
Carat	3.6 GiB	1.7–1.9 GiB	26.9–32.0 GiB	0.6–0.8 GiB	33.9–34.2 GiB
EMLDA	0.9 GiB*	0.1–0.1 GiB	0.8–1.2 GiB	0.0–0.0 GiB	1.2–1.2 GiB
GLM	3.7 GiB*	0.1–0.1 GiB	14.9–16.1 GiB	0.3–0.3 GiB	0.0–0.0 GiB
PageRank	1.1 GiB	0.1–0.1 GiB	9.6–9.6 GiB	0.2–0.2 GiB	3.2–3.2 GiB
KMeans	3.7 GiB*	0.1–0.1 GiB	7.9–8.6 GiB	0.3–0.3 GiB	0.0–0.0 GiB
ALS	2.8 GiB	3.5–3.5 GiB	10.6–10.6 GiB	0.3–0.3 GiB	37.2–37.2 GiB
Naive Bayes	3.7 GiB*	0.1–0.1 GiB	7.6–7.9 GiB	0.3–0.3 GiB	0.0–0.0 GiB

Table 5.7: Range of recommendations from varying storage size from the experiments used to construct Figure 5.1. Results of incomplete executions are ignored. * indicates input size is synthetic data.

shows strong indications that this overcounting is occuring: although the peak size Spark assigns (on workers) to RDD partitions is around 9 GB, the program can be successfully run with only around 5 GB of memory available. Given this, decreasing the partition count probably increases the accuracy of size estimates because, for example, shared strings are only counted once within each partition.

5.6.6 Consistency of Recommendations

I examined the recommendations that would be produced based on runs where insufficient memory was allocated. In particular, I am interested in whether my predictions are consistent even when additional computation is required. To evaluate predictions of storage requirements, I examined the recommendations that would be made from each of the runs used for my storage sensitivity evaluation in Section 5.6.4.1.

As Table 5.7 shows, most of the recommendations are consistent, but there is some variation in the storage size recommendations. The largest variation occurs for the LDA example, which experiences around 60% differences. However, results from our sensitivity analysis for this example reveal that there is not much difference in performance from whatever recomputation is occurring. A possible explanation for this variation is that there is choice in the order in which stages are executed which changes when recomputation occurs. With the exception of the LDA example, where there was variation, the variation is all clearly in the range where there is no gain in performance. A likely cause of this variation is that there is some variation in the order tasks are executed, triggered by the change in timing from recomputation. But my tool deals with this separately by adding extra space for being-computed partitions.

5.7 Conclusion

By taking advantage the information easily available through analytics framework abstractions, I can obtain quick estimates of memory requirements for analytics program. I demonstrated this by constructing a model of memory usage based on a record of Apache Spark application's accesses to framework-managed data. As the evaluation showed, the resulting tool produces recommendations for the configurations required to keep the application's computations in memory. The instrumentation my tool needs is a small additional step from the tracking already done by Spark to manage its own memory or for diagnostics. The overhead of this additional instrumentation is very small, so it would be viable to enable it for all runs of an application.

My tool's recommendations make a compromise: they do not precisely navigate the memory/performance tradeoffs a program faces. Consequently, they are frequently more conservative than they need to be — a user would experience nearly the same performance by allocating less memory. From this compromise, I gain efficiency, there is little overhead in terms of user effort or lost performance, and consistency, the recommendations are large enough, unlikely to be offset by measurement anomalies. This consistency comes from targeting not performance per se, but choosing the 'memory' side of memory/performance tradeoffs offered by a user application.

Chapter 6

Predicting Garbage Collection Overheads

Most memory usage in a batch analytics program typically has a definite tradeoff — either data fits in memory or does not. Garbage collection overheads arguably follow this pattern, but the data to fit in memory is everything allocated by a program — it is not like a cache where one only pays for current active items. In practice, where garbage collection overhead is significant, it is not practical to store all allocations; they amount to hundreds or thousands of times the active memory of the program. Thus, memory recommendations must, instead, account for a memory/time tradeoff.

For Spark, the garbage collection overhead varies significantly between programs — because of different behavior of application-defined functions and different types of objects being stored by applications within Spark's data structures. For these programs, I propose a simplified model of their garbage collection overhead. It captures the behavior of a family of throughput-oriented garbage collectors. Combined with this model, I capture a summary of how the program's memory demands drive the garbage collector.

To make collecting this data and evaluating the model efficient, I focus on measuring an 'average' behavior over the entire program, and I avoid predicting short-term effects and garbage collection latencies. Rather than precisely model real garbage collectors, I model a simpler garbage collector that should provide a bound on typical throughput-oriented garbage collectors. The primary goal of this model is not precise predictions but avoiding configurations with very high overheads by predicting an upper-bound on the overhead of a real system.

In this chapter, I describe and evaluate this model that forms a component of my memory recommendation tool SLAMR. I begin by describing the operation of garbage collectors generally, introducing related terminology. Then, I review a particular simplified garbage collector that is the basis of my model. To apply this to real Apache Spark programs, I describe what data I need to collect and the mechanisms I use within a particular language runtime, OpenJDK, to collect it. Given this data and the simplified model, I then show how to predict the number of garbage collector cycles across a range of configurations. Based on the correlations in overhead observed across a variety of Spark programs, I extend predictions of the amount of garbage collector activity to an estimate of the overall garbage collection overhead of programs. I then present an evaluation the effectiveness of these predictions across a variety of Apache Spark programs.

6.1 Garbage Collector Overview and Terminology

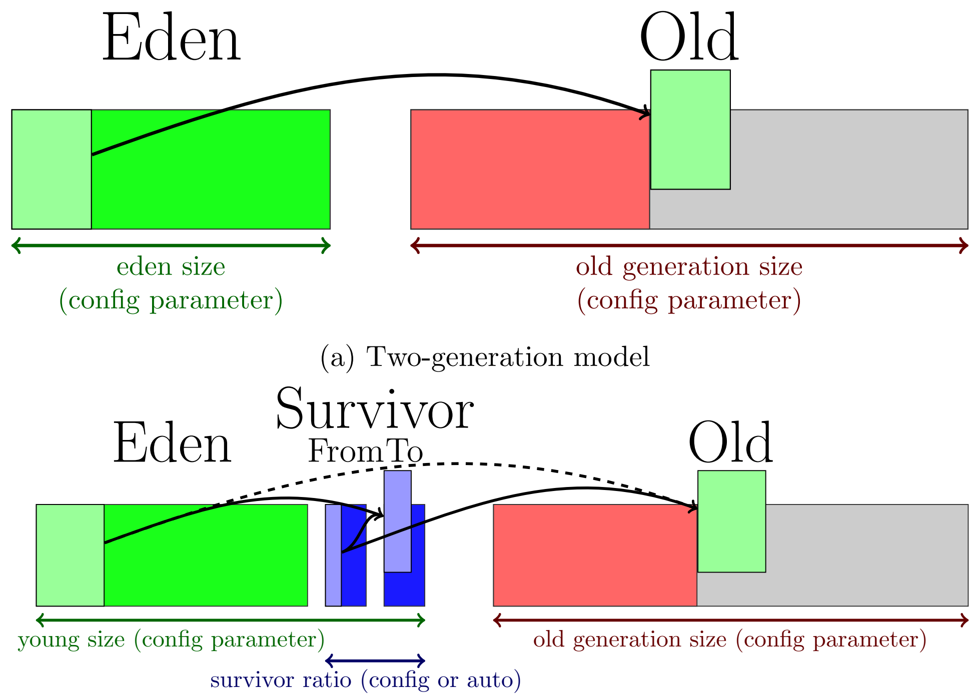
To provide a background for describing garbage collectors, I will briefly review the operation of garbage collection and some terminology to describe it. The terminology I use in this dissertation is mirrors the terminology used by OpenJDK in describing its garbage collectors.

Fundamentally, garbage collectors manage a *heap*, the entire managed memory region for a program. User programs allocate *objects* in this heap. These objects may contain *references* to other objects on the heap. In addition, there may be references to objects from outside the heap, such as in the stacks and registers of running threads, tables of global variables, etc. In fact, some set of references outside the heap represents all the ways the running program has to identify objects on the heap. This set of references is known as the *root set*. An object is considered *reachable* or *live* if there is any path via references from the root set. All the paths from the root set form a DAG of reachable objects. Unreachable objects are *garbage*, and making the space they occupy available for future objects is *collecting* them.

In the simplest form of garbage collection, one scans the graph of reachable objects starting with the root set by following the references in the root set and each object discovered. Each object found in this scan is *marked* as reachable. Then, there are multiple options about how to reclaim the space used by unreachable objects. The most common choice is to perform *compaction*. All reachable objects found are moved to a contiguous part of the heap, with references contained in the root set and these objects adjusted accordingly. I will call a garbage collection pass of this sort a *full collection*.

In developing SLAMR's model of garbage collector operation, I closely model a somewhat more complicated design, which divides the heap into multiple *spaces* or *generations*. These generational garbage collectors, first described by Lieberman, et al., [50], call these spaces *generations* because the objects are typically partitioned based on when they were allocated — their *age*. This partitioning takes advantage of objects with shorter ages being more likely to become garbage.

These generations can be collected independently from each other. As with a full collection, one marks all reachable objects and then somehow reclaims the memory used with a generation from each other. This scanning needs to account for objects in other generations that contain references. This could be done by scanning through all objects, regardless of which generation they are in. But, most commonly, this is done by having a *card table* to identify which parts of other generations might contain references and limit the amount of scanning which must be performed. A card table generally has one entry for every page or similarly sized region, indicating whether any objects in that region might contain references


Eden
Old
eden size
(config parameter)
old generation size
(config parameter)
Eden
Survivor
From To
Old
young size (config parameter)
old generation size (config parameter)
survivor ratio (config or auto)

(a) Two-generation model

(b) Three-generation model

Figure 6.1: A simplified model of garbage collector regions (two-generation model) and a better approximation (three generation model) of the garbage collector that is the default in OpenJDK.
Shaded parts in each region represented used memory within each region. Solid lines and raised boxes represent typical movement of objects between regions; dotted lines represent less common movements.

to other generations. This card table can be maintained with a *write barrier*, some code inserted before every reference update to also update the corresponding card table entries.

After identifying the reachable objects within a generation, the collector needs to mark the remaining memory as free. In a generational collector, this is most commonly done by *promoting* or *evacuating* the objects to another generation. Like with compaction, the collector needs to fix all references to these objects.

6.2 Two-Generation Model

In my simplified model based on a typical generational garbage collector, managed memory is divided into two generations: the *eden space* and the *tenured space*. These generations have fixed sizes which do not change over the course of a program. The configuration problem for the garbage collector is limited to choosing these sizes, which in results in choosing the overall memory footprint of the program.

Objects are allocated in eden space. When there is not sufficient room to allocate an object there, then an *eden collection* occurs first. During this, all threads are stopped, and a marking phase occurs. This marking phase scans references to the eden space starting with the root set and changed references in tenured space. After this marking phase, all objects marked as live are copied to the tenured space. If this would exhaust the tenured space, then a *full collection* occurs. In a full collection, threads remain stopped, a marking phase occurs starting with the root set but covering all accessible objects, not just objects in eden space. Then all live objects are moved to the beginning of the tenured space and the remainder of the tenured space is marked as free. Since some of these objects were previously in eden space, there may not be enough available space to do this; if so, an out of memory error is raised and the program crashes.

My goal in applying this model is to determine whether an out of memory error is likely and to estimate the amount of time spent with threads paused to perform collections. I do not try to model all overheads, but merely the change in overheads that will influence configuration decisions. For example, generational garbage collectors usually require an extra 'write barrier' before pointer manipulations [41], but these costs are paid regardless of the frequency of collections.

6.2.1 Variances from the real collector

In evaluating and building my model, I target OpenJDK's default parallel collector. OpenJDK, or java language runtimes that share most code with it like Oracle's JDK, are by far the most popular runtime for Spark. The parallel collector was the default garbage collector until OpenJDK 9 and remains the recommendation for applications where throughput is most important. Other Java implementations, including IBM's JDK and Jikes have similar collectors, and I believe the model can capture a wider range of common collectors with some modifications.

This model, as described above, matches the behavior of OpenJDK's default collector with heap size configured with some notable exceptions. For a small number of these exceptions, I make minor adjustments to my predictions to compensate.

Survivor spaces. Survivor spaces, a common feature of generational collectors including OpenJDK's, are not modeled. These are extra memory regions which are used as a 'second-chance' after eden space. The survivor space is divided into a *from* and *to* region; only the 'from' region is used between garbage collections, and it is sorted to track the age (number of survived collections) of objects.

In place of eden collections, OpenJDK has *minor collections*. During a minor collection, live objects in the eden and from space are marked; then, marked objects are copied to the 'to' space unless the it is full or the objects exceed an age threshold. In the latter cases, the objects are promoted to the tenured space instead.

There are several options for dealing with survivor spaces with my model. The easiest solution is to simply disable them by fixing the age threshold to 0. As will be shown in Section 6.7.3, they seemed surprisingly unhelpful on Spark programs I examined, so this seems to be a reasonable choice. A second solution is to pessimistically assume the addition of survivor spaces will have a small negative effect — perhaps doubling the eden collection time by requiring an extra copy assuming the 'second chance' is never helpful. A third solution would be to extend the model to estimate how effective the survivor space would be at preventing promotions to the tenured space, as I will discuss in Section 8.3.1.

Part of OpenJDK's implementation of survivor spaces is that, by default, one does not configure the size of the eden space, but sets the total size of the eden and survivor spaces put together. There are several options for accounting for this difference without modeling survivor spaces themselves: one would be to assume that OpenJDK's 'ergonomics' [88] for setting eden and survivor space sizes will arise on a solution at least as good as the performance obtained with something close to the maximum eden space. More conservative option would be to model the eden space as having the minimum possible size — one-third of the total 'new generation' space — or to arbitrarily fix the eden and survivor space sizes.

Allocations out of eden. My model assumes that all allocations start in eden space, but there are some exceptions. Allocations larger than the size of the eden space must be allocated elsewhere, probably in the tenured space. Even if the allocations would fit, assuming that very large objects will live longer than the next eden collection is likely to be a good heuristic. I account for these allocations by counting them as part of the amount promoted from eden space to tenured space, to compute the correct number of full collections. I do not model any changes to which allocations meet the threshold as configuration parameters change, which could cause misestimates of the number of eden collections.Generally, however, for the programs I examined, I found that I needed to select extremely small and thus poorly performing eden sizes to noticeably change this behavior.

Forced full collections. My model assumes that full collections are only triggered when tenured space is filled, but OpenJDK triggers full collections in some other circumstances. If, based on the recent amounts 'promoted' to tenured space during an eden collection, OpenJDK estimates that an eden collection would immediately trigger a full collection, and so proceeds directly to a full collection. Ordinarily, this is faster, but measurements of recent promotions may not reflect what would occur if eden collections were attempted. For programs which have an initial phase where they load data, almost all allocations during this phase to be promoted to eden space. If the loaded data fills up enough of the tenured space relative to the size of the eden space, then those large promotions will cause future eden collections to be skipped. Since the runtime will never attempt another eden collection, it might never fix its prediction of the amount promoted. The result will be that where my model would predict frequent but fast eden collections, there would actually be frequent and slow full collections. I can avoid this by requiring there to be enough extra space for the (typically small) eden space free in the tenured space.

Another way full collections can be triggerd because of metadata for classes. These are allocated in a separate region because they rarely need to be garbage collected. For recent OpenJDK versions, the rare cases where this memory region ('MetaSpace') fills up, a full collection is triggered. A program creates many new classes (not just objects of existing classes) can trigger this. I saw this with some prior versions of Spark that did not reuse temporary classes constructed when serializing closures as much as it should have. To control this effect, I record the total amount allocated in the space. Based on this total, I can estimate the number of forced collections for a given size; then I can set the minimum size for this region to ensure the number of forced collections will be similar to the number of full collections triggered by normal allocations.

6.3 Measuring Program Behavior

A program's demands on the garbage collector can be characterized by the order in which objects are allocated, their sizes, and their lifetimes. When the garbage collector only does work in response to allocations, as my model assumes, this trace of allocations completely determines its behavior and overhead. Most simply, one could collect this full trace directly from a program and run a simulation of the garbage collector, tracking the location of each allocated object after each additional allocation.

A full trace of object allocations and lifetimes is much too detailed. Many billions of objects may be allocated even in quite trivial programs that, for example, create an object to represent each line of input. Even neglecting the size of trace, collecting the exact information on lifetime is particularly difficult. Either one needs to bound the object's lifetime using program analysis or triggering garbage collections to obtain reasonable bounds. Since useful predictions must survive minor changes to the order of program execution (from multithreading, difference in I/O performance, minor changes to the calculations, etc.), such detail should not be so important. Instead, I will focus on using approximations that can use the bounds from observing normal garbage collection activity.

To make this possible, I make some simplifying assumptions. First, I try to capture the 'average' program behavior rather than modeling dependencies within the program explicitly. This will generally take the following form: when I require some statistic derived from object size and lifetime distribution, When possible, I measure the average value from a representative run. When this would not produce reasonable results, I instead use the empirical distribution of that value. The resulting model will assume that whenever a measurement is needed, it can be selected independently from that distribution when using it to make a prediction.

This independence assumption may neglects some common program behaviors. For example, if a greater proportion of objects is promoted to tenured space at the times when the tenured space is small — a likely result of periods of 'loading' — I would not account for this correlation. As a result, I might assume the average portion of objects promoted to tenured space implies a higher rate of full collections than it really does. Similarly, if there were a correlation of the opposite type, I would underestimate the rate of full collections. Fortunately, my evaluation shows that, after capturing the statistics I have chosen, these effects do not substantially effect predictions over a variety of programs.

To estimate the number of times garbage collections are triggered, the statistics I will measure are: the total number of allocated bytes (Section 6.5.1), the average number of promoted bytes for each possible eden space size (Section 6.5.2), and the empirical distribution of total bytes live (Section 6.5.3). For the first two, the value I am measuring is a mere average; for the latter, I found it necessary to capture the distribution to achieve reasonable accuracy as described in Section 6.3.1. Both the promoted sizes and total live sizes can reflect when there are clusters of related object allocations, so these summaries capture more than simply the distribution of live objects.

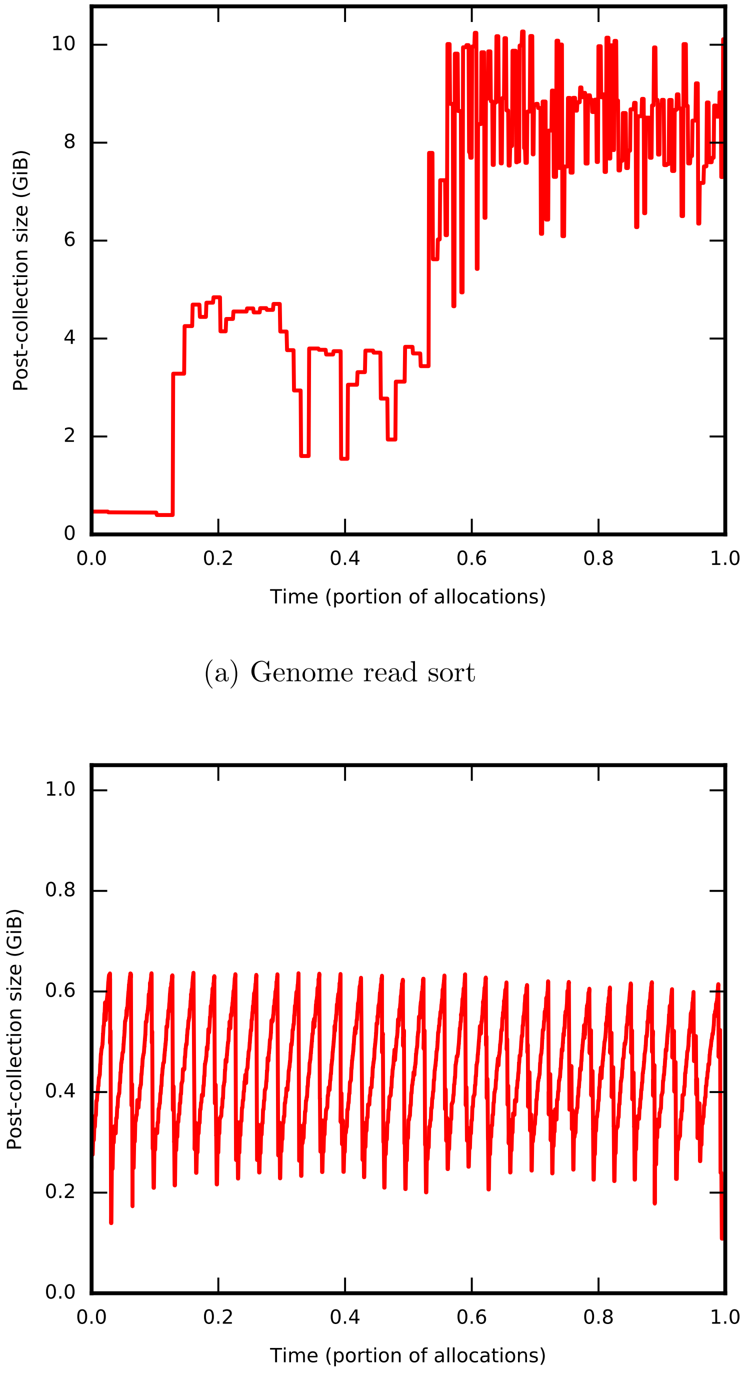
To estimate the total CPU overhead of garbage collections, I collect measurements of the CPU time of garbage collections and corresponding sizes and perform a linear regression as described in Section 6.6. This regression capture the relationships between garbage collection time and the mix objects being processed by the collector.

6.3.1 Phased Behavior of Programs

A major risk in making predictions from averages is that program behavior varies over time. Correlations make combining these averages risky. For example, if a program places more pressure on the garbage collector in the form of promotions to tenured space during the times when tenured space is tightest, then this will likely have worse performance than if they are uncorrelated. And averages are not often not a good summary; for example, the maximum size occupied determines the minimum space needed, not simply average.

Figure 6.2 shows some examples of different ‘phases’ of behavior with respect to the total memory. These measurements (on the y-axis) are obtained by plotting the size of a full collection. The time (x) axis is in terms of the number of bytes allocated in total (rather than the portion of old collections that occurred), which makes the resulting shape not change significantly as the tenured space size changes. To obtain a sufficiently detailed plot, the tenured space size is set to a small value to force many full collections.

These two plots show two different kinds of ‘phased’ behavior. In the case of the ADAM conversion program, none of the input needs to be kept in memory persistently, so the program loads a chunk of the input, processes it, and writes out a converted version, causing the cyclic memory usage shown in the figure. In contrast, the sorting operation has a distinct ‘loading’ phase and processing phase. The existence of useful programs, like the conversion program, with large, fast variation in the amount of live objects suggests that an approach based of identifying phases and modeling each phase as if it were a separate program would not always be effective.



(a) Genome read sort

(b) BAM to ADAM conversion

Figure 6.2: Estimate of total live object size over time. Time in this estimate is measured based on the approximate number of bytes allocated in total in the program, and the size estimates are based on the number of live bytes after a full garbage collection.

6.4 Instrumentation Logistics and Modifications

The language runtime most commonly used for Apache Spark programs, OpenJDK, includes several ways to obtain statistics from its garbage collector. These include the Java Management Extensions (JMX), a standard Java interface targeted at enabling monitoring, especially remote monitoring, of services; the JVM Tool Interface (JVMTI), a standard Java interface targeted primarily at profiling and debugging tools; an internal 'jvmstat' interface, exposed through the 'jstat' tool; and human-readable logs of garbage collector activity. Each of these interfaces does provide slightly different capabilities than any other, so my tool relied on a combination to collect its measurements.

Primarily, SLAMR uses measurements of each garbage collection pass, to record various sizes and overheads from that garbage collection pass. To trigger these measurements, JMX provides an ability to run code after each garbage collection operation. While JMX provides information about garbage collector region usage, it does not indicate the size of promotions into tenured space. It also does not make it easy to identify if, due to high CPU load, multiple garbage collections occurred since a measurement was triggered, so counters of garbage collection time reflect extra collections.

Fortunately, the jvmstat interface does provide easy access to this information, providing fast access to the last amount promoted and counts of the total number of garbage collections of each type since program startup. Missing from the jvmstat counters are sizes before a garbage collection occurred. This information is recorded in the human-readable logs, but I modified my version of OpenJDK to record the pre-collection eden space size as a new jvmstat counter to make measurement simpler. A less precise alternative would be to poll the counter for the current generation size to approximate the pre-collection size.

The most invasive instrumentation I required was to vary the effective eden space size, as described in Section 6.5.2.2. To do this efficiently, I required support to allocate an uninitialized array or to directly support changing the eden size, which I added to OpenJDK. Without such a modification, a less precise alternative is to allocate a normal array, at the cost of the substantially increased overhead from the compute time spent initializing it.

6.5 Counting Garbage Collections

The most important prediction of my model is estimating the number of garbage collections of each type. This is more important than estimates of the amount of CPU overhead during each of these garbage collection because of how the garbage collector behaves in constrained memory situations. When garbage collector performance is poor, it is likely because of an explosion in the number of collections, and not any change in the overhead of each collection.

As the tenured space becomes more and more constrained, the number of required full collections increases — potentially approaching the total number of objects allocated as almost every allocation could force a full collection. Similarly, as eden space becomes more and more constrained, the number of eden collections increases up to essentially the total number of objects allocated. Also, the number of full collections increases as the portion of objects promoted to tenured space increases. In the case of full collections, the total overhead per collection is essentially constant, as discussed elsewhere, so predicting the number of collections is by far the most important. For eden collections, there is a strong relationship between the eden space size and the cost of the collection, but the worst situation still occurs when eden space is very constrained and fixed per-collection costs dominate.

6.5.1 Measuring Total Allocations

Ultimately, SLAMR estimates the total amount of garbage collector overhead to make configuration decisions. Although it models 'average' behavior, it does not capture a throughput (overhead to useful computation) ratio. Instead, it measures the total amount of garbage collector activity of the program and eventually compare this to the observed non-garbage collector runtime (see Chapter 7) to determine if that overhead is reasonable. To measure the total amount of garbage collector activity, it estimates the total number of allocations by counting eden collections. To a first approximation, this is merely the number of eden collections that occur times the number of bytes in the eden space. There are likely to be some differences, however. I investigated several of these to determine whether adjustments were required.

One major concern is that the allocations would not fill up the entire eden space. When objects are large relative the eden space size, eden collections may be triggered when a substantial amount of the eden space is free. To account for this, I can examine the size of the eden space before collections instead of use the total capacity of the eden space. Generally, however, I found that the difference between assuming the entire eden space was filled and using the actual amount filled was negligible for the programs I examined, except in configurations with very small, poorly performing eden space sizes.

Another concern is that some allocations skip the eden space entirely. I do not attempt to adjust the total allocations measurement for this effect. I do include these allocations in my measurement estimate of promotions (Section 6.5.2), to avoid biasing the estimated number of full collections. Based on the lack of variation in the total allocation measurements seen in Figure 6.4 across a wide range of eden space sizes, this appears to be a reasonable simplification for the programs I examined..

When memory is tight in an instrumented run, a significant number of full collections may be triggered without an eden collection occurring. To adjust for this, I count full collections and assume that space corresponding the eden size was allocated to trigger each, representing the case where the collection was triggered because an eden collection was skipped. This is not always the cause of full collections, accounting for the distortion for small tenured space sizes in Figure 6.3.

After this adjustment, the measurement remains approximately constant even over variations in tenured space size (see Figure 6.3) and eden space size (see Figure 6.4). Thus, it seems safe to use it predict behavior across a wide range of configurations.

6.5.2 Estimating Promotions From Eden

The amounts of objects promoted from eden space to tenured space is probably the most important input to the model. These are dependent on both the program being run (its mix of object lifetimes) and the garbage collector configuration — most notably, on the size of the eden space. My goal is to instrument an execution of a program to determine its promotion behavior and how that behavior varies with eden size.

The most simple model of promotion behavior would be to take the generational assumption to the extreme — assume objects either stay in memory forever or are released almost immediately. If program behaviors were this binary, then a reasonable way to estimate the amount promoted would be to observe how much is promoted in some run of the program. Program behavior is often more subtle as shown in Figure 6.5; some of the programs exhibit the straight line one expects from this simple model and some do not. In particular, the assumption of 'binary' behavior is likely to break for very small or very large eden sizes — two extremes that are otherwise likely recommendations depending on the relative cost of eden and full collections.

Figure 6.5 shows the results of reference measurements — the circles — running the same program with varying eden size and measuring the average promoted. The solid line shows an example of the approximate measurement of this behavior I obtain by sampling over an single execution.

6.5.2.1 Safety of Interpolation

In measuring the amount promoted for many eden sizes, I want to be able to interpolate between individual measurements and extrapolate to unmeasured eden size ranges to make predictions. In doing so, I will use the assumption that short-term variation in collection behavior can be ignored.

One could construct a program that, if run at eden size N, each eden collection would occur immediately after the lifetime of all allocated objects. If one changed this program to start by allocating $N - 1$ bytes of useless objects, then each eden collection would instead promote almost all objects. Although possible, these pathological cases seem essentially impossible to trigger in practice; instead, real behavior is more continuous. So, I approximate behavior by assuming that eden collections behave the same throughput the program: the amount promoted for eden size N is simply the average portion of objects still alive over all possible N-unit window of allocations. This is essentially the 'steady-state' model of program behavior that motivates the control-loop model in [88]. The result of this assumption is that amount of objects promoted changes smoothly as eden sizes changes, so if it holds, one can interpolate between measured portions promoted.

This assumption also bounds the amount promoted even for values larger than those measured. The portion promoted must decrease monotonically as the size of the eden space increases. The amount promoted with some eden size N is the portion of bytes allocated that survive for $(N - 1)$ bytes of allocation, plus the portion that survive $(N - 2)$ and so on. The average amount promoted is thus some weighted average of the portion of bytes surviving for at most N bytes of allocation. For an eden size $N + \Delta$, the difference in the amount promoted are terms for allocations surviving $(N + 1)$, $(N + 2)$, . . . , $(N + \Delta)$ bytes. Thus, the average portion promoted for size $N + \Delta$ is the weighted average of the average portion promoted for eden size N and the portion of allocations that survive for strictly more than N bytes of allocations. Since the portion of allocations surviving more than N bytes of allocations is no more than that the portion surviving for N or less, this weighted average must be no more than the portion promoted for size N.

This also provides a test that may indicate if the this assumption of continuity is dangerously wrong. Namely, if there are significant decreases in the portion of objects promoted as eden sizes increase, this would be evidence that the synchronization of eden collections with object lifetimes was important. Empirical measurements of my example programs did not uncover such contradictory behavior.

6.5.2.2 Measuring Multiple Eden Sizes

The easiest way to estimate the amount that will be promoted for a range of possible eden sizes is to run a program several times with a variety of eden sizes. This requires many sample executions to obtain reasonable estimates, some at very inefficient (small) eden sizes.

To avoid this, instead, I sample multiple different eden sizes from one run. Typical garbage collectors do not include an interface for changing the eden size without restarting the program. Instead, I simulate this effect, by setting the eden size larger than the maximum size to sample and allocate space after each eden collection to shrink the available eden space. To minimize overhead, I allocate a single array of non-references and immediately make it unreachable. The unreachable array will not be scanned or copied by the garbage collector and the allocation itself only requires incrementing a pointer, so it ought to have essentially no effect on performance.

While the array itself imposes almost no garbage collector overhead, its initialization is not free. In Java, per the language specification, arrays start initialized to all 0s (or nulls in the case of an array of references), and this initialization can require a non-trivial amount of time. To avoid this overhead, I made a small JVM modification: support for allocating an uninitialized array. In OpenJDK, this already existed internally for use by the optimizer (in case it can prove that an array being uninitialized will have no effect); I merely made this functionality accessible externally.

Given a way to change the eden size, one policy would be to sweep across the range of eden sizes as the run completes. But, because of the existence of ‘phases’ of garbage collector behavior, this would be unwise; such sweeping would be likely to sample unevenly from each phase. To avoid this, I sample randomly from the range of eden sizes. To avoid getting correlated samples for any given size, I choose a new random size after each eden collection.

One concern with this measurement technique is that a garbage collection could be triggered while the dummy array was reachable. In this case, it may be promoted from the eden space, significantly biasing our measurements. Because real programs have ‘loading’ phases with substantially larger promotions than the rest of a program, simply discarding ‘outlier’ promotions is not an acceptable technique to remove this bias. Instead, I avoid this by allocating at least half the eden space. Thus, if the‘promoted space is ever big enough to hold the array, then the measurement is invalid, so such mismeasurements can be reliably detected at the cost of allocating more eden space. In practice, such invalid measurements are very rare.

A safer, more invasive option would be to modify the JVM to support actually resizing the eden space. OpenJDK’s default garbage collector supports this internally, which it uses for a mode where it adjusts the eden size via a control loop. To check the accuracy of array allocation technique, I compared it to using this internal interface. I compared the resulting measurements using a two-dimensional variant of a Kolmogorov-Smirnov test [69], and it did not show the two distributions were significantly different ($p > 0.5$). Modifying the garbage collector is a less general technique because it requires region resizing support a garbage collector might reasonably omit, as opposed to merely support for introspection of sizes and occurrences of garbage collection that at least Java standardized has felt could be universal across Java implementations.

6.5.2.3 Range of Measurements

After obtaining samples of the amount promoted at varying eden sizes, it is necessary to interpolate this into a function mapping eden sizes to the amount promoted. To do this, I take a sliding window of a 10% portion of all the measurements and use that as a lower-bound for the portion of items promoted at the high end of the window. I constrain this measurement to be monotonically increasing, taking a running maximum of the amount promoted in the window and amount promoted in all windows for smaller eden sizes. Assuming there are enough samples in the window to be representative of all phases of the program being studied this should be a conservative estimate.

It would be ideal to obtain some confidence interval on average amount promoted and some firm guidance about suitable window sizes. Since the phase-based program behavior implies that individual measurements are *not* independent, it is difficult to see how to obtain a reasonable bound. Certainly, given multiple sample runs, one could do this. Each run would give an independent measurement of the promotions at each eden size, so obtaining a sensible confidence would require a large number of sample runs.

Vertical jumps seen for some programs in the measured lines in Figure 6.5 illustrate some of the loss of precision from short-term variations in object lifetimes. For some programs, there are brief phases when many times a normal amount is promoted from eden — such as when data is being loaded. When there are relatively few eden collections, the window containing these exceptionally large samples has a large jump. Based on the interpolation rule, all subsequent measurements are constrained to this minimum portion. Generally, this distortion does not seem to result in unsafe (too low) promotion predictions because the sliding window was generous enough to hide to the distortion.

One concern with these measurements is whether they produce consistent results when one measures samples very different ranges of eden sizes, or if the measurements are only predictive over a small range of eden sizes. Figure 6.6c shows promotion estimates obtained from runs with an approximately factor of ten difference in sampled eden size range (lines) and actual measurements of promotions at different eden sizes for those programs (dots). This factor of ten variation in the in the range of eden sizes measured produces measurements at most around two or three times too conservative. And, over most of the range of the measurement from the smaller eden size, the values from the larger measurement were too conservative by much less than 50%. Given this, even if one starts from a very poor configuration guess, it is reasonable to believe that using one instrumented run to choose the next instrumented run should quickly converge to well-informed configuration.

6.5.3 Measuring Live Bytes

To use promotions from eden to predict when full collections must occur, one must predict how much of the tenured space these promotions have to fill between full collections. In my model, this requires measuring the amount of free space after a tenured collection. Like for promotions, I rely on obtaining samples of these sizes from a example run.

At any time during a program, there is some definite amount of objects that are live. My tool attempts to capture this as a function of time — with time measured in terms of the number of allocations that have occurred. For a given trace of allocations, this value should be the same regardless of the actual garbage collector parameters.

The only reliable way of which I am aware to obtain a good measurement of the number of live objects is to observe a full garbage collection. I simply interpolate between whichever of these measurements I observe. This interpolation produces a result like those shown in Figure 6.2.

Rather than attempt to simulate full collections over time, I assume that full collections are approximately uniformly distributed over the program's lifetime. This allows the timeline of live bytes to be summarized as an empirical CDF like those shown in Figure 6.8. Given a particular tenured space size, the number of eden collections is the amount promoted per eden collection divided by the space remaining. I derive a distribution for the computed number of eden collections from the distribution of live bytes. Taking the mean of this new distribution gives the expected number of full collections per eden collection.

Figure 6.7 shows how these predictions differ from the simpler approaches of only using the mean or maximum live bytes. The curved line in each graph is the effective amount live after a full collection using the distribution-based model — that is, the constant live objects size that would produce the same result. Alongside it, the maximum (dashed lines) and mean (straight, solid lines) amounts of live objects are plotted. For programs that have substantial variation in their memory demand, using the full distribution interpolates between assuming performance is dominated by the maximum memory usage (for small capacities) and by the mean memory usage (for large capacities).

Unlike eden collections, it is not uncommon for a program execution to have very few full collections. Also, unlike measurements of promotions, these measurements have unlimited sensitivity to outliers: the maximum observation determines the minimum size with finite estimated overhead. For promotion sizes, in contrast, adding a single possible observation will only change estimates by a bounded amount. With few collections and high sensitivity, there is risk that given few samples, one may predict a configuration has low overhead when it would, in fact, result in failure.

As long as program's memory usage is dominated by memory managed by the framework, following the framework's size estimates can constrain this problem. For programs that take advantage of the framework well enough, either the program will keep little data in memory or almost all of the memory of the program will be in memory. Thus, absent samples from actual collections, the maximum amount live will be bounded by the framework's size in memory is in memory plus some small factor for unmanaged data. In my evaluation, I primarily rely on this heuristic to keep my configurations safe.

Without trusted framework estimates, one can instead obtain more information by explicitly forcing extra full garbage collections to happen, which will reduce the maximum amount used, allowing one to conservatively configure a smaller tenured space size after obtaining these observations. These forced collections may add substantial overhead, but if one triggers the collections above a certain space threshold, one can guarantee a reduction in the memory configurations. Less conservatively, one could assume that, when starting from a program with few full collections relative to the amount of allocations which is not extremely garbage collector friendly, at least some small portion of memory will be garbage, perhaps 5%, which would permit one to immediately lower the memory allocation.

In practice, with more than around five full collections, I have not observed cases where the maximum amount observed live is not close to the amount required (based on what tenured sizes complete successfully), even without any adjustments for framework estimated memory usage. This makes sense for a typical batch analytics program as one expects a majority of the program's time to be spent with the working set loaded.

The stability of the measured amount live with different numbers of collections can be seen in Figure 6.9. This figure shows CDFs for measurements of the amount live varying by about a factor of two in old generation. Generally, the instrumented run producing the dashed green curve is constructed from less than ten full collections, and the instrumented run producing the red curve is constructed from many tens of full collections. This shows that the bias of runs with few measurements is usually towards higher observations, probably leading more conservative estimates. Occasions when there is relatively little memory usage, times at which tenured space is less likely to fill up triggering a garbage collections, are what tends to be missed. In addition, in the cases where the maximum memory usage was 'missed' by having fewer full collection observations, the difference is not so large that a configuration that would fail is a likely recommendation.

6.6 Timing Garbage Collections

Although explosions in the sheer number of garbage collections dominate the bad cases in performance, it is important to have a reasonable estimate of the overhead they induce. The most important reason for this is that there are pathological estimates one might make otherwise. For example, a typical eden collection is fast — on the order of tens of milliseconds — and a typical full collection with a large heap is slow — often several seconds. But this has less to do with the differences the mechanisms of eden and full collections than the much smaller amount of data involved in eden collection.

A model which assumed constant time per eden collection would, given a some time corresponding to a small eden size, infer that having eden collections effectively substitute for full collections would produce much better performance when usually performance is substantially worse. Similarly, given a constant time from a large eden size, a model would drastically overestimate the overhead of smaller eden sizes and never suggest trying them. I do not attempt to do precise performance profiling, but I try to collect information that will correct for gross errors like this and permit more reasonable extrapolations.

In general, I take advantage of OpenJDK already recording the total time spent in garbage collection. Based on my understanding of the important operations within each type of garbage collection, I correlate this with other measurements over the observed executions with a linear regression to make rough predictions of the amount of time per garbage collection with other parameters.

To consolidate these predictions into a model, my goal is to predict the mean time per garbage collection, which is not the same as predicting the typical time per garbage collection. Garbage collection times tend to have a long-tailed distribution, which I wish to take into account — recognizing that they increase the total overhead.

6.6.1 Eden Collection Time

For eden collections, there are two tasks that should determine the collection time. One is spent moving promoted objects. For most programs, this seems to dominate garbage collection times, but if a program promotes very little from eden space, instead, the amount of time spent marking objects may dominate.

The amount of time spent for moving objects should be proportional to their sizes — because they will be copied — and the number of references to them that must be updated. Consequently, the time will vary based on the types of objects a program uses. Rather than attempting to measure the mix of types of objects for a program and feed that to a generic model of garbage collection performance, I measure the correlation between total sizes and times, assuming that they will take into account that program's mix of objects. This makes it possible to evaluate the model with information that is much easier to obtain from a Java virtual machine implementation.

In assuming this correlation is linear, I assume that the mix of object sizes will not change dramatically as the relevant configuration parameters change. The objects that survive eden collections for smaller eden sizes may be qualitatively different than those for much larger eden sizes. By not taking this into account, I am making an assumption about the representativeness of the instrumented run that trains the model: it is 'close enough' to the true runtime behavior of the program. My goal is that a user using the recommendations as a feedback loop — feeding instrumentation from one run to suggest the next instrumented run — will converge on a configuration of approximately equal cost no matter their starting point. Since I incorporate correct notions about the directions in which garbage collection time changes, I believe there will not be false minimas that would prevent this convergence.

6.6.1.1 Promotions and Time

Because of the cost of copying objects during eden collections, I estimate that the eden collection time is proportional to the number of bytes copied. From a single example run with many eden collections, one can capture examples of the variation in time when different amounts are promoted, especially if one is also varying the effective eden size to ensure a wide range of possible amounts promoted are covered. Figure 6.11 shows a density plot of this data for my example programs. From this data, a linear regression produces a simple model of the amount of time given predictions of promotions. As a modification to the regression, I enforced that garbage collections take at least the minimum amount of time ever observed during the example program. This avoids degenerate predictions where shrinking the young generation would apparently make garbage collections free.

The predictions from this per-garbage collection model are generally not as accurate as predictions obtained by running the program multiple times with different eden sizes, then training a linear model on the mean amounts promoted and garbage collection times over the entire program. The easily found correlations can be seen in Figure 6.10.

One likely reason is that different phases of the original program have both a different typical portion promoted and a different mix of types of objects. As a result, the regression using many points from a single run mixes the effects of varying amounts promoted with the effects of varying the phase being run. These effects of variation between program phases can be seen in Figure 6.11 in several ways. Many programs have periods with few promotions and consequently have exceptional density at very small promotion sizes. Some programs, like the ALS example, seem to have two distinct types of eden collection costs with their own straight regression lines — likely representing two different phases of behavior. A regression over whole-program averages hides these phase effects entirely, but requires much more data collection time.

6.6.1.2 Old Size and Eden Collection Time

For the time spent during eden collections, the size of the tenured space can matter. This can be seen empirically in the results in Figure 6.12. This is surprising because the number of objects to promote from eden space does not change substantially. There are two likely causes, both related to the card table where changed references in the tenured space are marked.

One step during an eden collection is scanning and clearing all the entries in the card table. This should typically be low overhead compared to the cost of copying live objects out of the eden space, but programs where the amount of data copied out of eden is very small are likely exceptions. In OpenJDK, there is one, one-byte card table entry for every 512 bytes of tenured space, so this scanning should be dominated by reading and writing around 0.2% of the tenured space size. For most example programs, the configurations determined by my tool result in substantially larger amounts being copied out of the eden space than the likely size of the card table.

I recorded how many objects were scanned during eden collections and found that, surprisingly, this increased somewhat when the tenured space size was increased. This suggests that the card table also became less efficient for larger heap sizes. Each card table entry represents multiple objects, but can only be marked as 'modified' or 'not modified'. Even though only one reference may have changed in a 512-byte region, all objects in that region are scanned. With less frequent compacting full collections, there is more fragmentation in the heap. Each full collection effectively defragments the heap, so the time since the last collection should determine how likely it is that an object is next to an inactive object in memory.

Whatever the cause of the change in time per garbage collection, empirically it appears from Figure 6.12 that a linear model is a good predictor of these effects. This linear model is straightforward to produce given samples of execution at multiple tenured space sizes.

Thus, I only model this effect when multiple samples are available. For the programs where this substantially effects accuracy — mostly ones where garbage collection overhead is generally low — this would start to compensate for the bias in predictions after two or three runs.

6.6.2 Full collection time

If full collections worked like eden collections, one might expect their costs predicted by the distribution of sizes of the and total number of live objects at each full collection. Since, unlike eden collections, full collections consider all live objects, the set marked should not vary much based on garbage collection parameters, but rather on the particular points of time sampled by collections will vary. Under the assumption that full collections are uniformly distributed over allocations, this would imply that the overhead per collection will not vary — but this is not what happens empirically.

The results in Figure 6.13 show that the amount of time required for a full collection varies based on the capacity of the tenured space. This does not occur for all programs. For example, the ALS recommender program and genomics sorting examples experienced no significant change in the amount of time per full collection as garbage collection region sizes were changed, but some other examples experienced much more significant changes.

Unlike the eden collection operation, however, the full collection operation is in-place. This means that copying might be avoided when objects were already compacted in a previous pass. A typical strategy scans over the entire heap to avoid problems with overlap between the old and new locations of objects and allow for easy management of pointer redirection metadata. These differences can explain an empirical observation of Figure 6.13 — the size of the tenured space has a significant influence on the amount of time a full garbage collection takes. Empirically, this difference appears to be approximately linear.

To model the time per tenured space collection, then, I use two approaches: when there is not data from a variety of tenured space sizes, I assume that the time per collection is constant. Since, generally, one will start with a larger tenured space size to produce a configuration with a smaller tenured space size, this should generally result in conservative estimates. When given observations of multiple tenured space sizes, I use a linear regression to estimate the time per old collection as a function of the tenured space size.

There are factors that suggest that the time per tenured space collection could instead decrease as heap sizes increases. There ought to be more garbage collections when the heap capacity is closer to its usage — based on the same logic by which I predict the number of tenured space collections. These garbage collections should require moving more objects. This effect would mean that the average amount used after garbage collections would be larger for smaller tenured space sizes. In practice, this change is not significant for the Spark programs I examined as can be seen in Figure 6.14. This is likely because any differences are too small to observe. For example, for a program which alternates between equally long 2GB-usage quiet periods and 5GB busy periods, full collections during the busy period would account for around 80% at a 6GB heap size and 66% at a 7GB heap size. In this case, the average space used after a full collection would increase by only around 10%. Not only is this hypothetical effect small, but the actual variation in live data sizes observed in Figure 6.7 is rarely so extreme.

6.7 Evaluation

To evaluate the garbage collector modeling predictions, I ran the example programs from the prior chapter on the same modified version of Spark 1.4.2. Since I make some assumptions about the regularity of program behavior, I tested across the same variety of analytics programs as described in Chapter 5. In addition to including a variety of types of framework usage, I believe these programs have variety in their garbage collector demands, including both cases where persistent data dominates and data processing is mostly temporary data and including both programs that represent their data mostly as primitive arrays and as many small objects.

I first choose the framework parameters — storage size, etc. — using the procedure in the previous chapter. Then, using these parameters, I choose a training configuration that provided at least 150% of the memory estimated to be required for framework usage in the tenured space, and an eden space one-sixth the size of total memory allocated. To handle cases where the Spark-managed memory usage was very small, I additionally required that this allocate at least 0.5 GB per core. When using multiple worker nodes to run the training program, I expanded the memory allocated to the JVM to fill left over space on each node — essentially 'rounding up' memory usage as one would do when training over multiple fixed-sized nodes.

For most runs, I disable use of survivor spaces, to make the configurations most closely match my garbage collector model. In the training run, I enabled the random sampling of the eden size, making the effective average eden size to around 1/24th of the total memory allocated. The default configuration for OpenJDK on Linux allocates around one-sixth of the total memory to eden and survivor spaces, so the effective size is less than the default ratios (even if the survivor spaces have the maximum size); however, for almost all Spark programs I have observed, the best performing garbage collector configuration appears to use a much smaller young generation than the default.

From this sample run, I inferred a base configuration by minimizing a utility metric: the estimated total runtime times amount of memory used. Starting from this base configuration, I performed a sensitivity analysis: I ran programs with eden space and tenured space sizes at leas 60%, 80%, 120%, 140% and 160% of the base configuration — as long as these sizes would fit in the memory available on the virtual machine in use.

When comparing predictions and actual observations, I use graphs like those in Figure 6.16. In these graphs, circles represent actual measurements. The solid green line represents conservative predictions and the dashed blue line represents less conservative predictions. The difference between the two predictions is whether the rule that the tenured space must always have enough space to store a copy of the entire eden space is in effect. The left of the graph contains regions marked with horizontal green lines and red vertical lines. The green markings represent regions that are predicted to cause crashes; the red markings represent regions where crashes or requiring more than a generous time limit (of at least twice the original runtime) occurred, if any such runs were attempted.

6.7.1 Number of collections

6.7.1.1 Eden collections

Since the number of eden collections are proportional to the amount allocated, and, as established in Section 6.5.1, this is very stable, predicting the number of eden collections given the size of the eden space is trivial. Figure 6.15 shows that such predictions have very high accuracy so long as the size of eden space is fixed.

6.7.1.2 Full collections

Predictions of full collections are not as straightforward as eden collections because they are sensitive to both the total allocations and promotions. Figure 6.16 shows predictions for the number of full collections (lines) versus actual observations as the tenured space size is varied. For my example programs, the conservative are consistently slightly higher than the actual observations, and the less conservative predictions are around exactly the value.

6.7.2 Time estimates

To evaluate estimates of individual garbage collection times, I compared the mean time per garbage collection of each kind, as measured by the JVM's performance counters, to my predictions of the time per cycle. In each case, the mean time I compare is the estimated mean time per collection from my model based on the corresponding linear regression model.

6.7.2.1 Eden collections

Figure 6.17 shows predictions of young collection times versus actual collection times from some example programs. These predictions are made based on the results of the single example configuration which usually has a much larger eden size than the instrumented run. As a consequence, the predictions are sometimes poor — off by perhaps a factor of two — but this can likely be corrected by referring to closer eden sizes.

6.7.2.2 Full collections

Obtaining useful predictions of the time per full GC requires instrumentation at different tenured space sizes. Figure 6.18 shows results using just two tenured space sizes. The resulting regression lines sometimes yield estimates of full collection time that would suggest negative collection times at small heap sizes. Presumably, this is a result of a combination of measurement error and non-linear effects (such as from memory locality). To prevent this from resulting in ridiculous predictions, I can fix the y-intercept at 0; the result of doing this is shown as the solid green line. The lower dotted blue line does not include this adjustment.

6.7.3 Effects of Survivor Space

Figures 6.19 and 6.20 show the difference between performance with survivor spaces enabled and disabled for several programs as garbage collector parameters were varied. Generally, I found that Spark programs had more GC overhead when survivor spaces were enabled. The likely reason is that these programs do not take good advantage of this extra region — either most objects are freed very soon or kept in memory for a long time. Consequently, the benefit of adding survivor spaces is freeing the few objects with short lifetimes which happened to be around at the time of the eden collection faster at the very substantial cost of copying long-lived objects a second time.

This is supported by an examination of the garbage collector counters from these runs. Based on the initial loading phase, the garbage collector often reached to a configuration where objects are promoted to the survivor space, but were then all copied to the tenured space in the next collection. This occurs rapidly, presumably because the 'data loading' phase

Program	Base	+ GC counters	+ Eden Vary (x1)	+ Eden Vary (x2)
EM-LDA	169 s ± 2 s	171 s ± 2 s	185 s ± 0 s	175 s ± 3 s
ALS	799 s ± 7 s	799 s ± 8 s	826 s ± 13 s	806 s ± 11 s
ADAM sort	4563 s ± 0 s	4568 s ± 30 s	4673 s ± 0 s	4558 s ± 10 s
Carat	2400 s ± 31 s	2495 s ± 124 s	2647 s ± 29 s	2502 s ± 7 s

Table 6.1: Runtime with various amounts of instrumentation enabled, showing mostly negligible runtime overhead. Instrumentation of eden collection decreasing the effective eden size, and this can easily have much larger effects on performance than my instrumentation. The difference in cost when increasing or not increasing the allocated eden space to compensate shown.

means that a majority of early allocations are for objects which will have long lifetimes. This easily exhausts the survivor space with any threshold that would hold objects from the tenured space for longer. The amount surviving tracks very closely with the amount promoted, showing that most objects copied to the survivor space are promoted immediately thereafter.

One can compensate for the tendency towards keeping items in survivor spaces only one collection by forcing the survivor space size to be larger. As shown in Figure 6.21, without tuning that fixed size, this results in about the same performance as disabling survivor spaces entirely.

6.7.4 Measurement Overheads

Unlike measurement of Spark, measurements of garbage collection overhead are far more invasive. These measurements require effort after each garbage collection and deliberately change eden collections to explore different, potentially more costly behavior.

To evaluate the measurement overhead, I selected a configuration for a program based on what minimized the predicted memory times time. This generally has more constrained eden space sizes than the default configuration and so will exhibit higher overhead than a typical training run. I measured the difference in time between running the program with only extra instrumentation of the framework (from the last chapter), with recording after GC, and finally, with the changes to the effective eden size discussed in section 6.5.2.2. Since a major cause of extra overhead for changing the eden size is from the additional garbage collector activity required, and not measurement overhead per se, I evaluated two options. In one, I increase the eden size by a factor of two to make the decrease in the average effective size of the eden space less significant, in the other I leave it the same.

The results are shown in Table 6.1. Each measurements is derived from three runs, and the mean and standard deviation are shown. The only significant (versus other variations in performance) overhead comes from the effect of the eden size measurement. This overhead is generally not more than 10%, and typically much less. However, the overhead can become very large if the instrumented runs use a very small eden size. We believe that this can be avoided by starting with larger configurations and using conservative estimates from this configuration to choose future runs. In addition, when choosing the configuration for a run with eden size instrumentation from the model, I choose an eden size large enough that the mean sampled size will match the target, not the maximum sampled size.

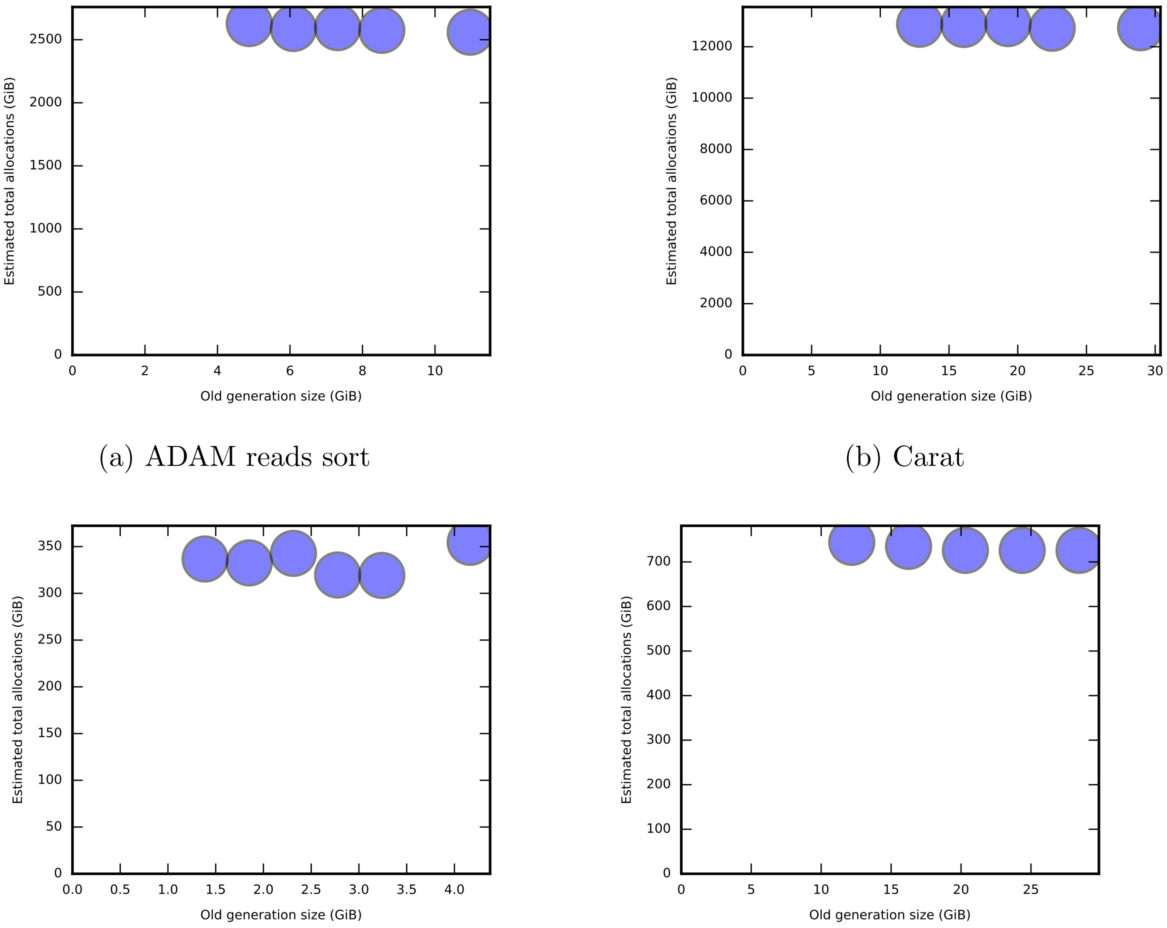
In addition to the time overhead, this sampling has space overhead from the additional size of the eden space. With the dummy allocation technique, sampling eden sizes from 0 to x requires $4x$ eden space. Alternately, with uniform sampling over the range, obtaining 'about' the performance of eden space y requires $2y$ eden space. Good configurations generally require less than 10% eden space, or the total memory required is very small (the program does not store much persistent data), so the overhead of this is usually less than 10%. In general, the largest eden space the JVM supports is around 33% of the total memory size, so the space overhead should not be much more than that. In general, it seems that the space overheads from starting with a more conservative than needed memory configuration would dominate even a 10-20% measurement overhead.

6.8 Conclusion

By making careful summaries of a program's garbage collection behavior from an example run, one can make reasonable estimates about its behavior in other configurations. Taking advantage of the desire to make throughput and not precise latency predictions and allowing the predictions to be wrong — as long as they are conservative — makes this task substantially similar. One simplification this enables is not trying to simulate the exact garbage collector, but instead simulating a simpler one. From these simplifications, I produce a model whose inputs are practical to collect with little extra overhead and whose outputs can be computed without doing something akin to a simulation of the program.

As a theme, the summaries I collect are single numbers, sizes and times per unit size, when that will suffice and distributions when it will not. For very important behavior of eden generation — the most efficient garbage collection pass, and one which feeds into all garbage collector passes — I collect many samples over a program. To predict how this behavior varies for different configuration parameters, I vary the configuration over these samples — choosing randomly to avoid correlation between configuration parameter and the 'phase' of the program. For other quantities, I primarily rely on one observation, and choose quantities that are independent of configuration changes. When this is not possible, the errors introduced tend to be very small compared to the 'performance breakdown' effects I correctly predict and I device techniques to eventually correct these given information from additional runs.

Generally, the model predicts garbage collector performance across a variety of configurations from examples of behavior under one or two configurations. I demonstrated that it produces good estimates across a variety of analytics programs, which impose a range of demands on the garbage collector. These predictions can be obtained from a single, naively configured program execution and are consistently conservative enough to allow them to be used to recommend configurations. Although I relied on some modifications and interfaces to obtain statistics to the underlying language runtime, these modifications and interfaces are trivial (allocating uninitialized objects, accessing sizes) and depend only on the most general aspects of the design of the language runtime's garbage collection system. Using this model and its corresponding instrumentation should be an effective way to find good garbage collector configurations — and because of careful choices about 'dangerous' behavior, conservative configurations — starting from trivial, likely wasteful configuration.



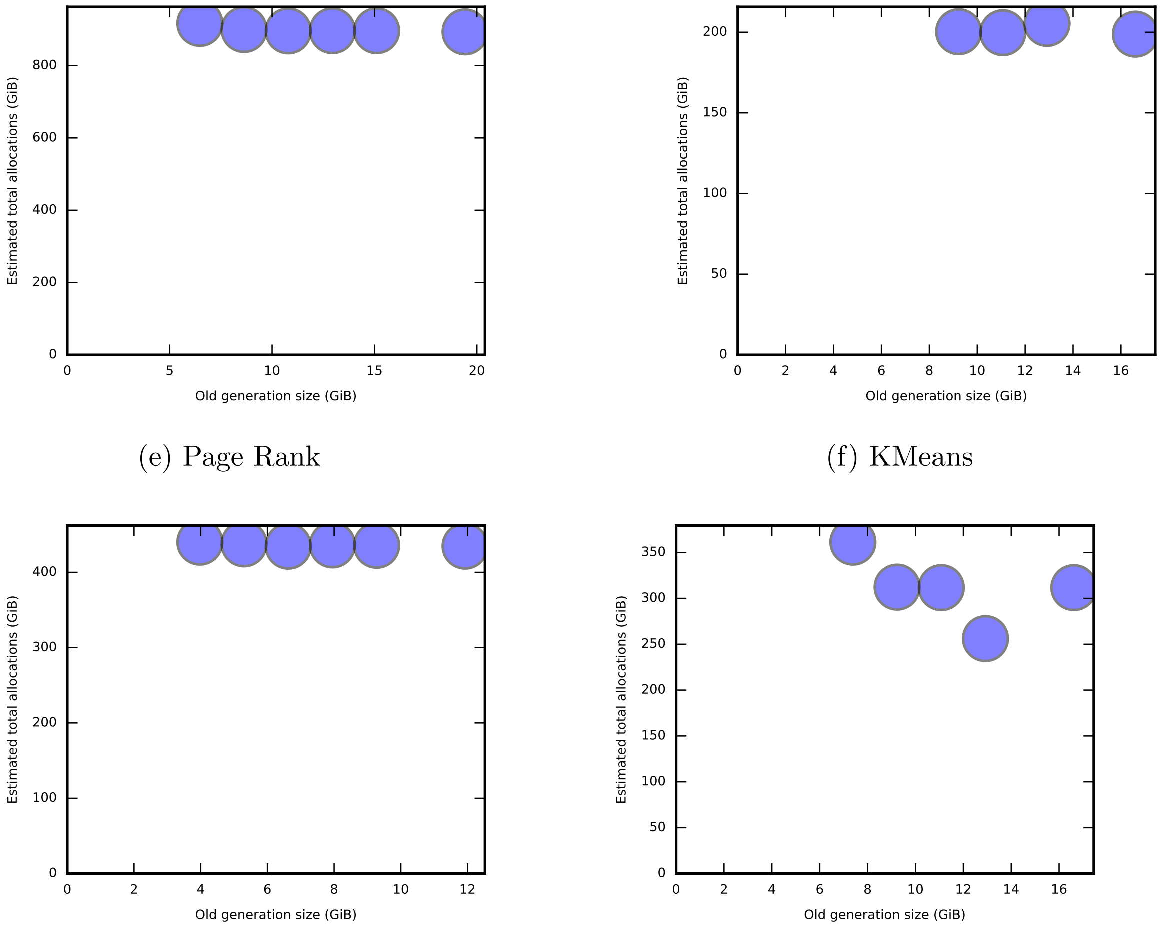
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.3: Measurement of total allocations across multiple tenured space sizes.



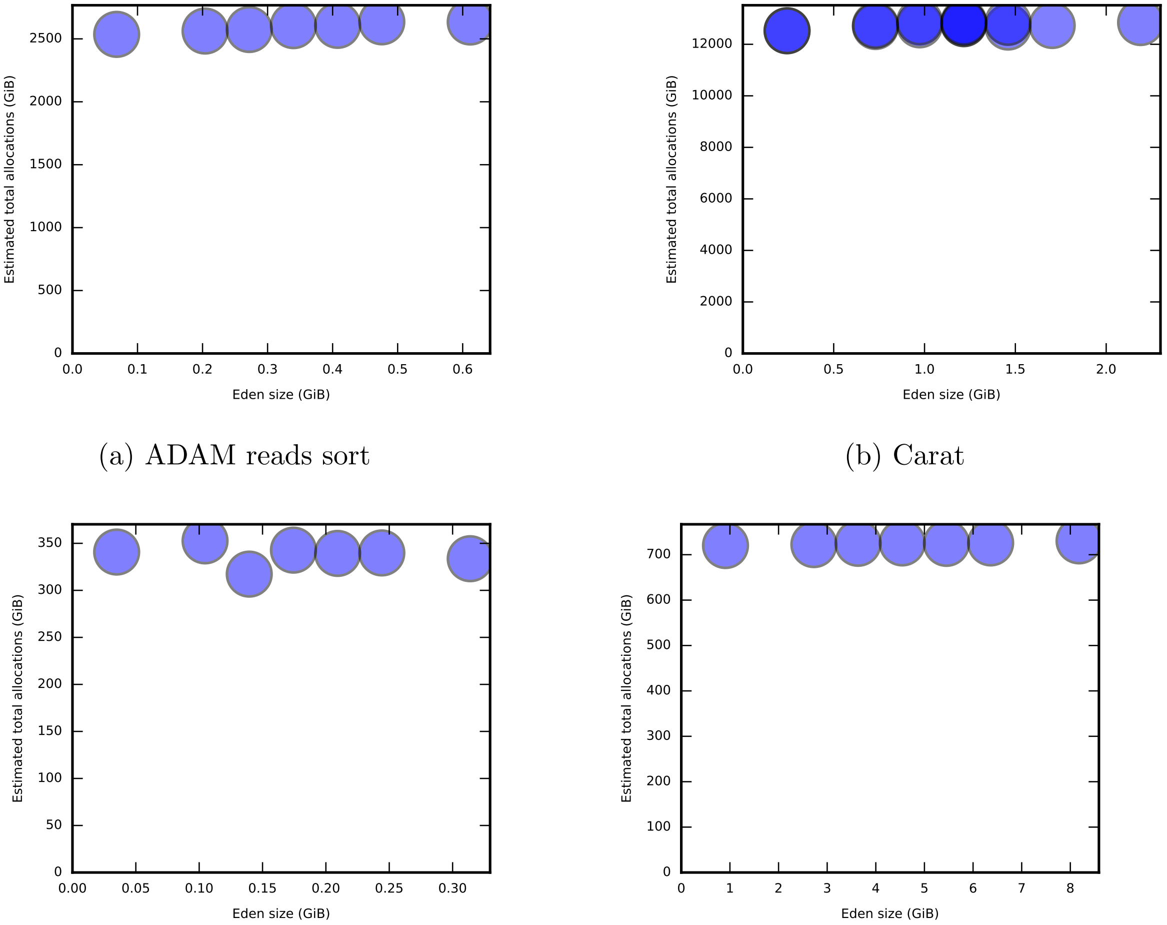
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.3 (con't): Measurement of total allocations across multiple tenured space sizes.



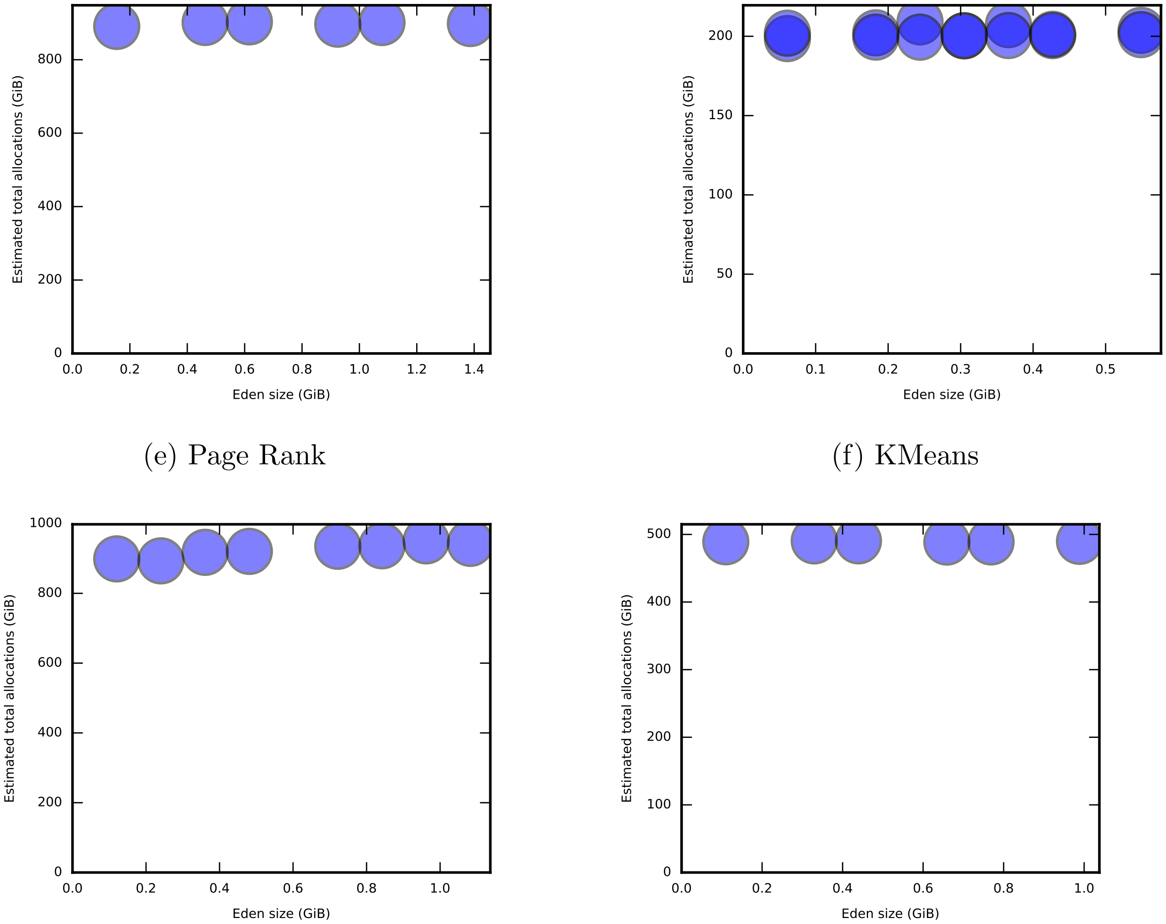
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.4: Measurement of total allocations across multiple eden sizes (with survivor space disabled).



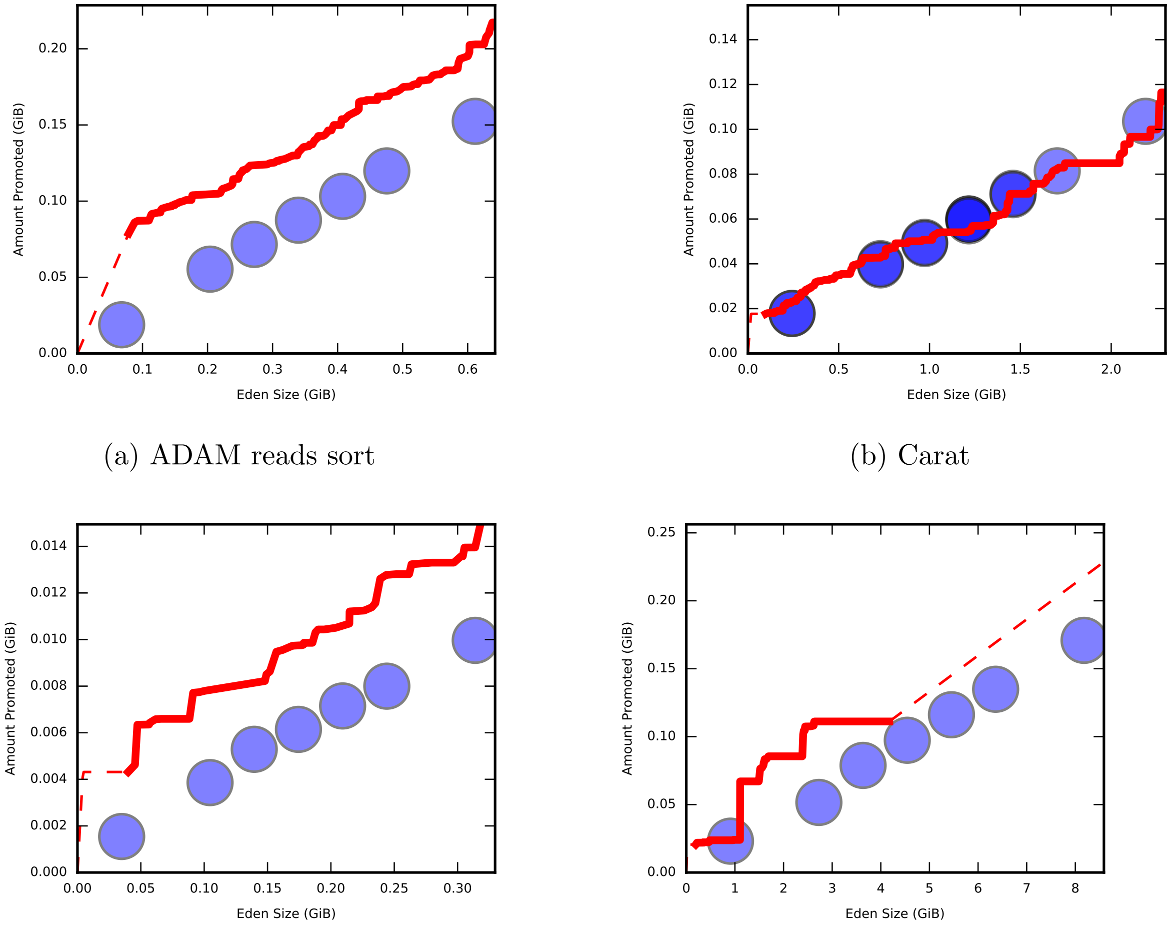
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.4 (con’t): Measurement of total allocations across multiple eden sizes (with survivor space disabled).



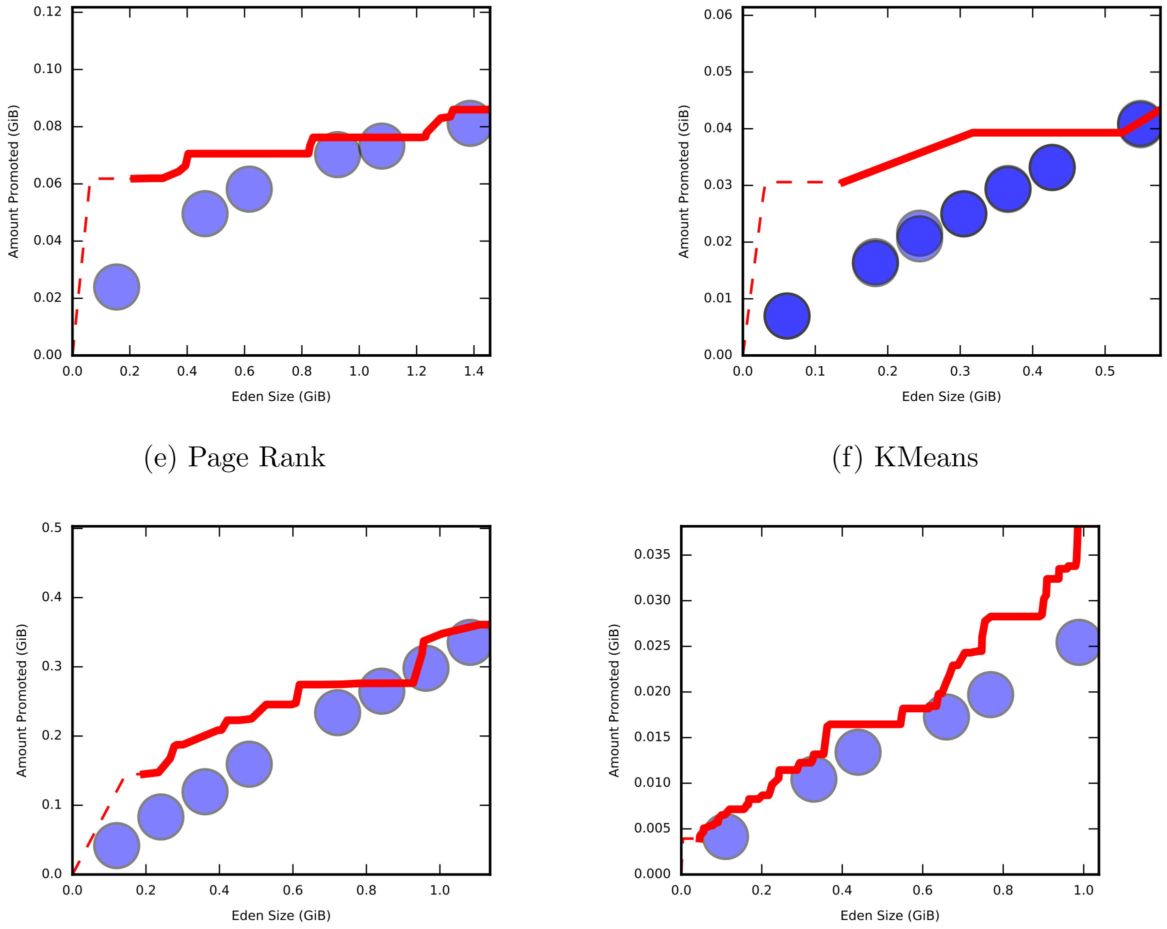
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.5: Single run measurement of total promoted (line, sliding window average) versus actual runs with fixed eden sizes (circles).



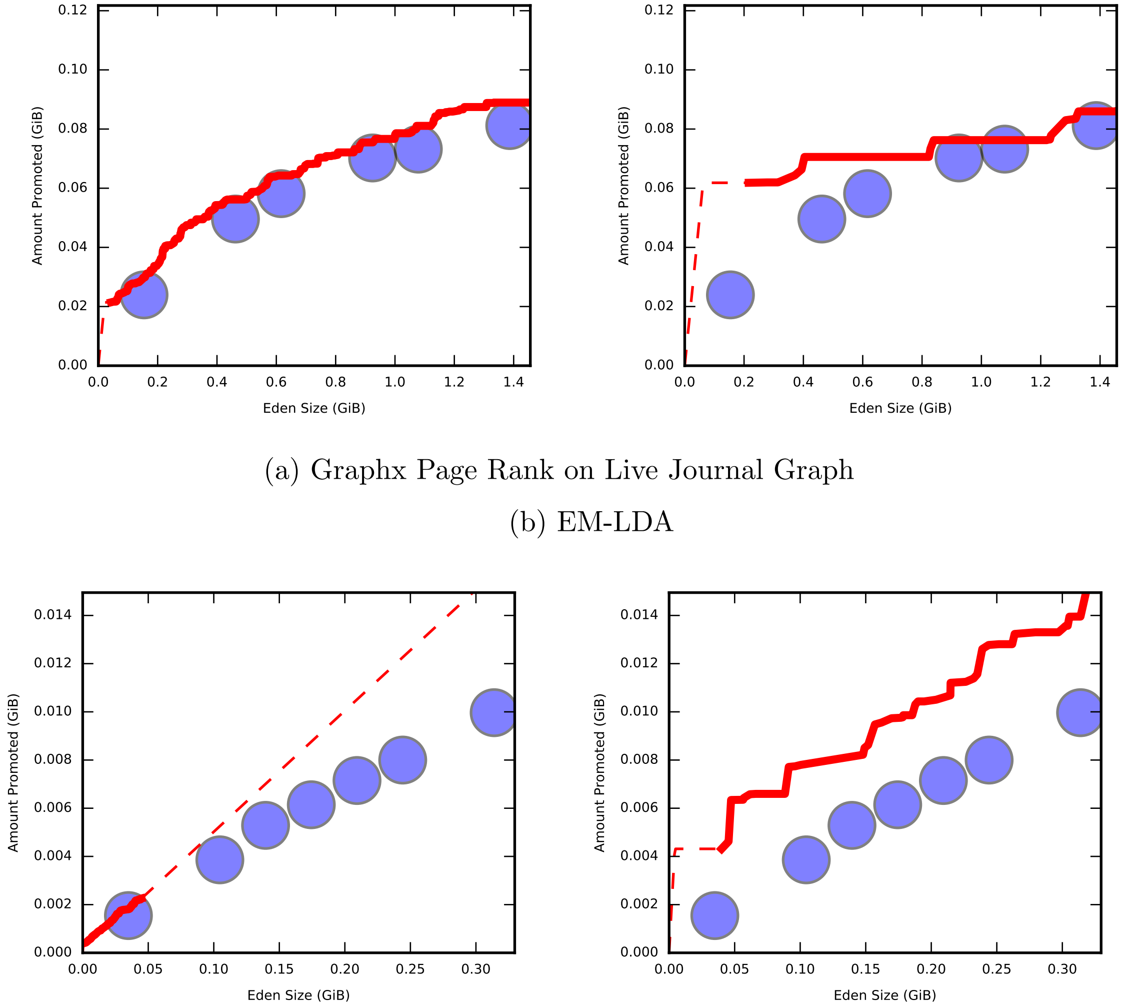
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

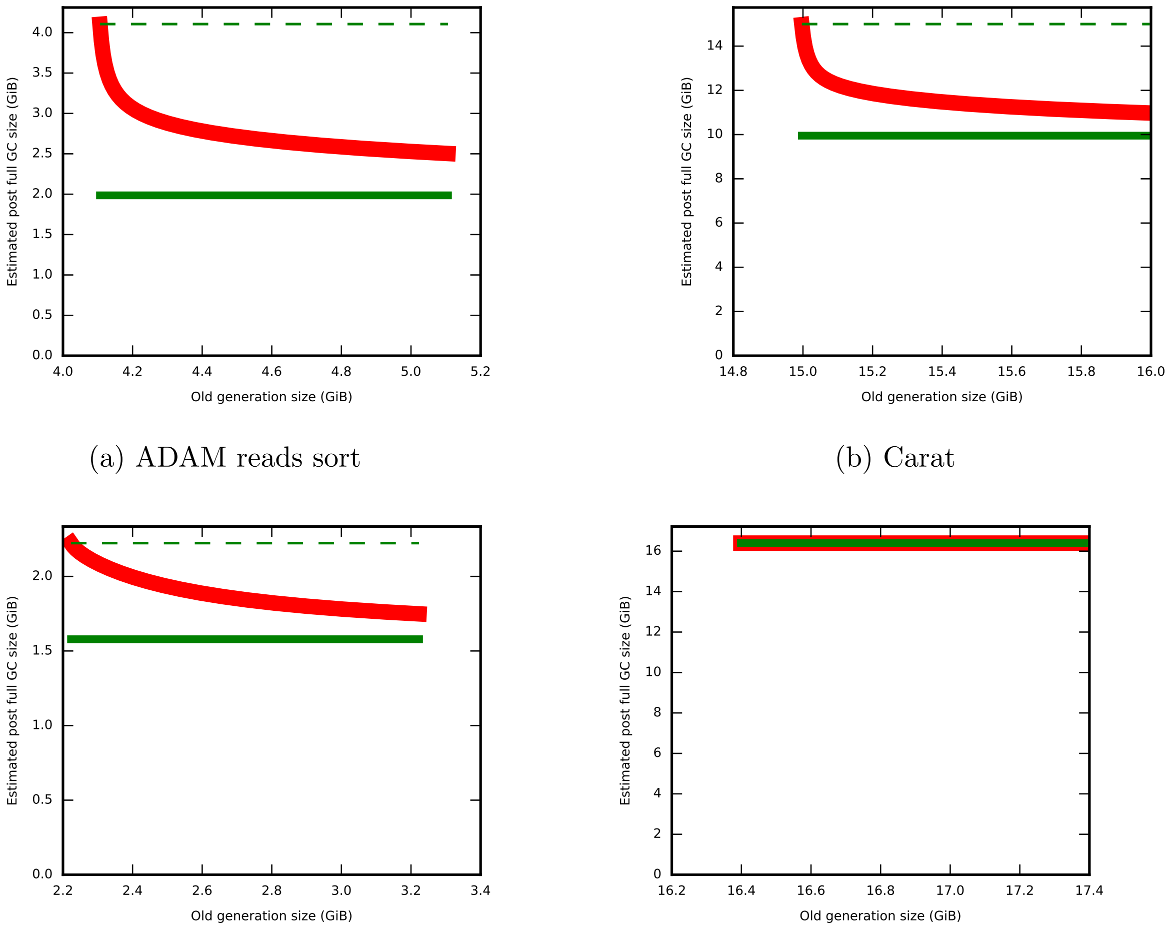
Figure 6.5 (con't): Single run measurement of total promoted (line, sliding window average) versus actual runs with fixed eden sizes (circles).



(a) Graphx Page Rank on Live Journal Graph

(b) EM-LDA

(c) Differences in the empirical recordings of amount promoted based on extreme eden sizes (lines) along with measurements from actual runs for comparisons. Graphs on the left represent recording using a small maximum eden size; on the right using a large maximum eden size.



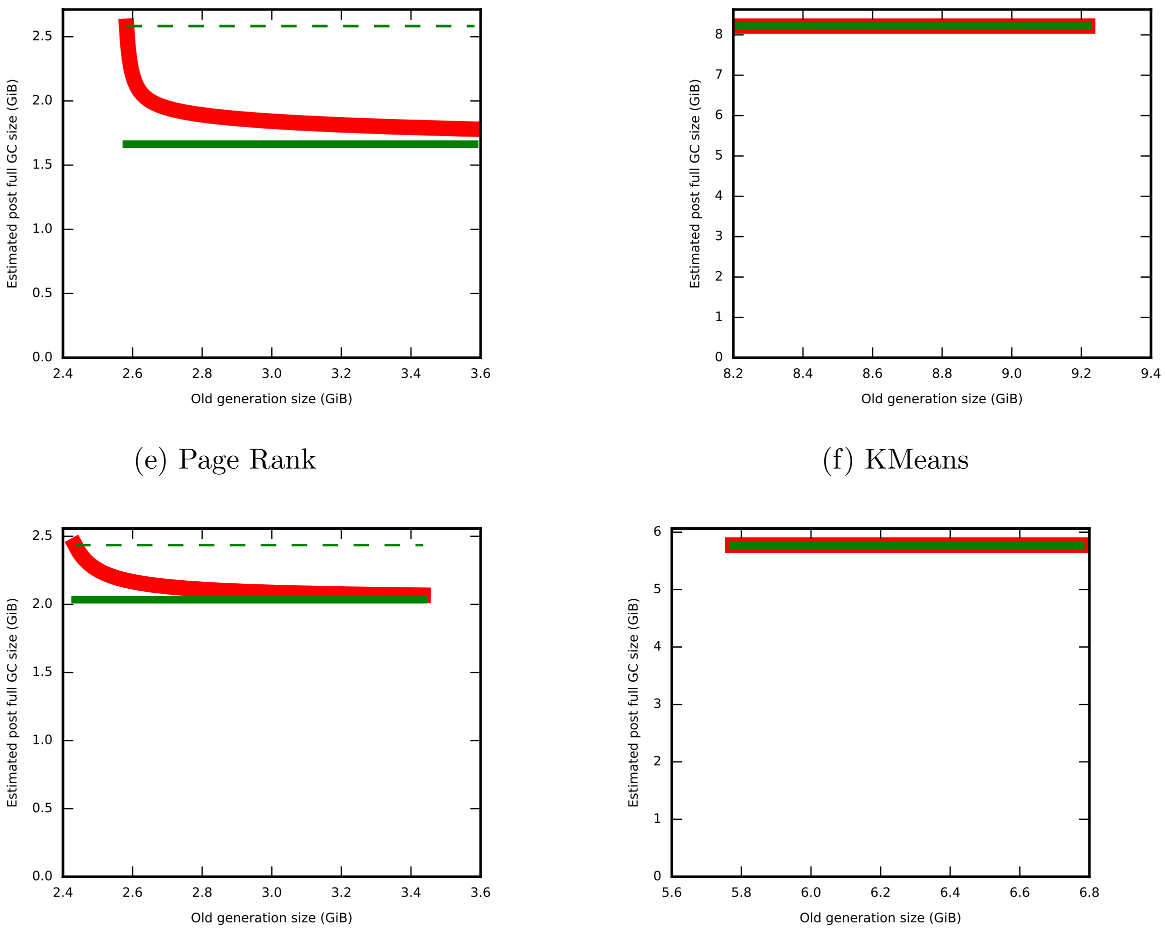
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.7: Estimated space left over for a particular tenured space size based on distribution of sizes observed after full collections (red, non-straight line), and based on average and maximum sizes observed after full collections (green lines).



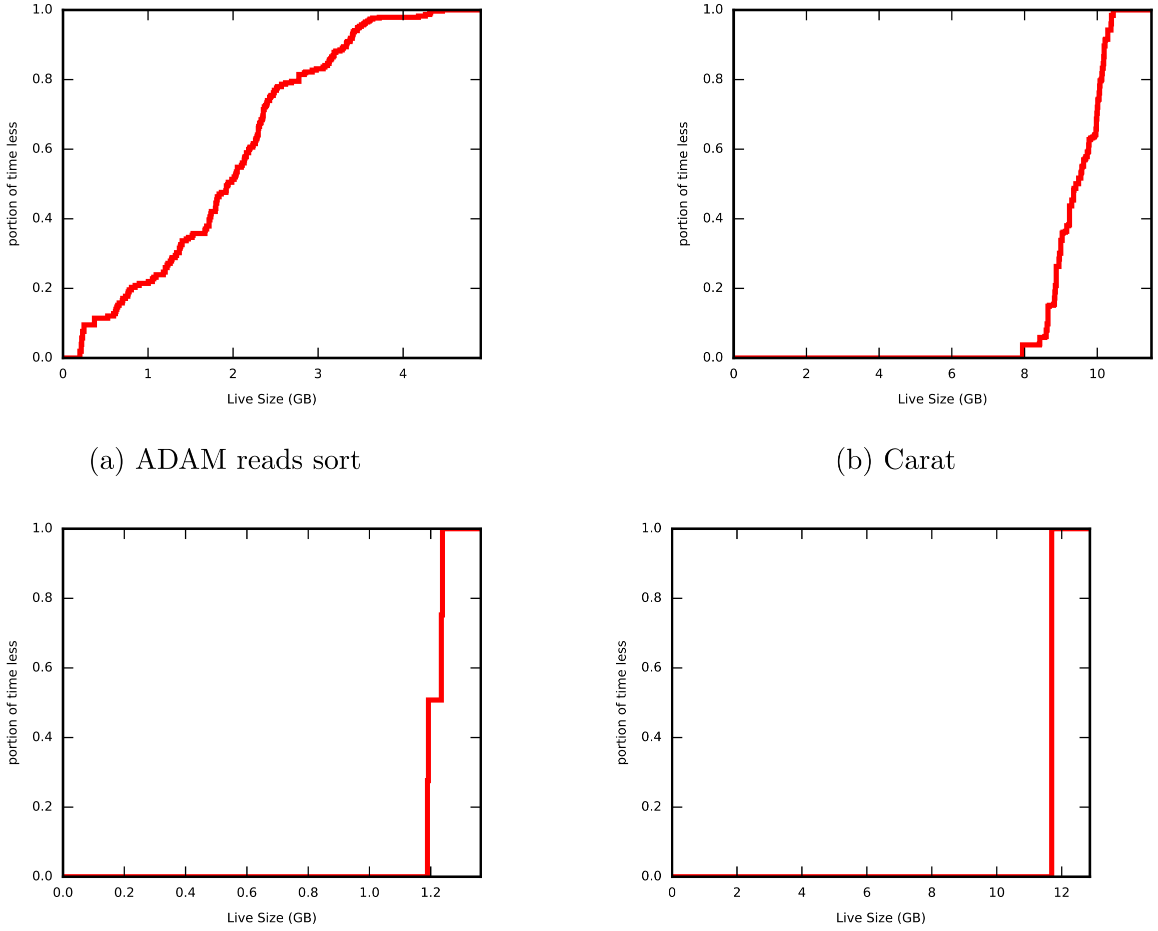
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.7 (con't): Estimated space left over for a particular tenured space size based on distribution of sizes observed after full collections (red, non-straight line), and based on average and maximum sizes observed after full collections (green lines).



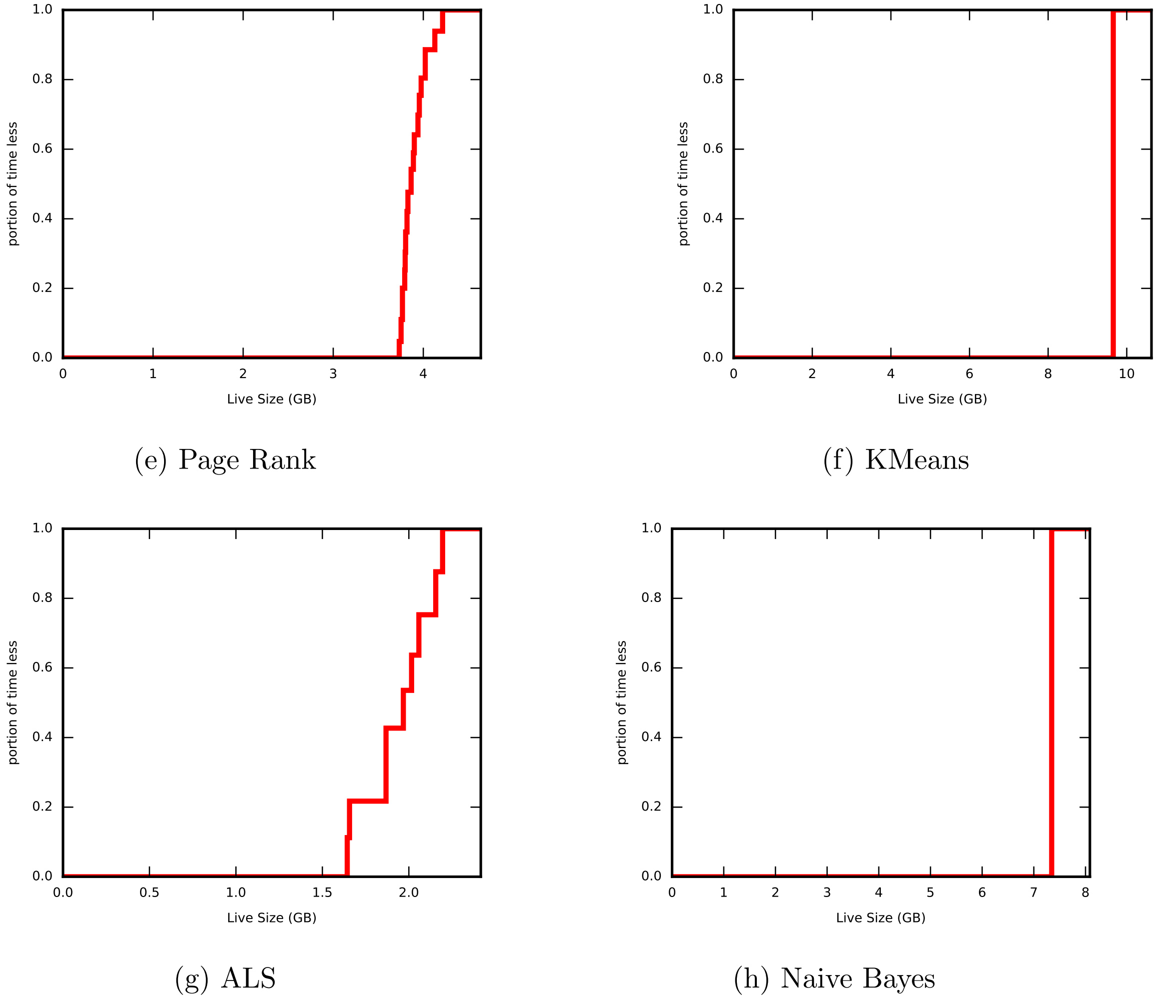
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.8: Weighted CDFs of amount live after full collections captured from instrumented runs. The weight is the portion of allocations experienced when the program is at approximately that live size. Note the variety of behavior — many programs have essentially a fixed amount of memory they require, but some have substantially variation across their lifetime.



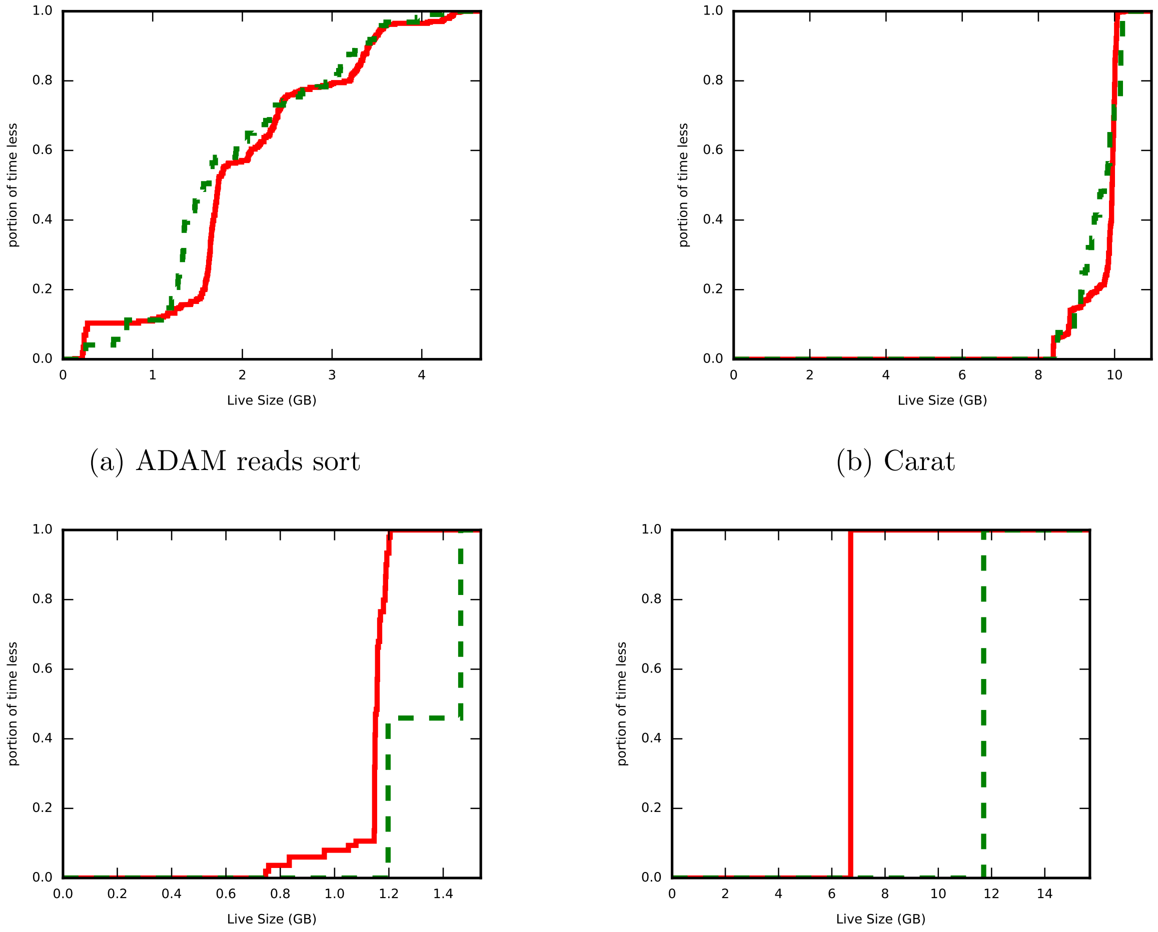
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.8 (con't): Weighted CDFs of amount live after full collections captured from instrumented runs. The weight is the portion of allocations experienced when the program is at approximately that live size. Note the variety of behavior — many programs have essentially a fixed amount of memory they require, but some have substantially variation across their lifetime.



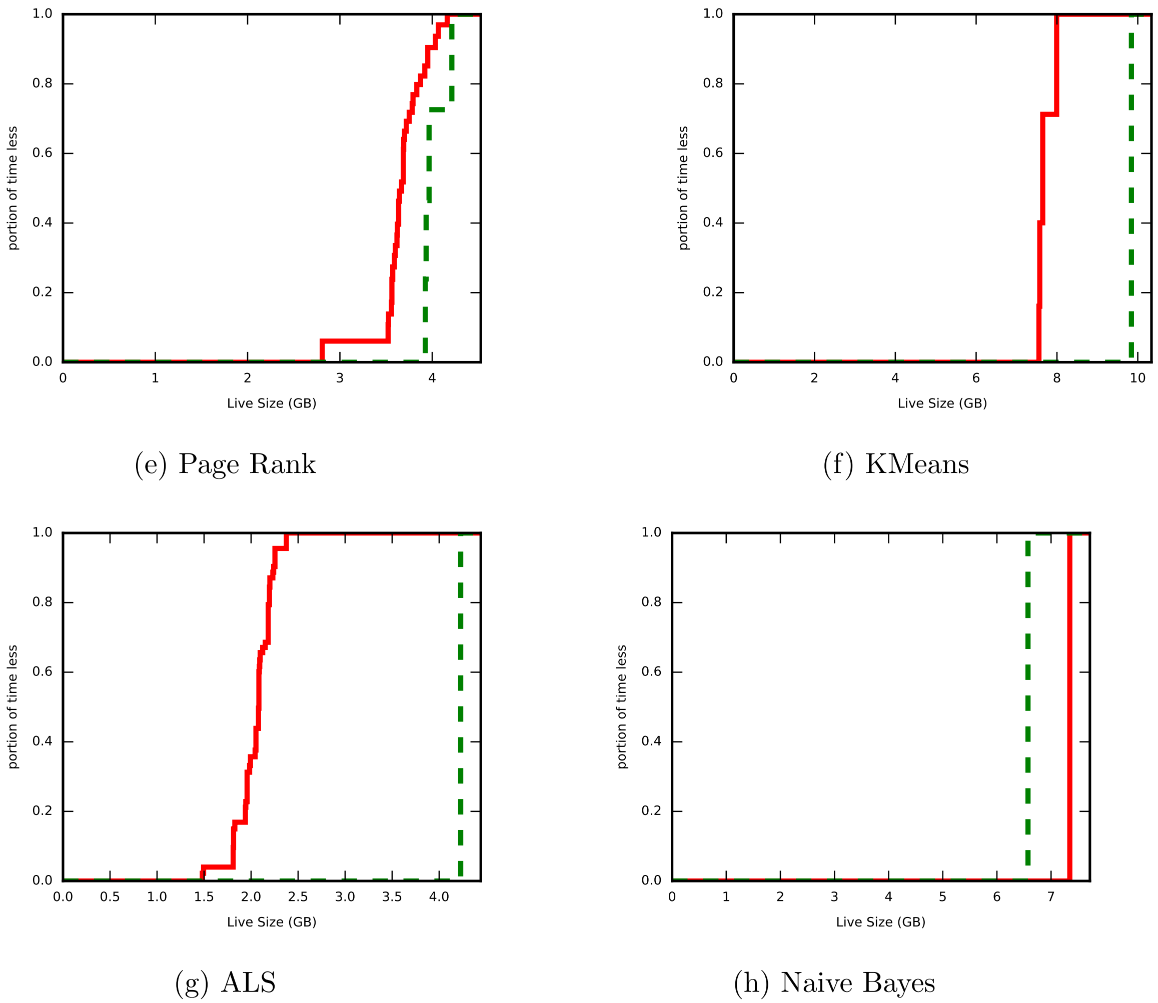
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.9: Difference in recorded estimates of amount of live with varying tenured space sizes (by factor of 2). Lines show estimates from runs with minimum and maximum tenured space size. When there are no full collection to base an estimate on, the average occupancy is used, which generally results in an overestimate.



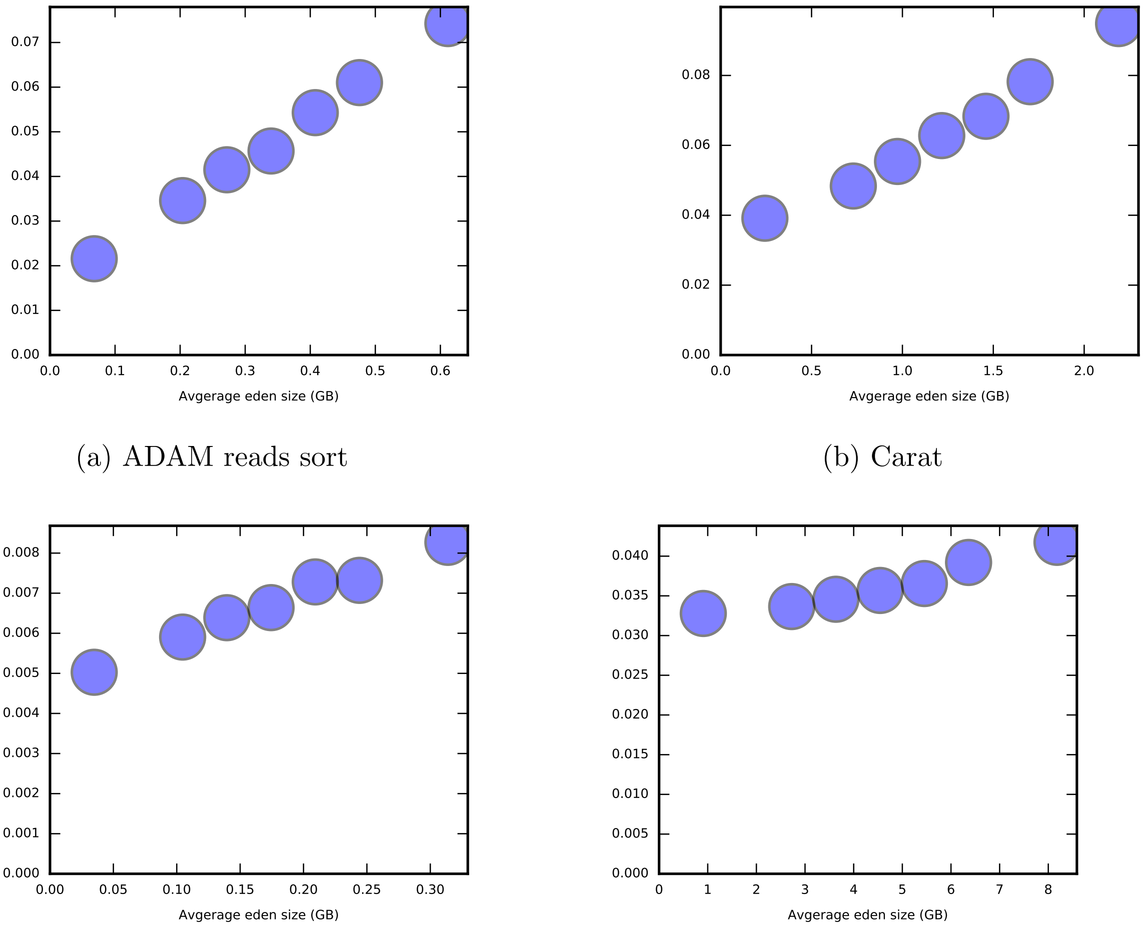
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.9 (con't): Difference in recorded estimates of amount of live with varying tenured space sizes (by factor of 2). Lines show estimates from runs with minimum and maximum tenured space size. When there are no full collection to base an estimate on, the average occupancy is used, which generally results in an overestimate.



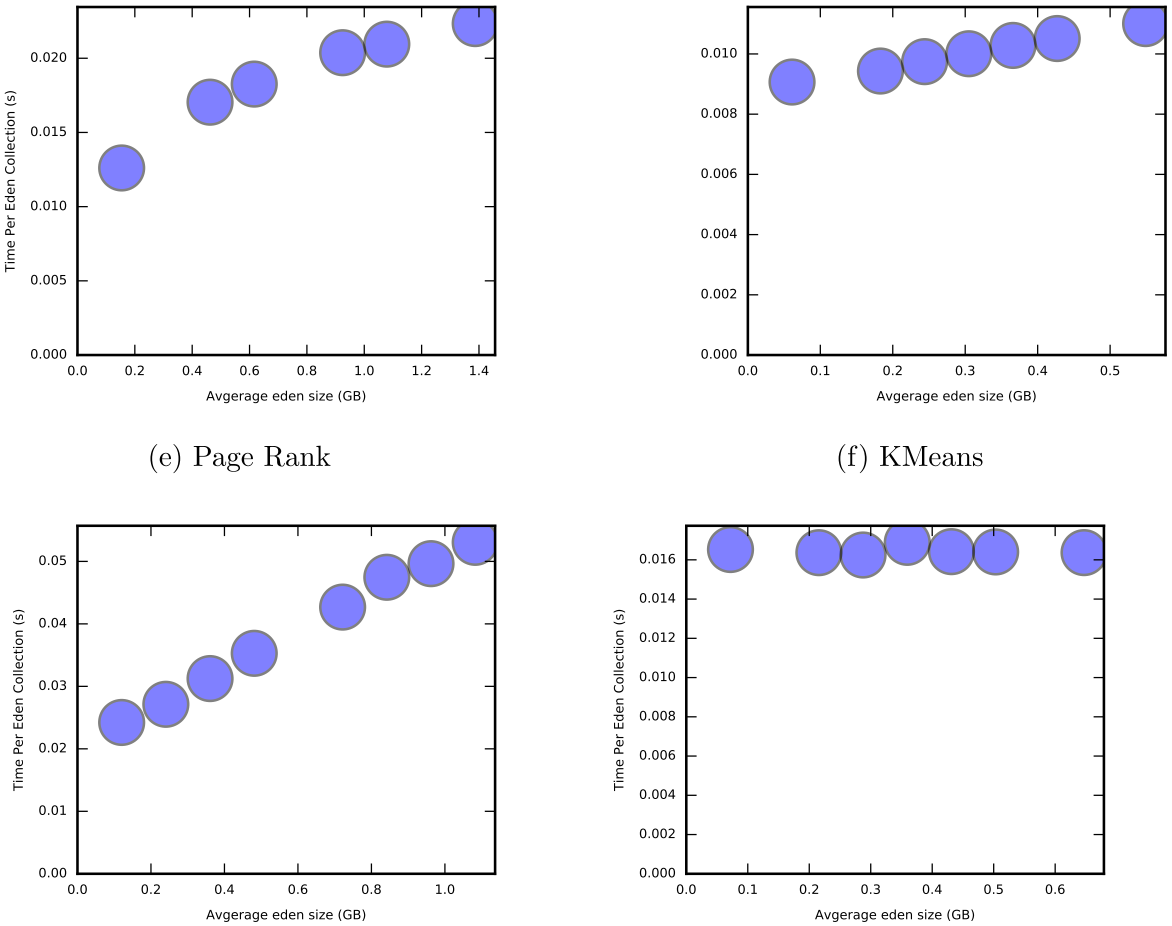
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.10: Average time per eden collection versus average eden size.



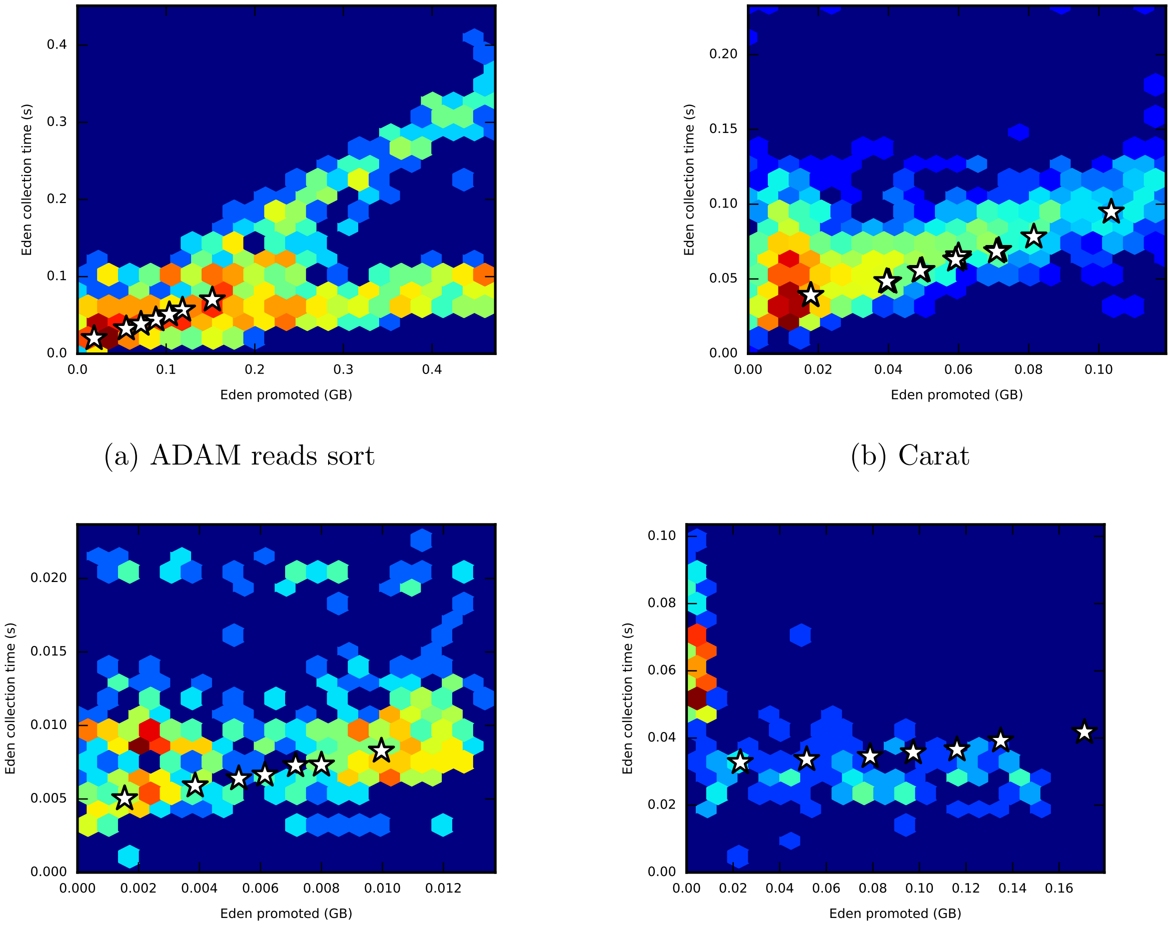
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.10 (con't): Average time per eden collection versus average eden size.



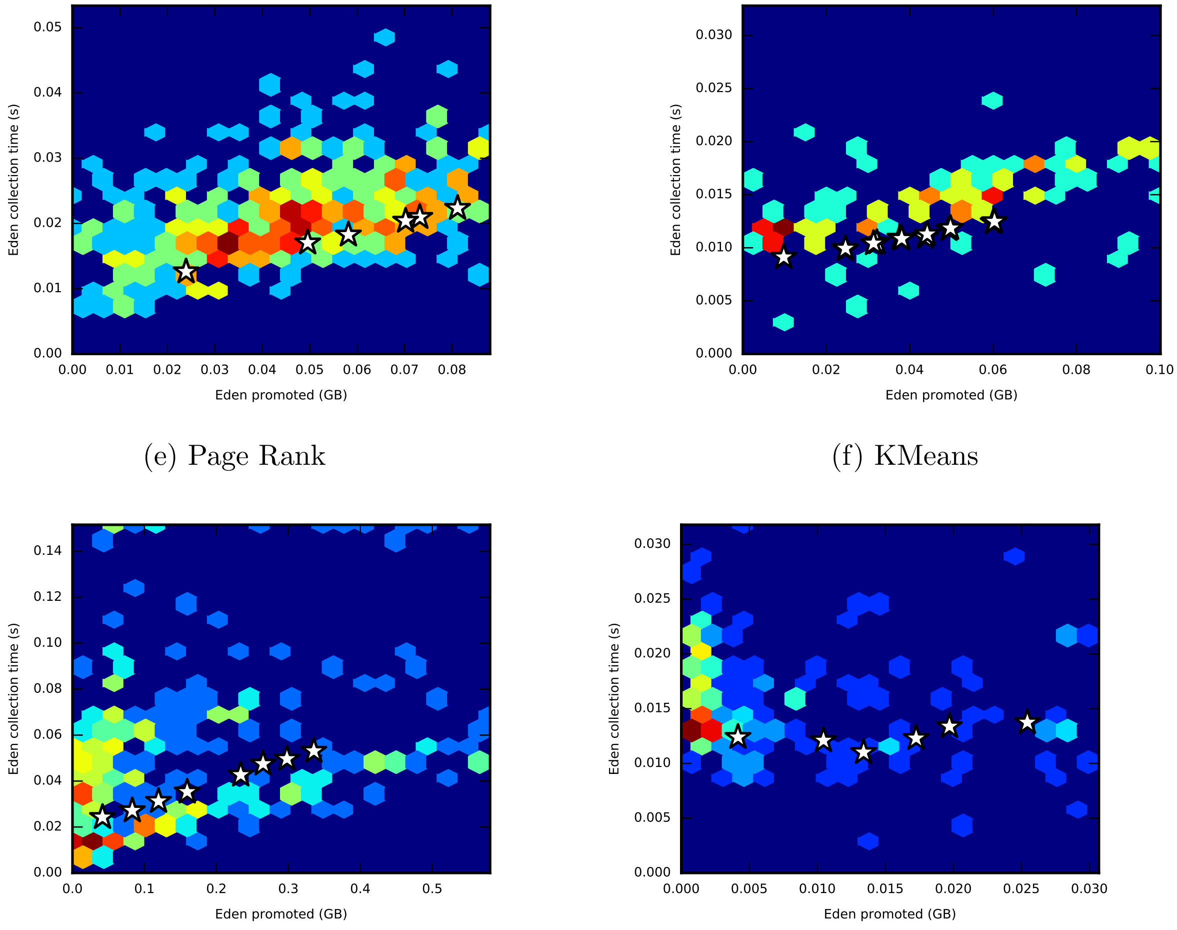
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.11: Time per eden collection versus promoted per eden collection across the eden collections (density, blue is least common) from an instrumented sample run along with actual measurements from varying eden space sizes (stars). The observations use a fixed, but different (usually smaller, based on the suggested garbage collector configuration) tenured space size than the instrumented run, which may account for why they seem slightly offset from the linear regression line. Section 6.6.1.2 describes how I can compensate for this effect.



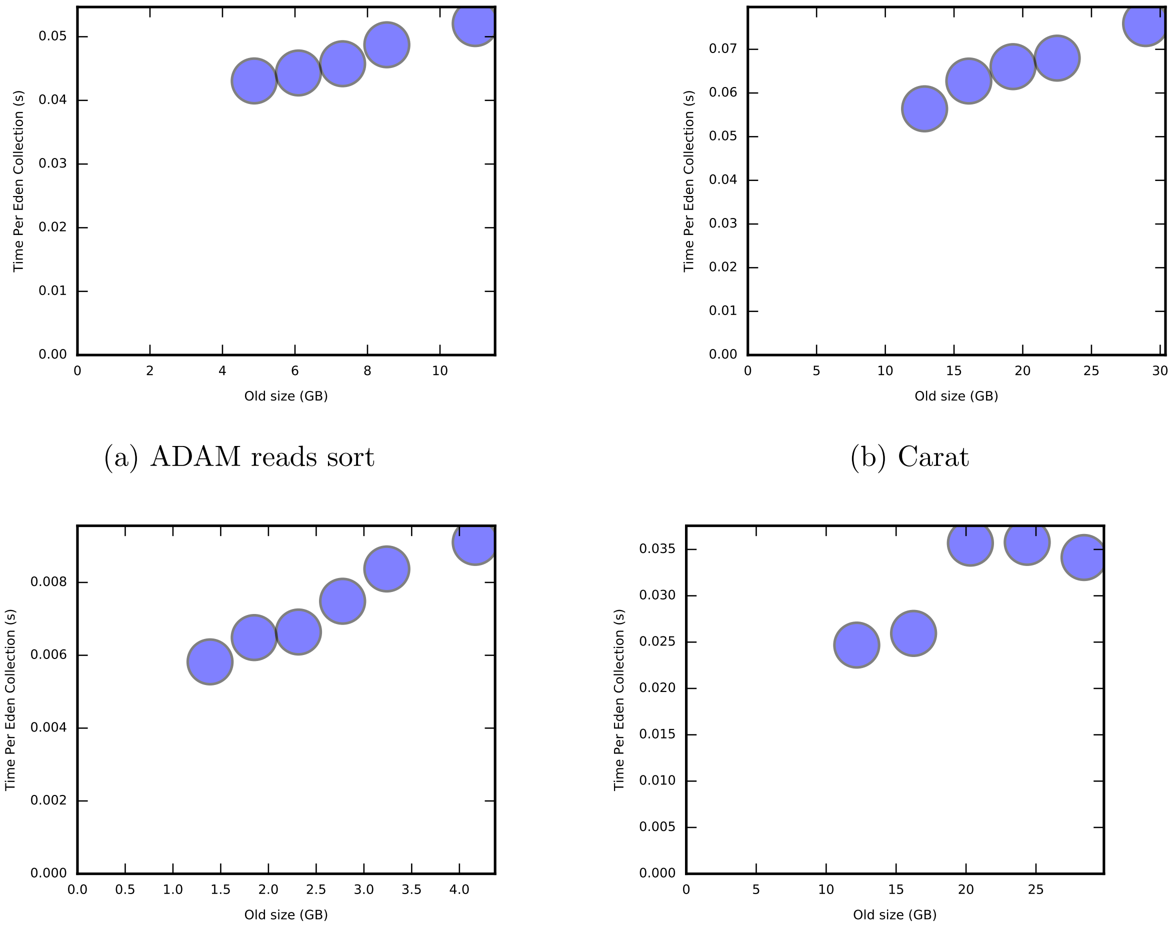
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.11 (con't): Time per eden collection versus promoted per eden collection across the eden collections (density, blue is least common) from an instrumented sample run along with actual measurements from varying eden space sizes (stars). The observations use a fixed, but different (usually smaller, based on the suggested garbage collector configuration) tenured space size than the instrumented run, which may account for why they seem slightly offset from the linear regression line. Section 6.6.1.2 describes how I can compensate for this effect.



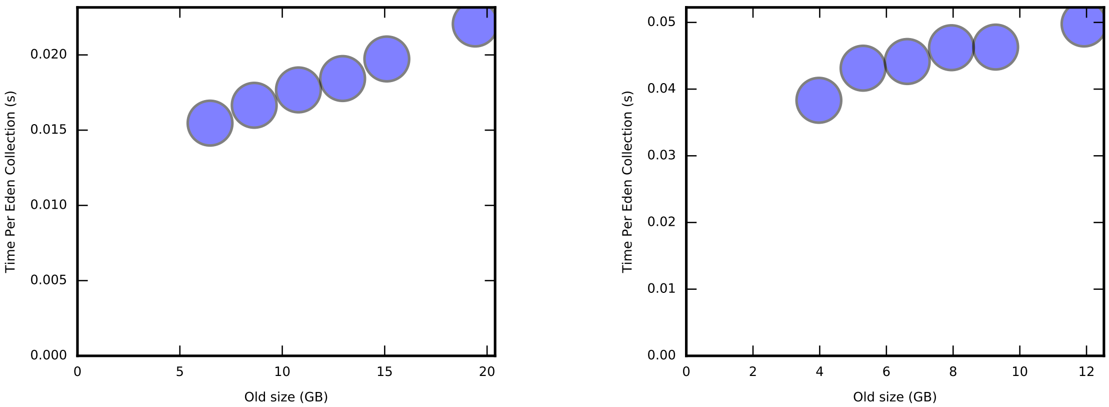
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

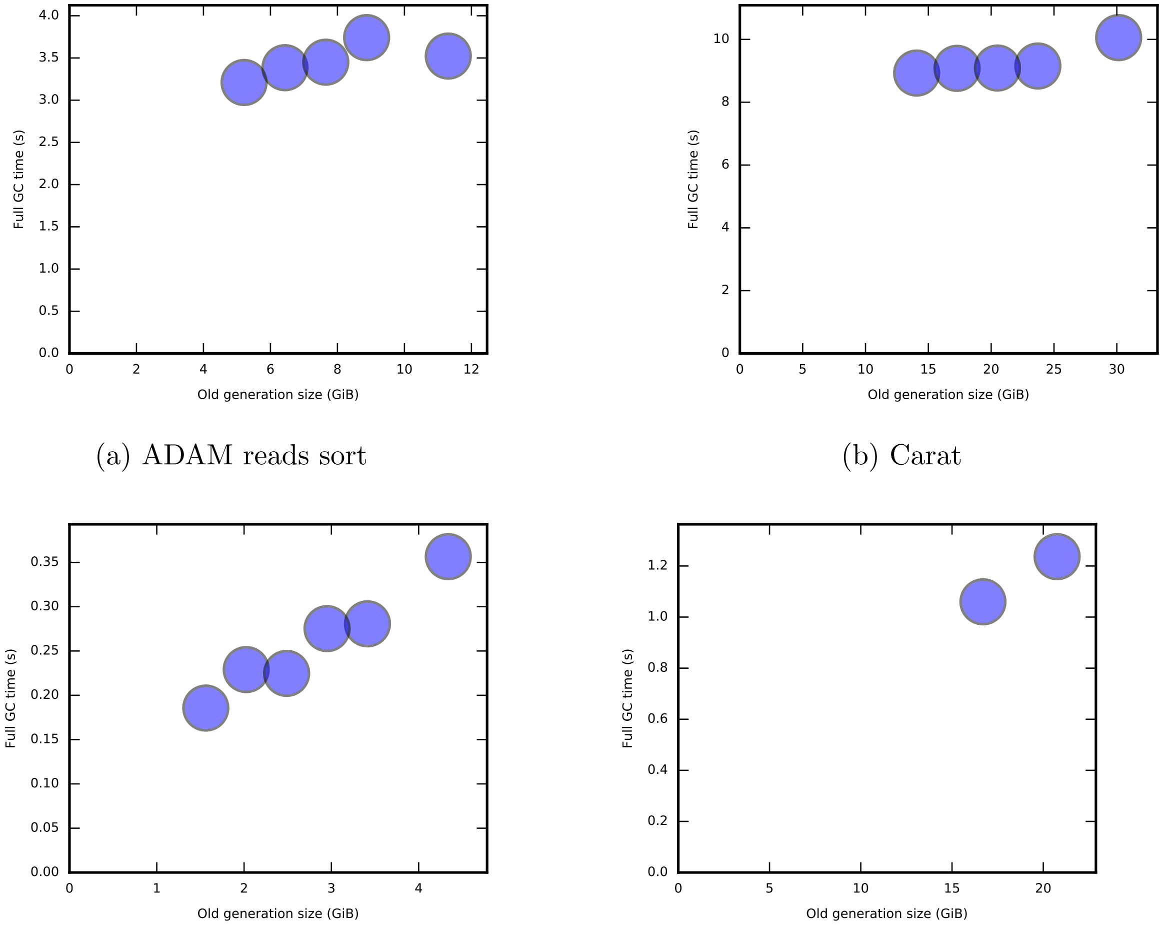
Figure 6.12: Time per eden collection versus tenured space size.



(e) Page Rank

(f) ALS

Figure 6.12 (con't): Time per eden collection versus tenured space size.



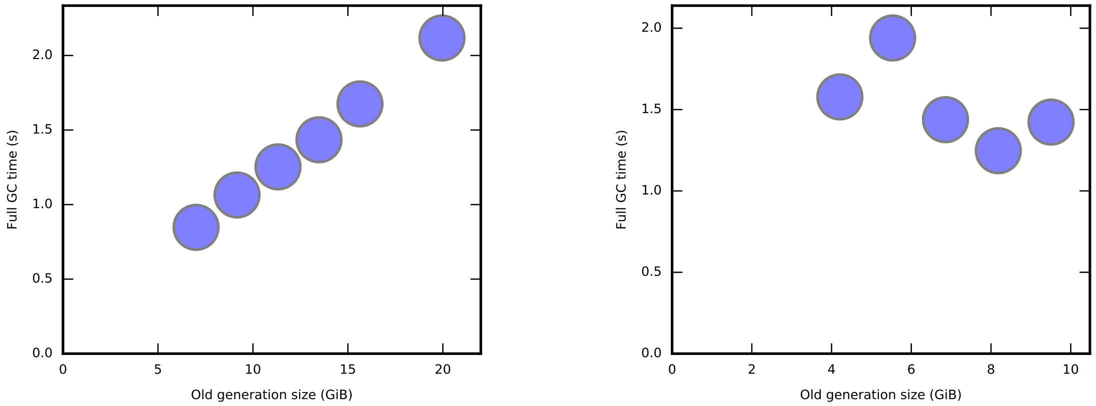
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

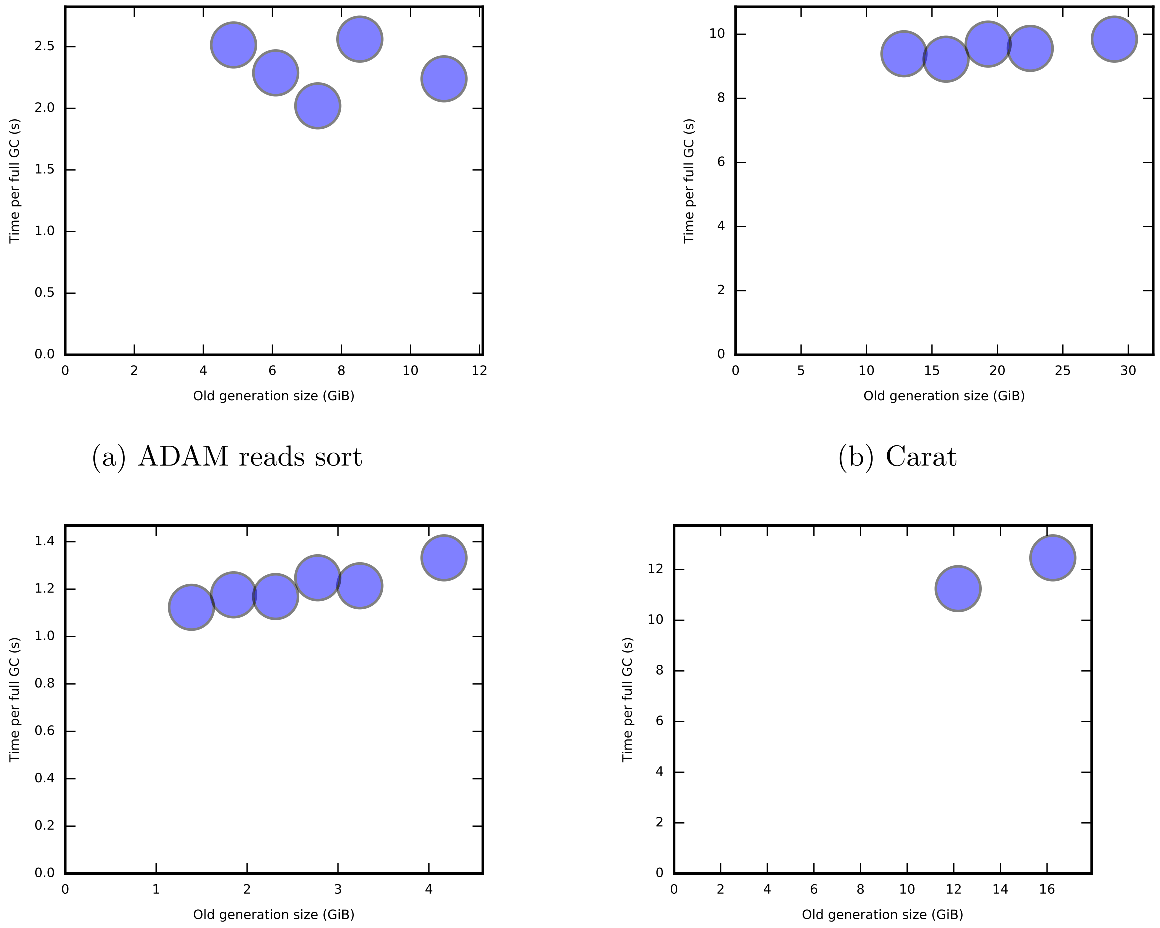
Figure 6.13: Mean time per full collection versus heap capacity. The heap capacity is varied by only changing the tenured space size.



(e) Page Rank

(f) ALS

Figure 6.13 (con't): Mean time per full collection versus heap capacity. The heap capacity is varied by only changing the tenured space size.



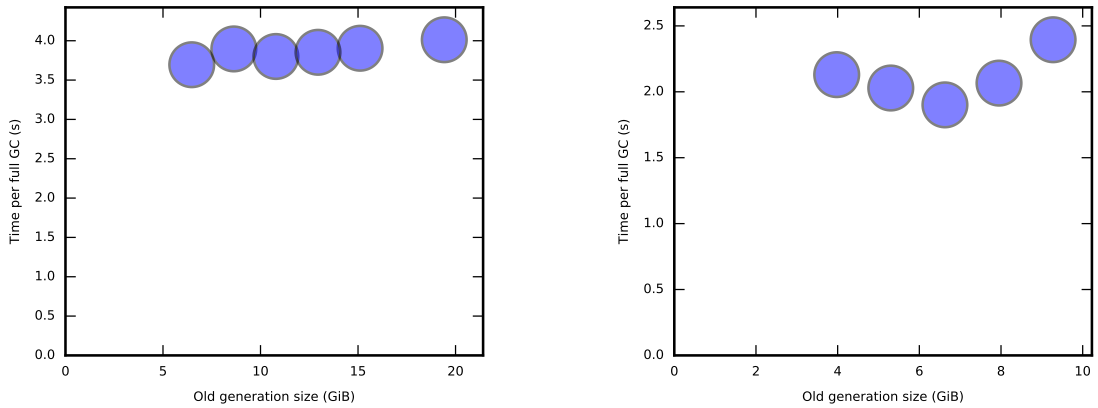
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

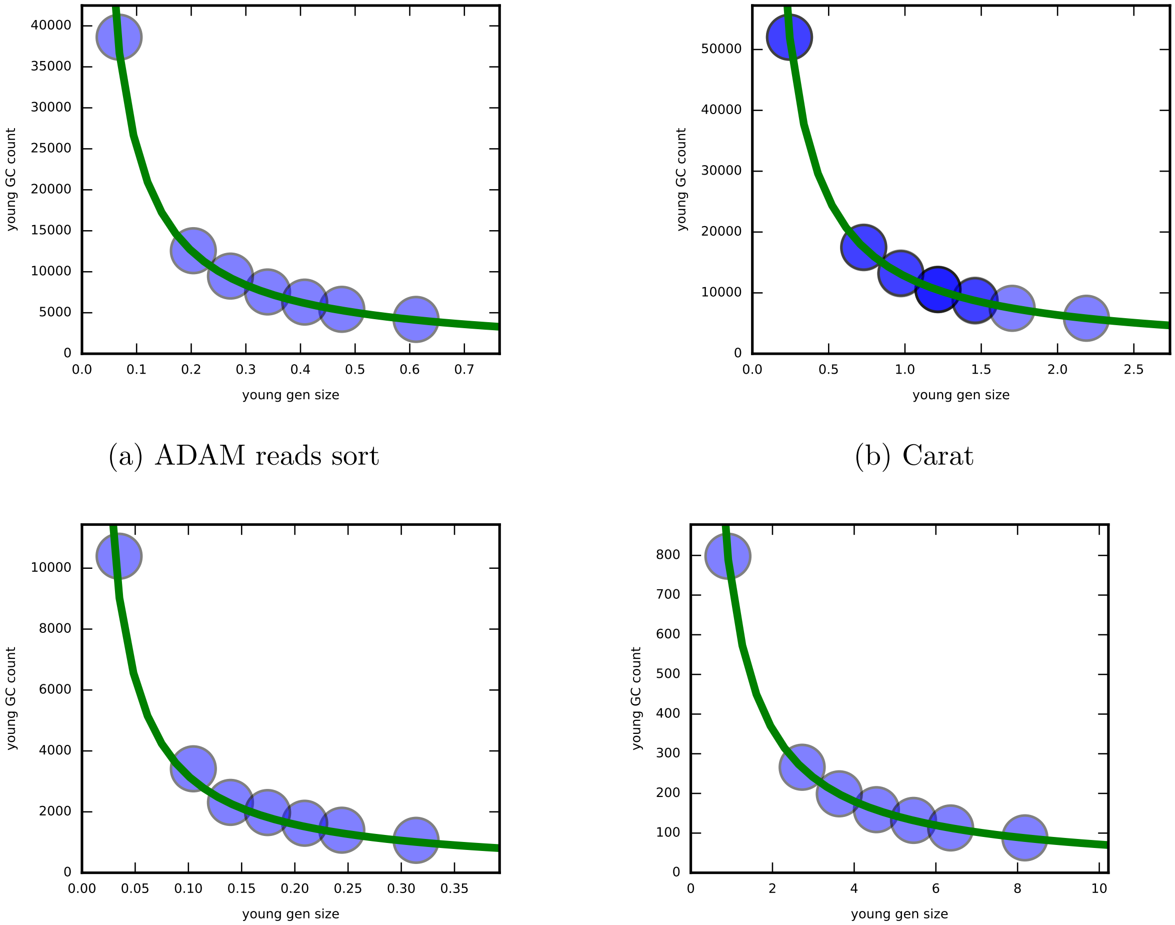
Figure 6.14: Mean space used after full collection versus tenured space capacity.



(e) Page Rank

(f) ALS

Figure 6.14 (con't): Mean space used after full collection versus tenured space capacity.



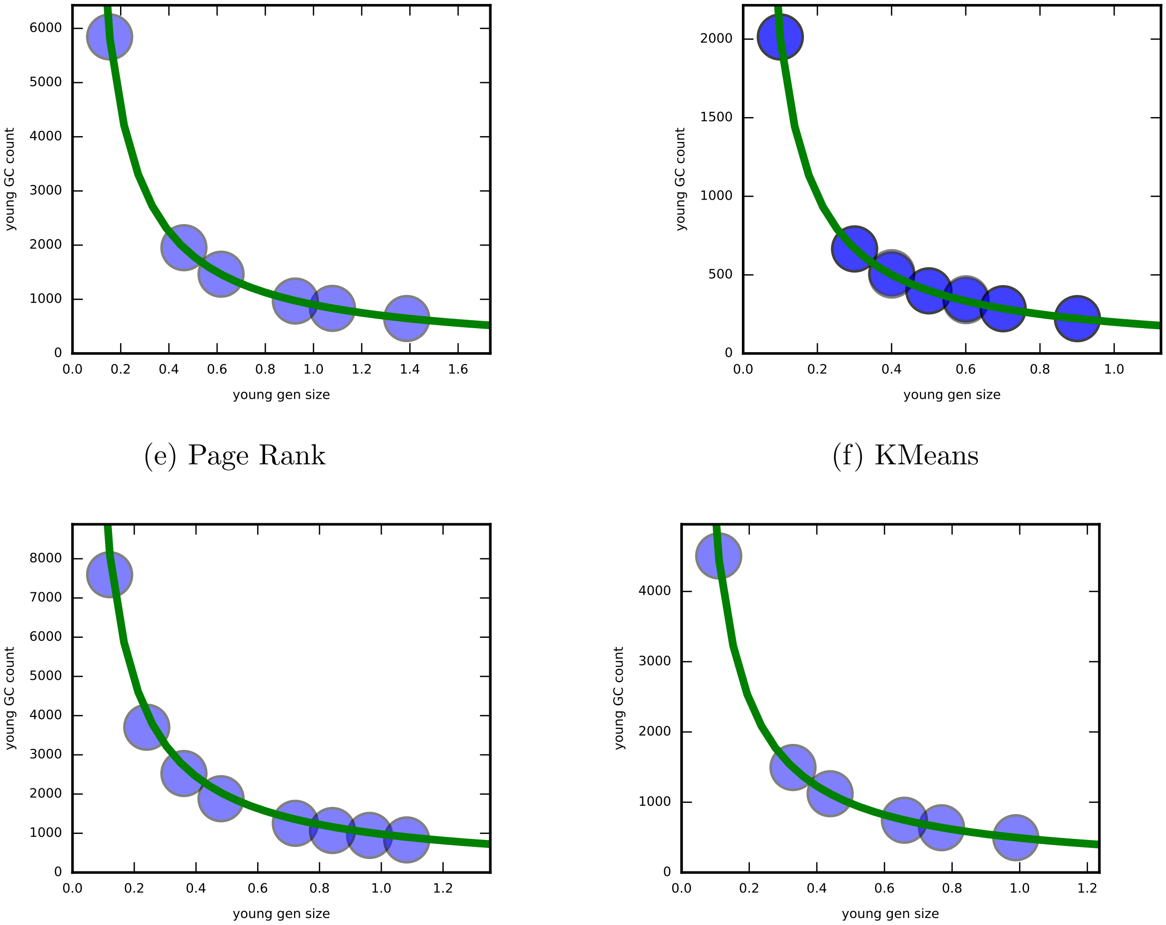
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.15: Predicted versus actual eden collection counts as eden space size is varied.



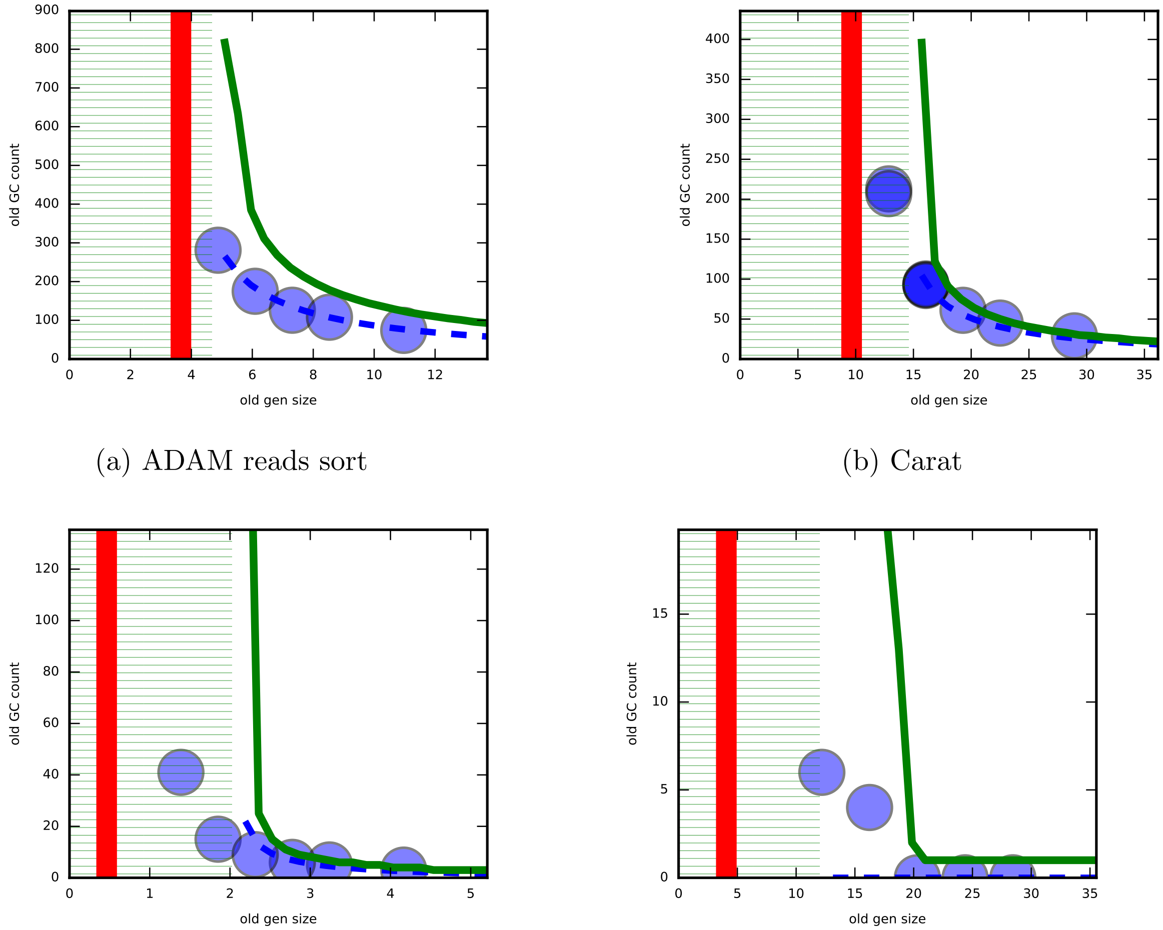
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.15 (con't): Predicted versus actual eden collection counts as eden space size is varied.



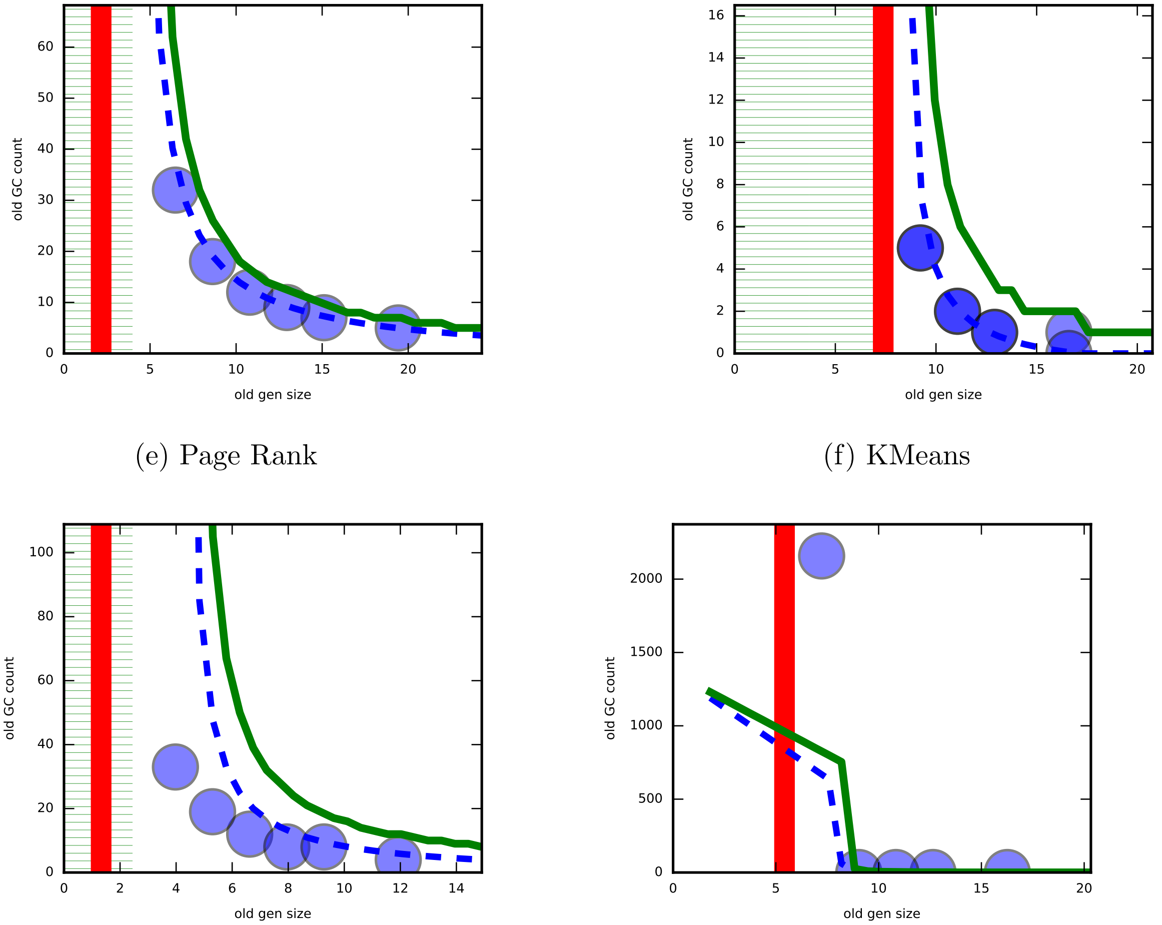
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.16: Predicted versus actual full collection counts as tenure space size is varied.



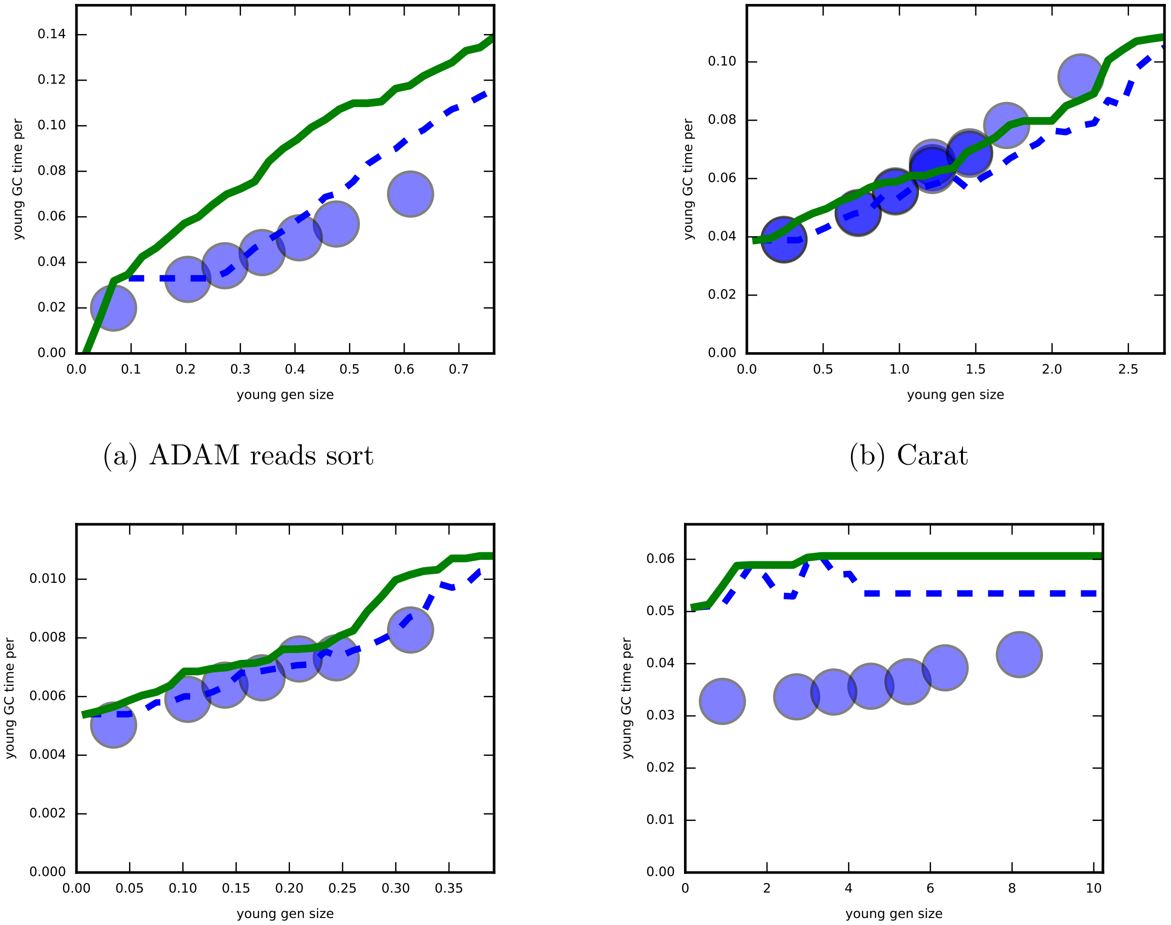
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.16 (con't): Predicted versus actual full collection counts as tenure space size is varied.



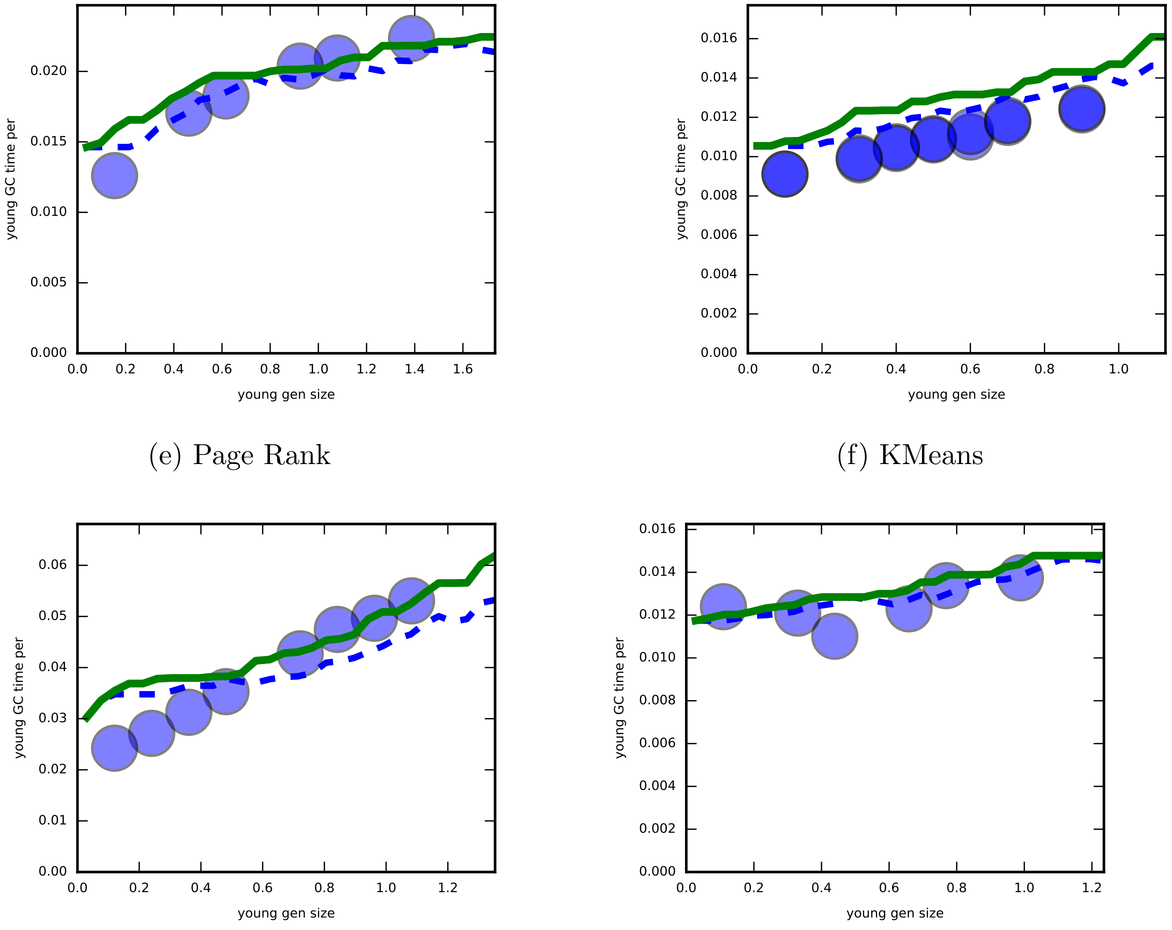
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 6.17: Predictions of time per eden collection from single instrumented run (line) as new generation size is varied versus actual times (circles).



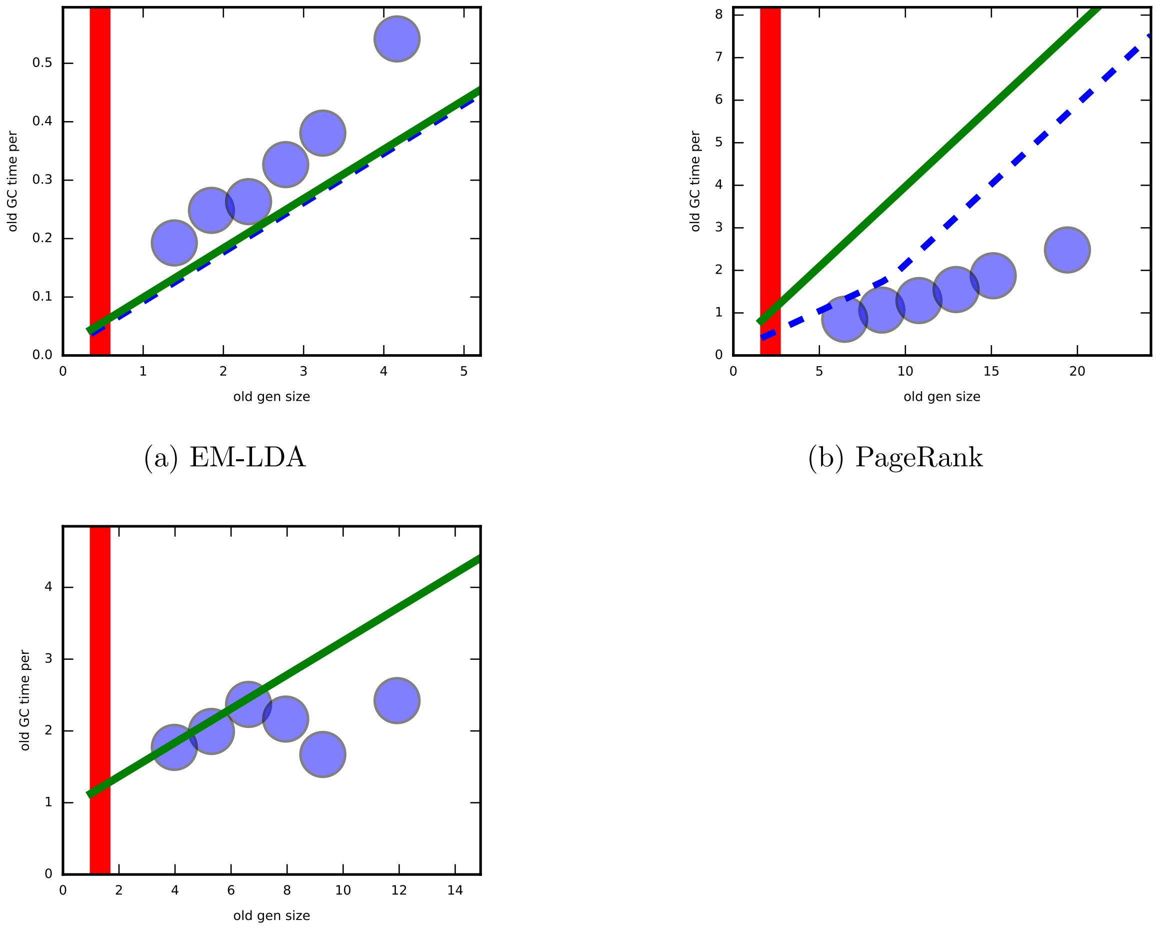
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 6.17 (con't): Predictions of time per eden collection from single instrumented run (line) as new generation size is varied versus actual times (circles).

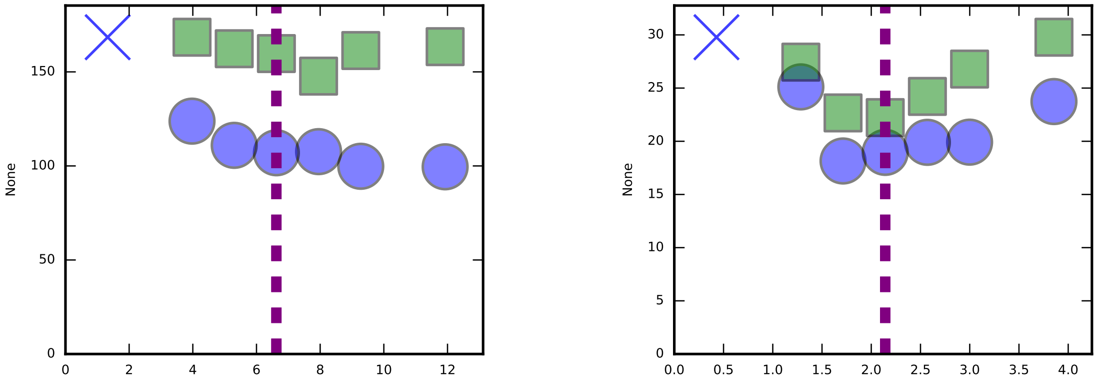


(a) EM-LDA

(b) PageRank

(c) ALS

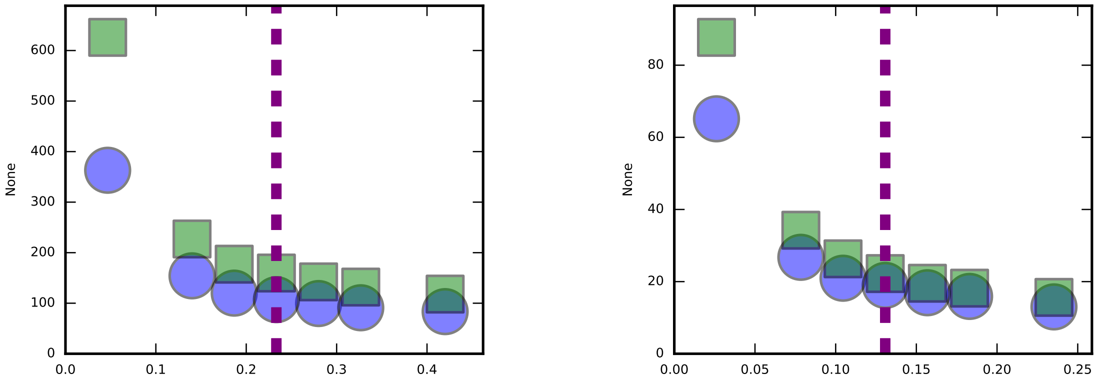
Figure 6.18: Prediction of time per full collection from two instrumented runs (lines) as tenured space size is varied versus actual times (circles).



(a) ALS

(b) EM-LDA

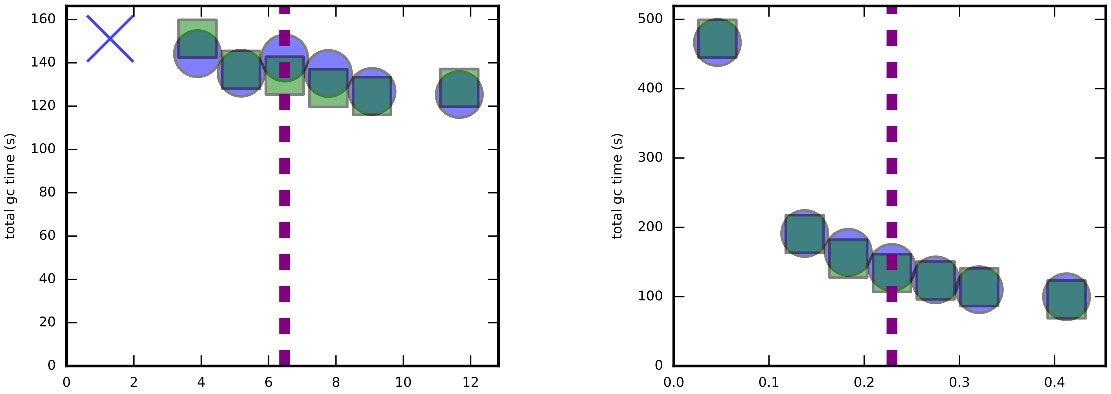
Figure 6.19: Difference in collection time between enabling survivor spaces as tenured space size varies. Note that enabling survivor spaces appears to cause substantially more overhead for each of these programs.



(a) ALS

(b) EM-LDA

Figure 6.20: Difference in collection time between enabling survivor spaces as young generation size (survivor + eden) varies.



(a) Tenured space varied

(b) Young space varied

Figure 6.21: Difference in collection time with fixed survivor space sizes (set to a fixed portion of the eden space size) for the ALS example for varying tenured and young generation size.

Chapter 7

Making Sizing Recommendations

The models described in the previous chapters do not directly yield configurations. In the simplest case, where users can ask for any portion of the memory of some massive machine — minimizing overhead is easy; the user can ask for it all. But to make sensible recommendations, my tool SLAMR must model the user's costs that make it worth asking for less. In addition to concerns about costs, a user's choices involve not only knowing how much to request but in what form to request it. On common shared computing platforms, as described in Section 2.3, users have a choice between requesting a small amount of memory across many machines or a large amount of memory on a few machines.

To turn framework minimums and garbage collection overheads into recommendations, SLAMR relies on a rough model of user utility. It does not spend the collect data sufficient to map out some time/space tradeoff precisely and especially not the benefits or lack of benefits of parallelism for each program. The goal of using a notion of utility, then, will be to find an efficient point and rule out ridiculous configurations. Given the lack of precision in my runtime estimates, I target this utility metric towards selecting a 'minimal viable resource' configuration; among configurations with approximately equal cost, absent other guidance, I bias my model towards the smallest.

7.1 User Utility Models

The biggest challenge in producing a single recommendation is mixing the 'keep everything in memory' heuristic with the time overhead model for the garbage collector. To do this, one needs to decide what is an acceptable amount of overhead. To make this choice, one must put that overhead in context. The way I will do this is by trying to estimate a virtual 'cost', based on the amount of machine resources (memory, compute) a user allocates and the estimated runtime of the program. Estimating the runtime will require taking into account both the time overhead of the garbage collector and the normal useful runtime of the program.

7.1.1 Quick Estimates of Non-GC time

To make sensible recommendations, some assumptions about program scaling will be required. One cannot simply ignore changes in parallelism because of how users change memory configurations. Users cannot abstractly choose any amount of memory to use. For distributed computing platforms like Spark, they choose a number of workers and the amount of memory each of those workers has. In any environment where they can run a framework like Spark, their choice of how much memory is coupled with their choice of how much compute resources.

Nor will the heuristic of specifying a 'minimum' save one from modeling performance scaling. Otherwise, absurd suggestions are likely: leaving half a machine's cores unused to save a tiny amount of memory or barely fitting in the memory of one machine at the cost of huge garbage collection overheads. I do not attempt to construct precise predictions of runtime, as modeling parallelism accurately likely requires more detailed and less program-agnostic instrumentation. Instead, my heuristic assumption is that there are diminishing returns from parallelism after the working set size is allocated. My goal will be to produce the point where these diminishing returns start. From this, users can choose to allocate larger amounts of resources to improve compute time if they choose.

A naive approximation for estimating runtime is assuming linear scaling. But this quickly produces absurd results. Garbage collection overheads are often superlinear: doubling the amount of free space available to a program can decrease garbage collection overhead by about a factor of two (by halving the number of full collections, at least). Doubling the memory available to a program will more than double the amount of free space available (since the initial free space will be more than half the memory). Thus, the logical conclusion would be that best efficiency is achieved at an essentially infinite degree of parallelism. The superlinear scaling of garbage collection means this absurdity could even survive minor relaxations of linear scaling, like assuming a small fixed overhead.

To avoid this, when suggesting degrees of parallelism, I make conservative assumptions about parallel scaling. I pessimistically model a large portion (around 50%) of the computation as serial. The primary goal of this choice is to avoid the absurd configurations. One absurdity to avoid is the 'infinite' size configurations that assuming super linear or near-linear scaling is likely to result in; this heuristic clearly avoids suggesting such configurations. Another are configurations that thrash and essentially never complete due insufficient memory; these are avoided with relatively little sensitivity to how good the scaling estimate is. So long as the scaling is not so extreme as to cause the 'cliff' of very high estimated overheads to appear good, configurations with absurdly few resources will still be avoided.

7.1.2 Exact Memory Charging

After estimating runtimes, one needs to concretely compare the 'costs' of a certain amount of memory and runtime with another amount. The simplest way I will do this will be an idealized form of utility computing. If memory is the most constrained resource in a shared cluster, then to a first approximation, users will pay only for memory for each unit of time.

In this approximation, the cost of the compute resources is negligible and users can pay for exact amounts of time and memory. Reality, of course, has other constraints. This approximation is particularly useful for evaluating recommendations. Although the real world is more discrete, this very precise charging lets any two performance results be meaningfully compared. Rounding to the nearest hour or nearest whole machine size would often prohibit comparisons of different recommendation.

7.1.3 Library of Node Choices

In reality, users cannot just purchase as much memory resources as they want in isolation. There are multiple models for renting computing resources. But, generally, the economies involve imply minimum and maximum ratio of compute, memory, and storage resources that must be obtained together. It is important to navigate these real-world complications.

The dominant charging model for utility computing providers like Amazon EC2 reflects these realities. They have users choose from a library of virtual machine sizes. In Amazon's case, these sizes are chosen so that it is plausible that Amazon statically partitions physical machines into fixed virtual machines. Ultimately, the size of these physical machines determines the 'boxes' into which users must fit their resource requests.

Even more flexible providers like Google Compute Engine (GCE) impose restrictions on the 'shape' of acceptable virtual machines that go beyond what's supported by the underlying physical hardware. For example, as of this writing, GCE requires that virtual machine allocations between 0.9 GB and 6.5 GB per virtual core, even though they support up to 32 virtual cores and up to 208 GB of memory per node is supported. Consequently, in any of these environments, the decision about how much memory to allocate is necessarily also a decision about how many worker machines to use and how many cores to allocate. The cost users pay is based on a combination of this factors.

To evaluate configurations for this environment, then, SLAMR estimates the cost based on the number and type of worker nodes allocated and how long they are allocated. To turn the time and worker usage estimates into a cost, I use a continuous estimate of the time even though charging is not continuous. Since SLAMR uses a rough approximation of runtime, it is not reasonable to imagine that I account for time rounding effects, and I assume users will have work of the type sampled to do for a long enough time to amortize these costs.

7.2 Refining Configurations

7.2.1 Instrumentation Runs

A user starts searching for configurations from one successful execution. From this execution's instrumentation, my tool obtains the framework-level configuration. Because it used a different framework configuration, the garbage collector demands of this instrumented run may not be typical of the final configurations. So rather than rely on my garbage collector model, I first suggest a configuration with what is generally more than enough extra space. In my tests, I added 50% to the space that I estimated to be required by Spark itself (according the framework model only) for garbage collector usage. A less conservative approach would be to use the initial misconfigured run to infer initial garbage collector configurations so long as that run allocated 'too much' memory to the framework and did not use many more workers.

While it is safe to make inferences for framework parameters from any successful run, the same is not true for garbage collector configuration. When Spark stores changes the amount stored in its cache, this may dramatically change garbage collector behavior. In addition to the occupancy of memory changing, additional recomputation, reading and writing data to persistent storage, etc., may change the quantity of allocations and the distributions of their lifetimes — precisely what the garbage collector instrumentation is supposed to measure.

In some cases, however, the only significant change will be the occupancy of memory. For example, when changing the number of workers while keeping framework data in memory, there should be little change to the mix of garbage collector demands. To roughly estimate the effect of these changes, I adjust the measured amount of memory free after full collections according to the portion by which Spark's storage size estimates changes in this case. To keep this from causing major misconfiguration, I still constrain estimates by Spark's total space estimates and do not make this adjustment when Spark's storage space was configured to be much larger than required, meaning that most of it would never be occupied. At the cost of not second-guessing sometimes inaccurate framework size estimates, this ensures that programs whose memory usage is managed by Spark complete with manageable overheads.

The measurement overheads to obtain SLAMR's recommendations are low enough that configuration recommendations can be a continuous process. In my experiments, I arise at a final, stable configuration after the second run where the Spark parameters are settled. For Spark estimates, the final configuration arises after one successful run. For garbage collector estimates, at least another execution is usually necessary, to avoid observing the effects of additional data kept in memory or recomputed. In cases where 'second-order' effects like tenured space sizes or scaling assumptions matter, more iterations may be required for garbage collector configurations to converge on a final value. These additional observations provide information about the effects of changing worker counts and tenured space sizes which cannot be inferred from any single run.

7.2.2 Applying the Utility Model

To choose the number of workers and type of workers, I apply one of the utility models described in Section 7.1. When using the 'library of node types' approach, SLAMR tries each node type — each possible amount of memory and compute resources per worker — and chooses the one with the best estimated cost. Generally, the final node type will be the one with highest memory density, but programs with smaller memory requirements or user constraints on the minimum number of cores to select could cause other node types to be selected. Once a type of node is fixed, I find the best number of nodes by first finding the minimum number of nodes that fit the framework configuration. Then, I find the cost of the best garbage collector configuration on that number of nodes, and larger number of nodes until the cost starts increasing.

In this search, I also include the 'virtual' node types created by not using all available cores. Usually, this will be inadvisable — it will be better to use all the compute resources one is paying for, since it's unlikely that this would require substantially more memory resources. In extreme situations, however, this might be necessary to fit the computation at all. For example, if partition sizes are large, computing on less partitions at a time may be the difference between fitting the partitions in memory and not even on the machine type with the highest memory:CPU ratio.

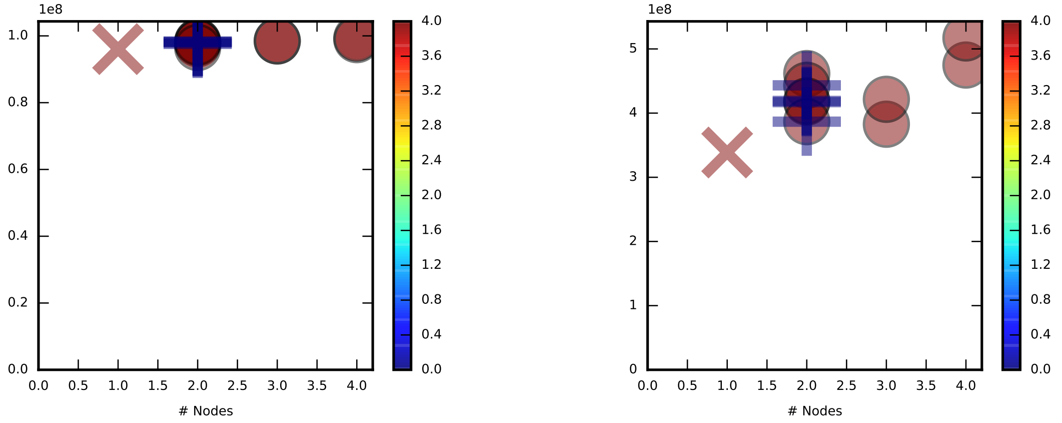
To produce a garbage collection configuration, I choose the eden and tenure space sizes in addition to the number and type of worker nodes. My tool finds the optimal eden and tenure space sizes by performing a grid search over the possible configurations, and then performing a heuristic local search using the downhill simplex method starting from the best feasible point found by the coarse grid search. For the local search, I assume that the predicted overhead is continuous round the minimum cost region. When a library of node types in in use, there is effectively only one parameter (either the eden size or tenured space size) to configure, since optimal configurations use the entire space available on each node (after accounting for page cache and OS requirements).

7.3 Evaluation

To evaluate recommendations, I primarily examine recommendations under the 'exact' model — where a user's cost is based on the exact amount of memory used and the exact time for which it used. I do this instead of directly evaluating the more realistic scenario of node choices, because dangerous mispredictions will only show up near the margins: predictions will be rounded up to the next whole number of nodes, so any mispredictions would often survive this. In addition to evaluating the overall accuracy of combined predictions, I will examine the sensitivity of recommendations. I am concerned about two types of sensitivity: sensitivity to the training run, especially whether my inferences about the number of nodes to configure are reasonable, and sensitivity to cost models, especially my assumption that because I am producing configurations close to a configurations that experience conservatively estimated performance cliff, my exact runtime estimates do not dramatically change recommendations.

7.3.1 Garbage Collector Predictions Across Node Counts

A major concern for observations of garbage collector activity is that, as the number of worker nodes changes, one might need an example run from the exact same number of



(a) KMeans

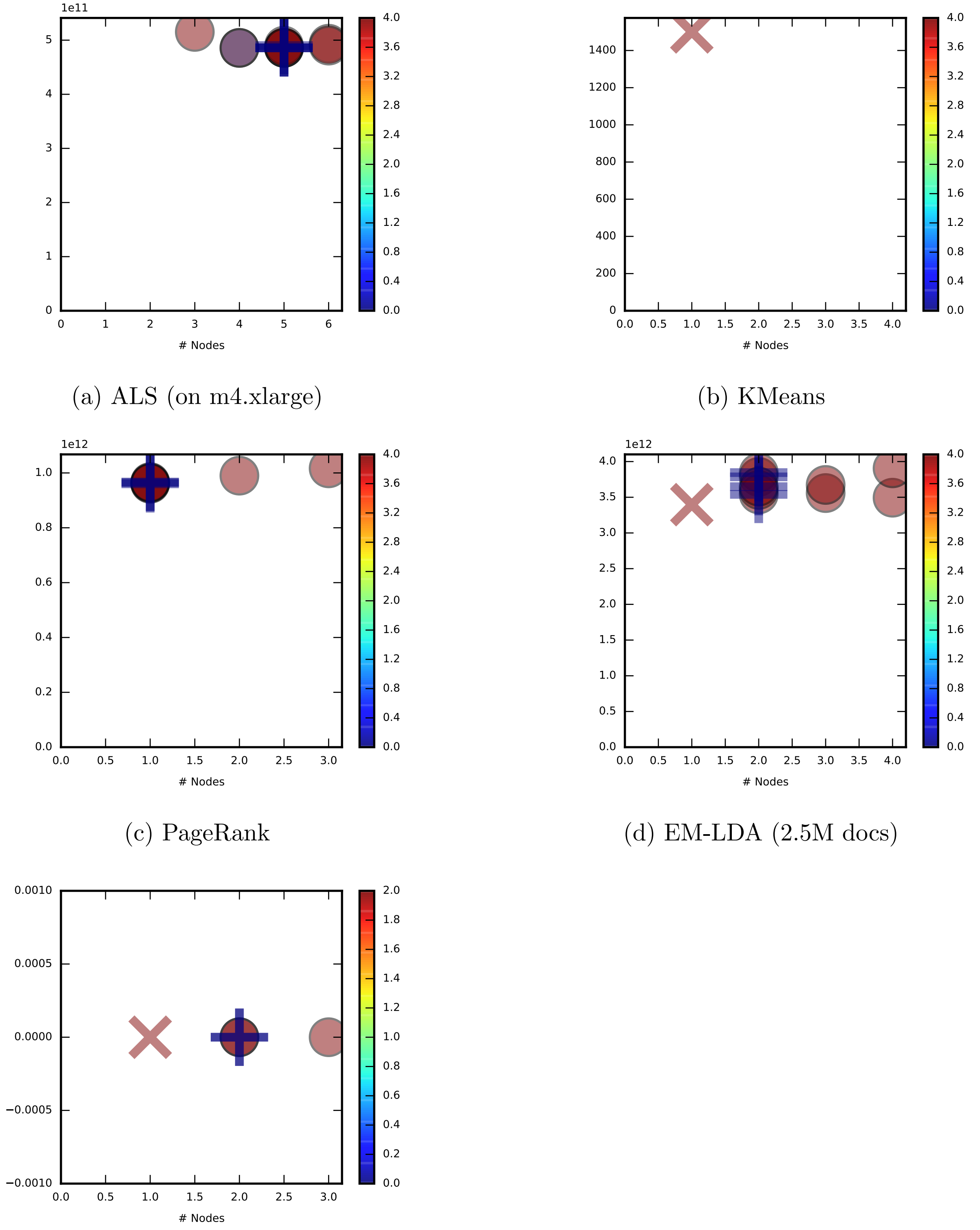
(b) ALS

Figure 7.1: Measured promoted bytes per eden collection for fixed eden size and varying worker counts. X's represent a failure to complete at a specified eden size. Outside of these failures, these results show that the behavior is consistent even when the work of the program is divided more ways.

nodes to make predictions. For general programs, this is infeasible given the measurements described in Chapter 6 because one cannot predict how much data will be active on each node. Fortunately, for Spark programs, one can use the higher-level framework information to reason about data which will must be replicated on each node and data which will be divided between nodes.

For several of the measurements drive my model of garbage collector overhead, no adjustment is necessary. The amounts promoted from a given eden size is stable as the number of worker nodes is varied, as shown in Figure 7.1. The total amount allocated is roughly constant as the number of nodes varies, as shown in Figure 7.2. Thus, the amount allocated on each node can simply be computed as the total amount allocated divided by the total number of nodes. Both of these reflect that these measurements essentially capture the activity of the computations being done on each worker rather than the data stored on each worker.

One measurement used in the garbage collector model, the distribution of live objects over time, will clearly change more significantly. To approximate this, I rely on data from the Spark framework. Based on the Spark configuration I know what the change in the amount of data Spark thinks is stored is between a sampled configuration and a new configuration. I compute a new distribution which is simply adjusted by this fixed difference. For the most concerning case, inferring the behavior of a configuration with fewer nodes, this should usually overestimate the requirement, since Spark's size estimates are usually overestimates.



(a) ALS (on m4.xlarge)

(b) KMeans

(c) PageRank

(d) EM-LDA (2.5M docs)

(e) ADAM sort

Figure 7.2: Measured total allocated bytes for varying node counts (or indication of failure to complete at specified node count)

7.3.2 Cost Accuracy and Trends

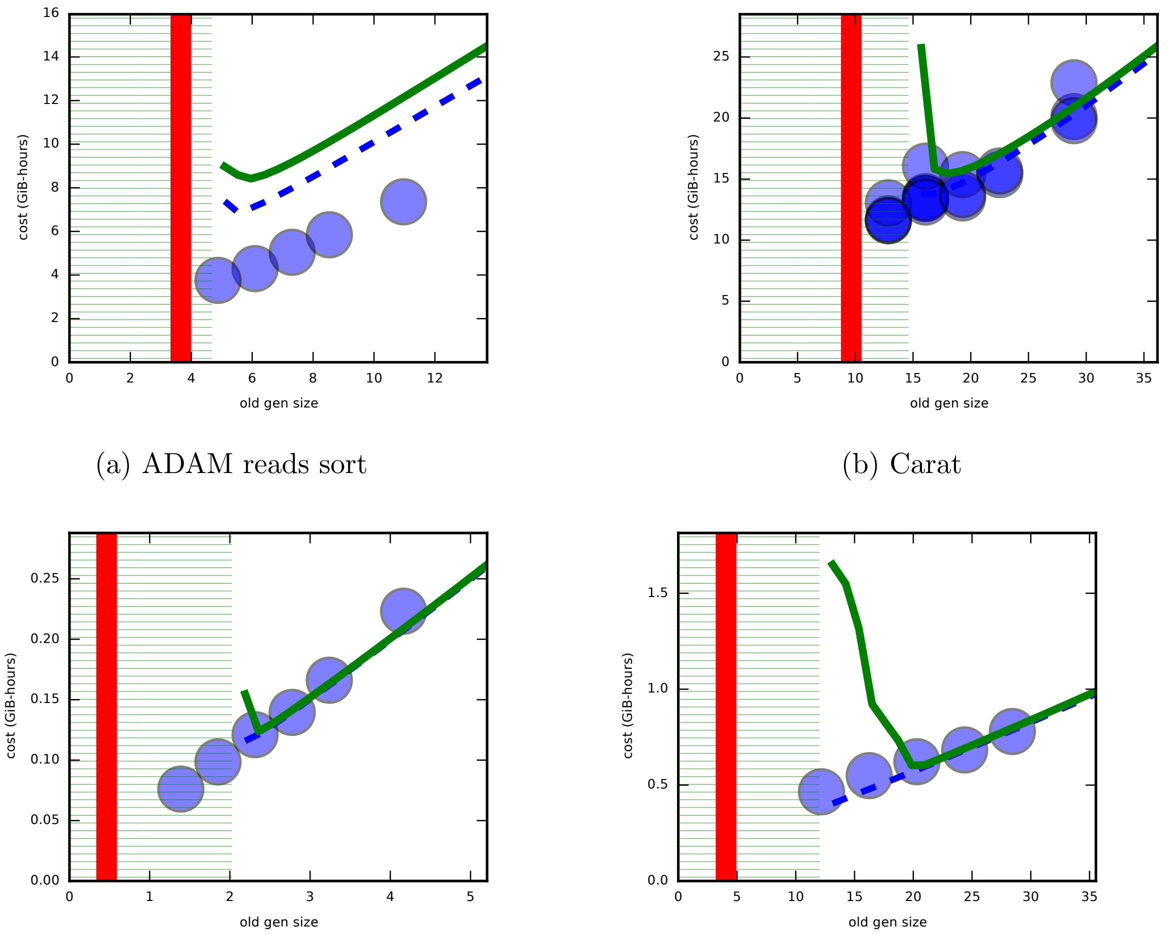
To evaluate predicted costs, I examined the change in SLAMR's predicted and the measured actual costs as each of the eden and tenure space size configurations were varied. Qualitatively, this should show whether the minimum cost matches the goal of being 'beyond' the performance knee, giving the effect of choosing a 'minimal' value for the garbage collector parameters. It also provides an indicator of accuracy. Although I evaluated whether overhead estimates are consistently conservative in Chapter 6, if overhead estimates were substantially more biased for large configurations than small ones, the resulting recommendations could still be too small. As this evaluation shows, that is not the case.

Figures 7.3 (tenured space size) and 7.4 (eden size) show the results. The cost function for each of these is memory used times runtime, without attempting to account for 'fixed' overheads for the operating system, driver program, or setup time. The graphs show two types of estimated costs: the solid represents a more conservative model while a dotted line represents a less conservative model. The less conservative model makes changes to two parts of the garbage collector model: First, it does not require there to be room for the entire eden space in the tenured space to account for variation in eden space size. Second, it uses the mean amount promoted in a local window of eden sizes rather than the running maximum of the mean amounts. In particular, this makes it less sensitive to outliers in the amount promoted, which makes a significant different for some programs.

In most cases, the estimates are based on the first run with the final framework configuration The configurations are varied around the recommended configuration based on that execution. In a small number of cases, the garbage collector configuration suggested based on this run used a different number of workers, so the evaluation shows estimates cost from an instrumented run with a matching number of worker nodes. I separately evaluate sensitivity to estimating costs with different numbers of worker nodes in Section 7.3.3.

For tenured size (Figure 7.3), there is a clear minimum for estimated cost. For the actual cost at and above this minimum value, the corresponding predictions are accurate, but they are too conservative below. For some programs, the measurement of the minimum amount of space required is too conservative because of constraining recommendations based on Spark's conservative size estimates or because there is insufficient data in the training runs to know the true maximum memory occupancy. As a result, configurations are considered infeasible when they are not.

For eden size (Figure 7.4), generally my model predicts a steep incline in cost for very small eden sizes, but the actual incline is more modest. This is dictated by the range of sizes covered by the example run. For eden sizes at the bottom end of this range and below it, there are few samples of the actual promotion behavior. Thus, the model conservatively predicts a larger amount will be promoted based on the behavior actually observed at slightly larger eden sizes. This dictates the recommendation in most cases. In some cases the predicted cost values are higher by an essentially fixed amount; this reflects cases where the full collection time because the training run had a very different tenured space size. Notably, this error does not substantially effect the overall recommendation.ch, reflecting the relative stability



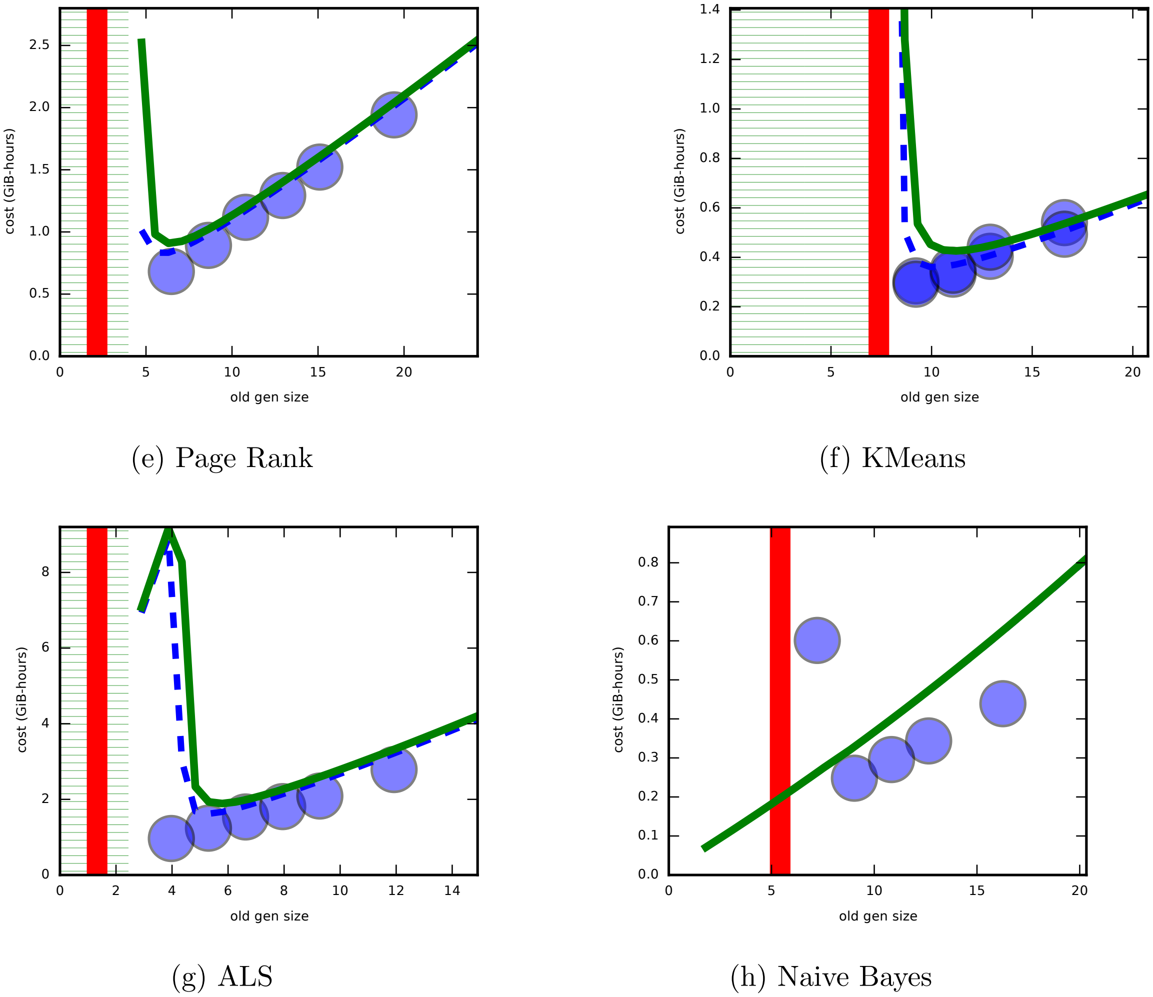
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 7.3: Estimated costs (lines) measured by memory used times time for varying tenured space sizes and single node configurations versus actual cost (circles). Two models are shown, one conservatively assuming that space for all of the eden space needs to be available in the tenured space (green, solid) and one ignoring this effect (blue, dashed)



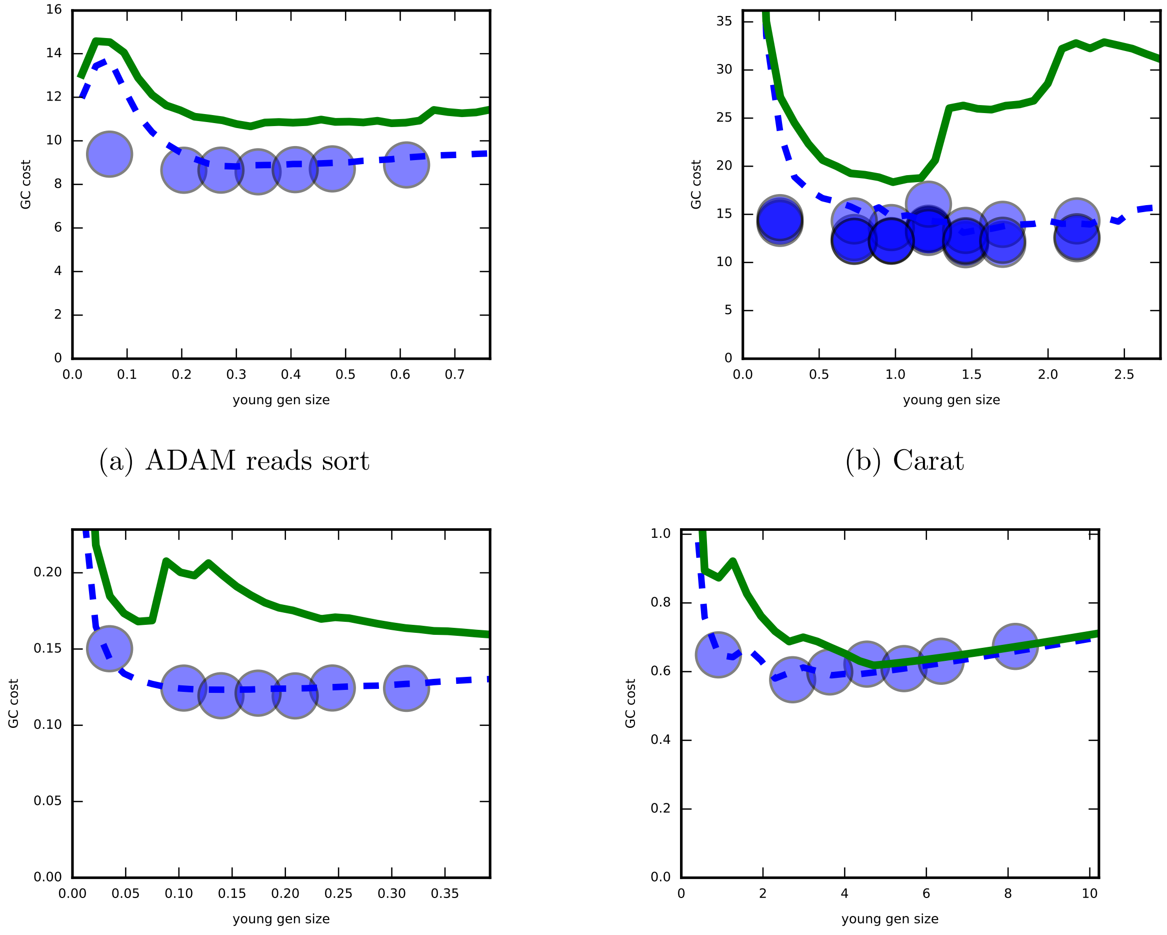
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 7.3 (con't): Estimated costs (lines) measured by memory used times time for varying tenured space sizes and single node configurations versus actual cost (circles). Two models are shown, one conservatively assuming that space for all of the eden space needs to be available in the tenured space (green, solid) and one ignoring this effect (blue, dashed)



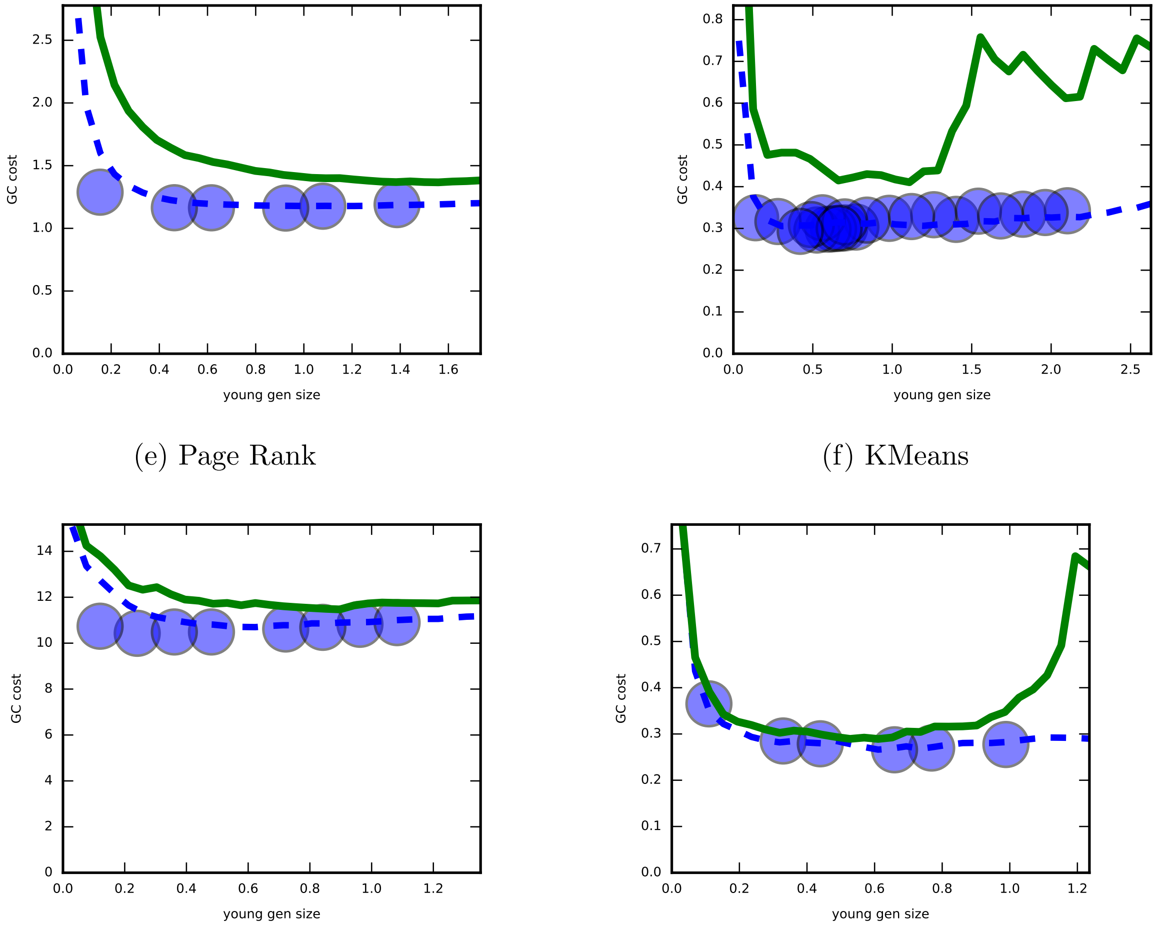
(a) ADAM reads sort

(b) Carat

(c) EM-LDA

(d) GLM

Figure 7.4: Estimated costs (lines) measured by memory used times time for varying eden space sizes and single node configurations in terms of memory-size and time (without fixed costs for a master node, startup costs, etc.) versus actual cost (circles).



(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 7.4 (con't): Estimated costs (lines) measured by memory used times time for varying eden space sizes and single node configurations in terms of memory-size and time (without fixed costs for a master node, startup costs, etc.) versus actual cost (circles).

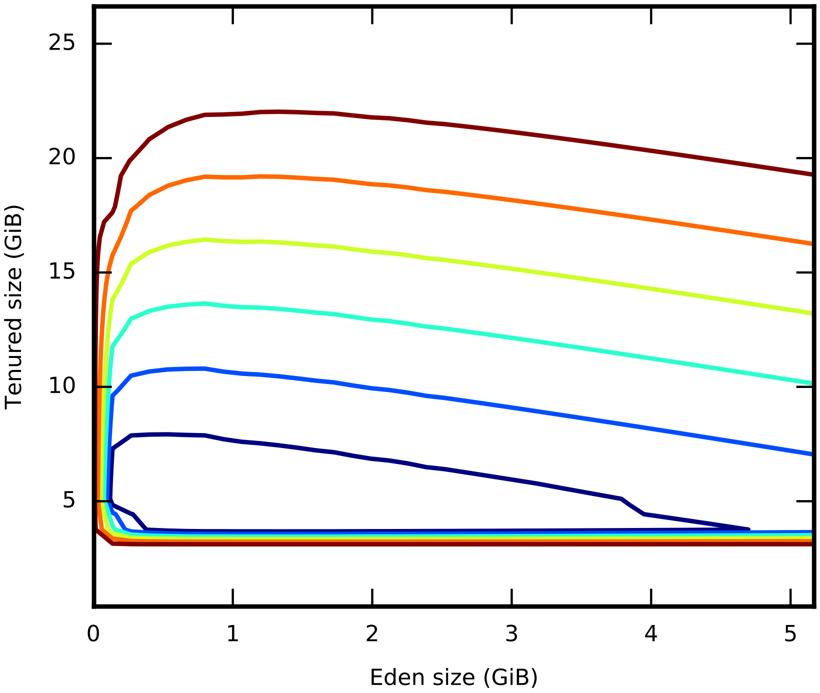


Figure 7.5: Contour plot of estimated costs for garbage collector configurations for ALS examples.

in the amount promoted

In some cases, like for the naive Bayes example, there is a sharp increase in the predicted cost after a certain point. This is a result of the heuristic that full collections occur when tenured space cannot fit the entire eden space for the less conservative model. In this configuration there is relatively little free space in the tenured space, so that constraint is met, even though it turns out that the corresponding bad behavior does not occur in practice.

There is an exception in some cases, which is caused by the amount of ‘extra’ space in the corresponding (constant) tenure size configuration being close to the eden size. As the eden size increases, our heuristic that there must be enough space for the entire eden space to be copied into the tenured space to make full collections seem worthwhile is binding. Making a corresponding increase to the tenured space size would avoid this.

Figures 7.3 and 7.4 show the effects of changing eden and tenured size are shown in isolation. In practice the optimal configuration is decided by the tradeoff between the two values. Finding a configuration effectively relies on being able to efficiently search for good tradeoffs. For this to be practical, it would be helpful for the estimated costs to be smooth in the region of viable costs, so that local optimization would be a viable approach. To test the viability of this approach, I examined the shape of the estimated cost space, shown for the PageRank example in Figure 7.5.

7.3.3 Cost consistency

[p]

[p]

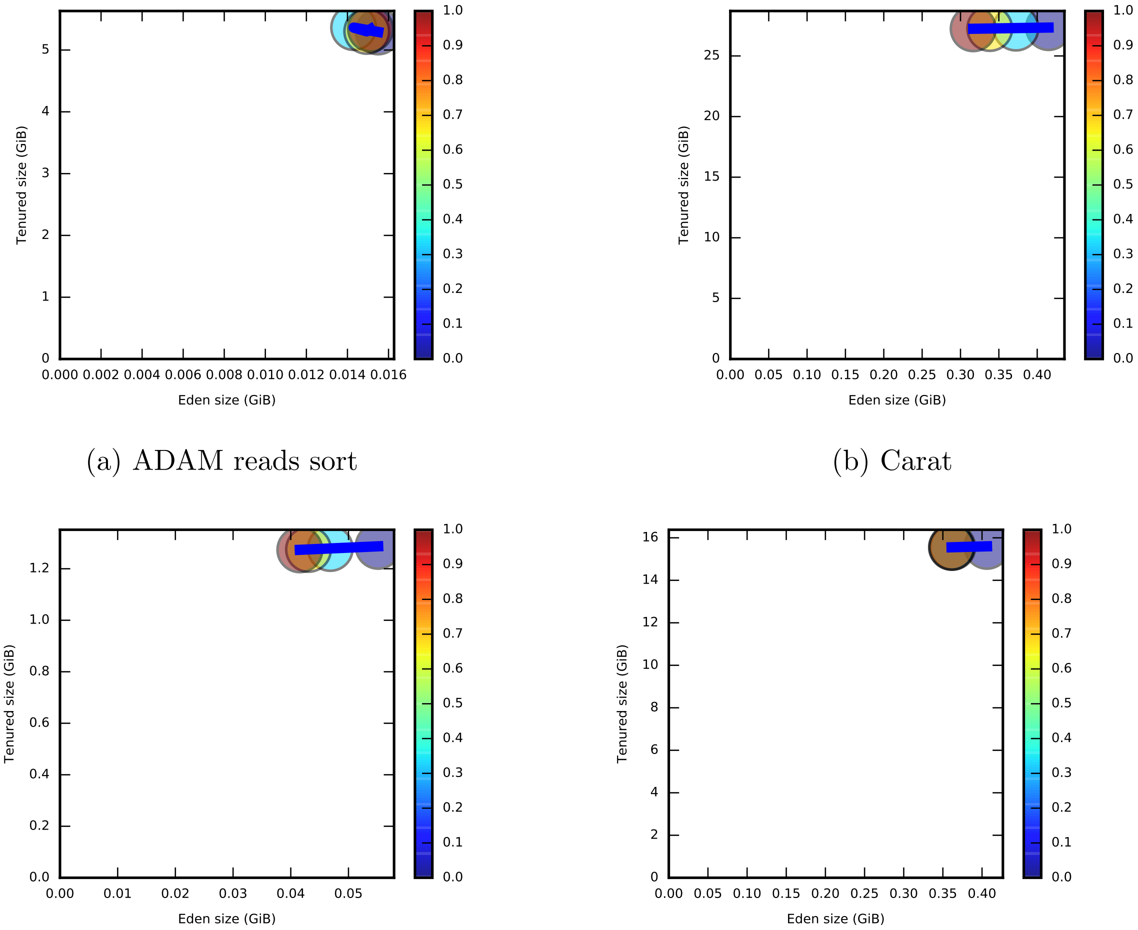
The cost estimates shown in the previous section generally use runtimes measured with a particular number of workers and using this to infer the runtime with a different configuration on the same number of workers. In reality, I would like to be able to take a configuration with an inappropriate number of workers and use it to recommend a more appropriate number. To do this, I made the same scaling assumptions described in Section 7.1.1. These scaling assumptions appear to have little effect on the overall recommendations.

To discover this, I examined how the recommendations based on inferring for a different number nodes than the training run. I varied how runtimes were estimated to choose the optimal configuration and examined the resulting change in minimum cost configurations. In Figure 7.6, I add a 'startup cost' to the estimated runtime of a varying proportion of the total time and see how the recommended configuration changes based on that. In Figure 7.7, I show predicted worker configurations given four workers and four cores per worker — more than each of training runs at issue — given estimates of varying portions (from 25 to 100%) of the workload being serial (color scale).

In both cases, the overall recommendation in terms of total memory allocated is quite stable. The amount of tenured space suggested is stable, varying by around 10% in the most extreme cases between assuming the entire program will enjoy parallel speedup and the entire program is serial. If one ignores the configurations that assume almost all of the program is serial, then the variation is well less than 5%. The amount of eden space recommended is less stable, though the absolute differences are small and, as shown in Figure 7.4, there is not much change once a threshold is reached. Thus, given the small change in absolute memory usage, the variation in eden size is from a set of configurations of approximately equivalent cost.

7.3.4 Finding minimum node counts

To evaluate our heuristic for estimating the impact of scaling, I examined the resulting estimated number of nodes versus performance as the number of nodes changed. Figure 7.8 shows total runtime as the number of nodes varies for several programs, along with the number of nodes estimated based on the heuristic that 50% of the execution in the observed run is serial. These programs are the indicated example programs, with smaller worker sizes to better illustrate the worker count choice. To find configurations for the smaller node counts than SLAMR's recommendation for comparison, I first tried to make a configuration that satisfied Spark's restrictions; if that failed, I tried to make a configuration ignoring non-JVM (shuffle data cached by OS) requirements; if that failed again, I copied the configuration



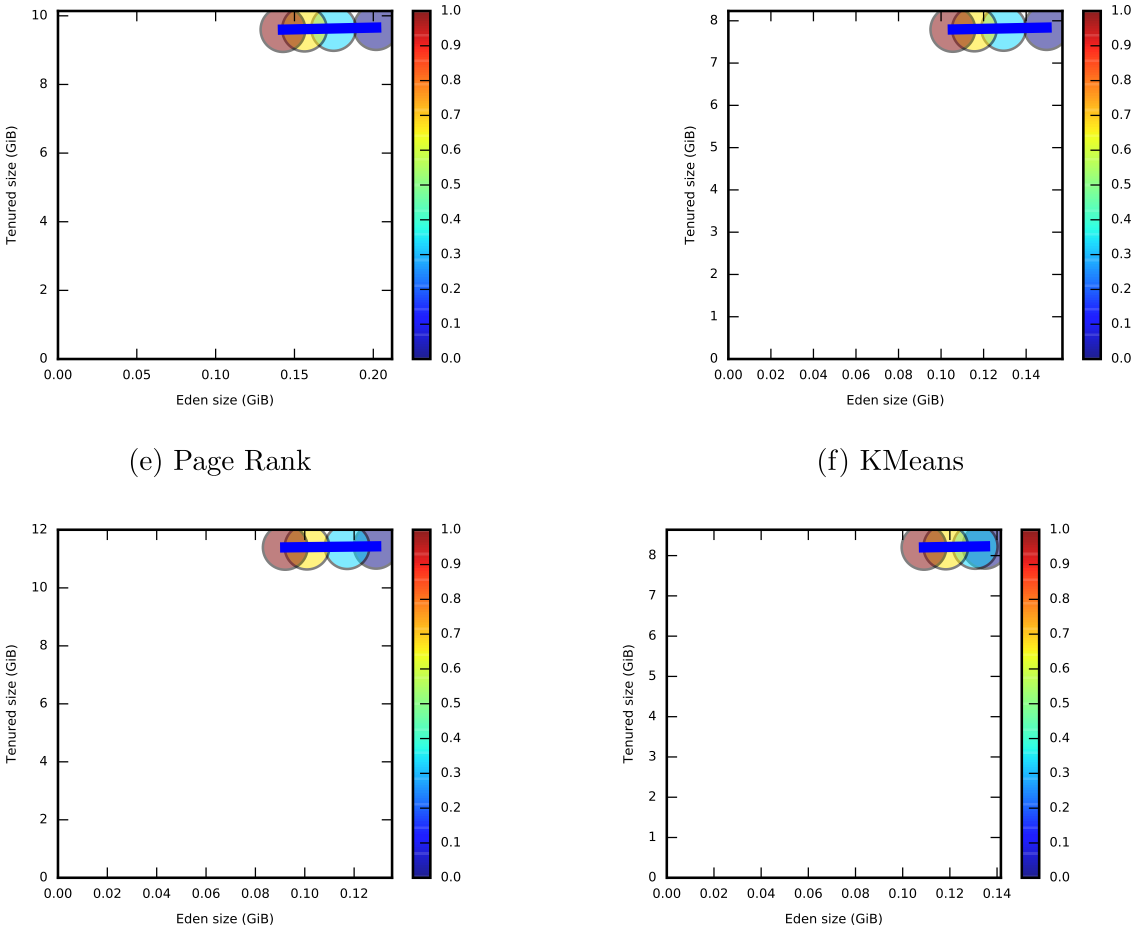
(a) ADAM reads sort

(b) Carat

(c) EMLDA

(d) GLM

Figure 7.6: Change in the lowest 'utility' configuration applying a variable penalty (ranging from 0% to 100% of the initial runtime) to the estimated runtime to account for less than predited changes in runtime.



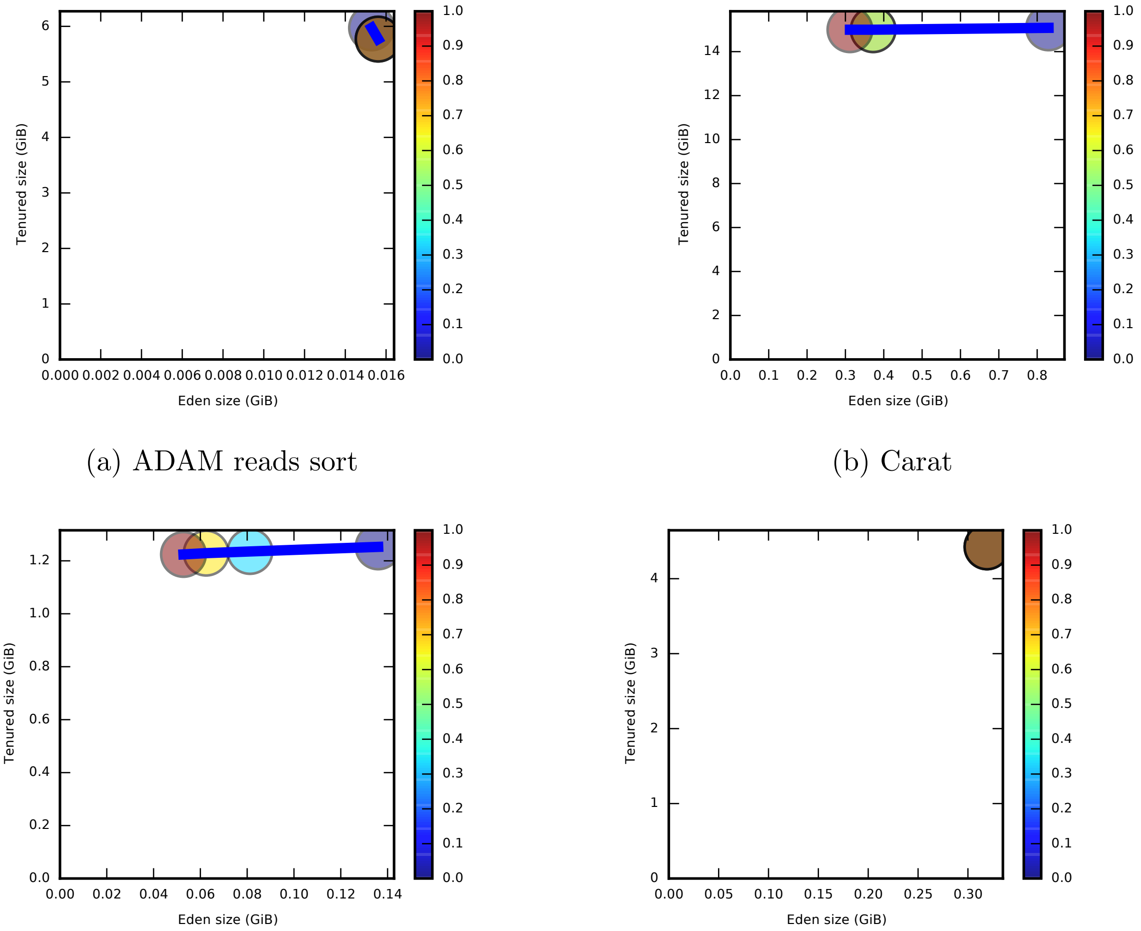
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 7.6 (con't): Change in the lowest 'utility' configuration applying a variable penalty (ranging from 0% to 100% of the initial runtime) to the estimated runtime to account for less than predited changes in runtime.



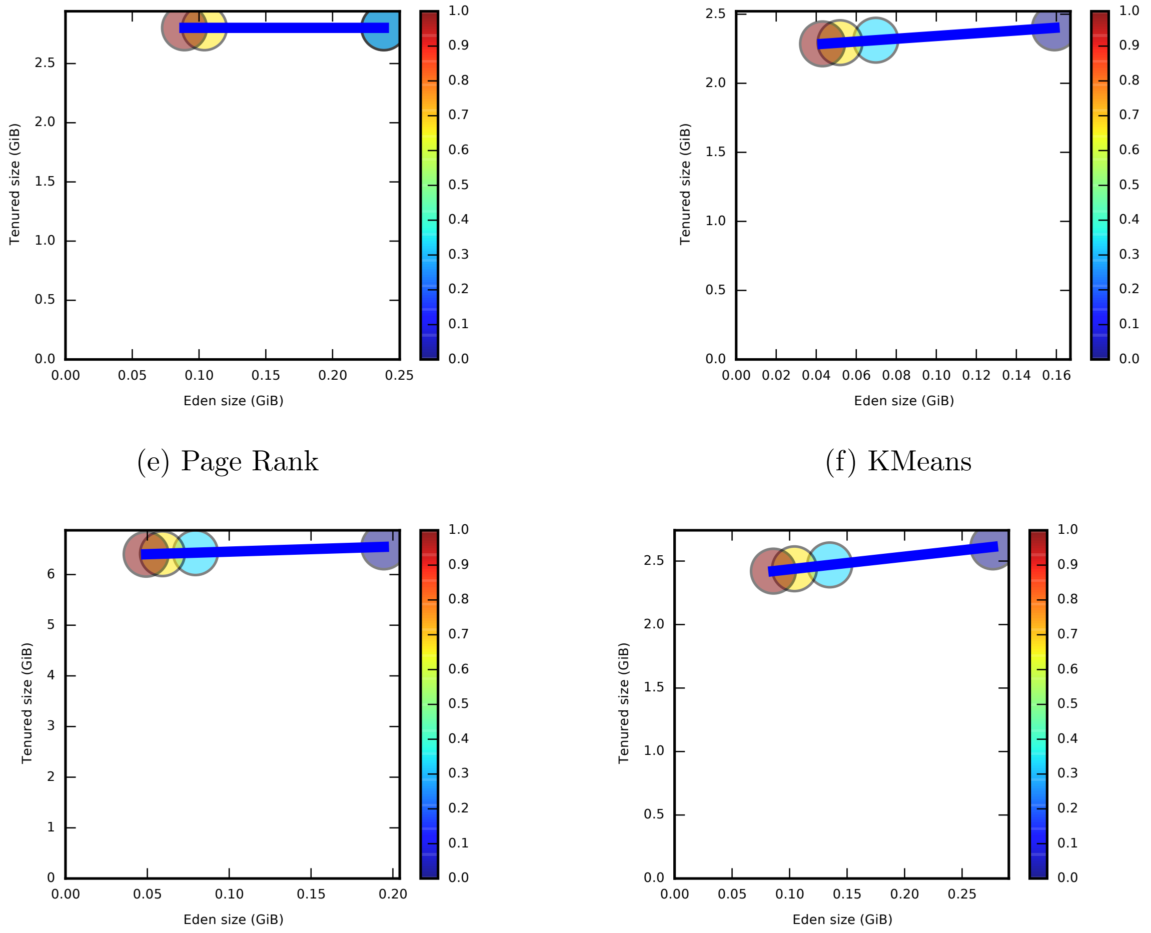
(a) ADAM reads sort

(b) Carat

(c) EMLDA

(d) GLM

Figure 7.7: Change in the lowest 'utility' configuration for four nodes with four cores each given variable assumptions portion of the initial runtime that was parallel work.



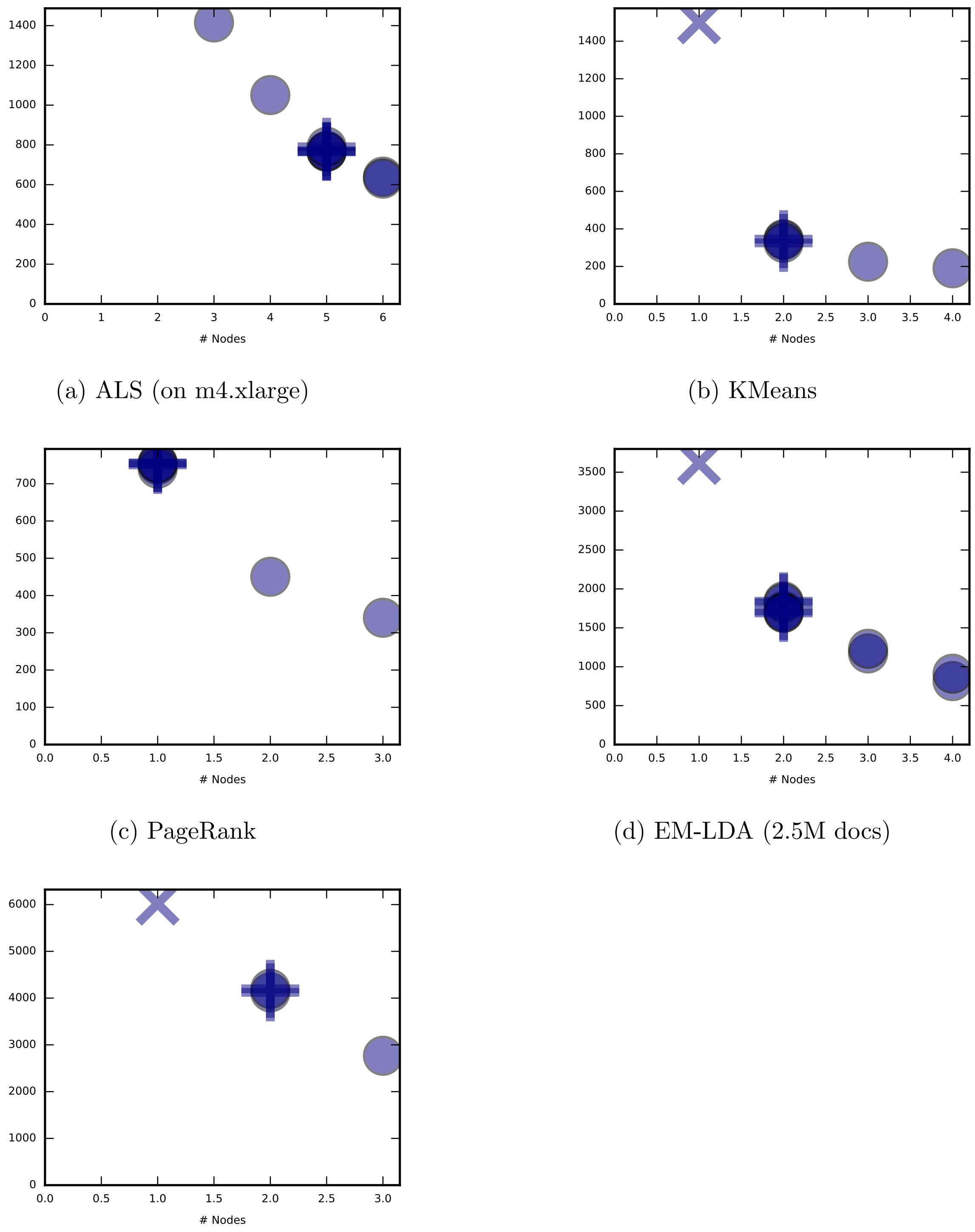
(e) Page Rank

(f) KMeans

(g) ALS

(h) Naive Bayes

Figure 7.7 (con't): Change in the lowest 'utility' configuration for four nodes with four cores each given variable assumptions portion of the initial runtime that was parallel work.



(a) ALS (on m4.xlarge)

(b) KMeans

(c) PageRank

(d) EM-LDA (2.5M docs)

(e) ADAM sort

Figure 7.8: Runtime with varying node counts versus recommended configurations (marked with +), assuming 50% serial overhead. Xs represent failing or timed-out runs

	Framework			**Runtime**	
Program	Storage	Execution	Sum	Tenured	+ Fit Framework
Carat	0% —9%	-5%—0%	-5% —9%	-20%—-16%	5%
GLM	-13%—0%	-0%—0%	-13%—0%	-1% —3%	-12%
PageRank	-4% —0%	-0%—0%	-4% —0%	-2% —14%	-34%
KMeans	-5% —0%	-0%—0%	-5% —0%	-11%—0%	-10%
ALS	-9% —0%	-0%—0%	-9% —0%	-16%—-5%	-49%
Naive-Bayes	-5% —0%	-0%—0%	-5% —0%	-5% —1%	-6%
ADAM sort	-0% —0%	3% —5%	2% —5%	-3% —0%	-10%

Table 7.1: Difference between predicted and 'optimal' value based on a sensitivity analysis. For framework parameters, the optimal value is chosen by counting recomputations (for storage space) or spilling (or execution space). For GC space, it is measured by finding the estimating best memory × time cost ignoring constraints on size from framework parameters (total tenured space size is allowed to be smaller than the framework 'memory' size). Values within 10% of the lowest observed cost are considered best (to discount random variation in runtime). Ranges represent the observations in the sensitivity analysis that bound the transition from the 'best' value to a non-best value that sit closest to the predicted point.

from the 'optimal' size. Although clearly this estimate is not accurate, it is sufficient to avoid the timing-out (taking much longer than twice the larger run) or failing configurations.

For most of these programs I identify the minimum working estimate for the given worker size. The result of increasing the worker count is at best comparable to the increase in number of cores assuming linear scaling. For two of the four programs I examined in this way, the program fails to complete successfully for smaller worker counts. For one program, the ALS example, my recommendation seems high. This is because the program uses a large amount of shuffled data. In an attempt to keep the shuffled data in memory, I require a large amount of page cache to be available. But, as shown in Section 5.1.4, this has some effect, but not a very strong effect on performance for this program.

7.3.5 Estimating Slack

To obtain an overall evaluation of my combined estimates, I estimated the total amount of 'slack'. Generally, the estimates are conservative — that is, they require more space than needed to meet configuration objectives. A fully accurate estimate would be no more conservative than necessary. My goal is to evaluate how close my overall recommendations are to this goal.

To do this, I look at two different kinds of accuracy. For framework estimates, I consider the most accurate estimate to be the one with minimum memory where the framework does not make use of alternatives to memory — recomputation or spilling. For garbage collection estimates, I use a utility-based approach, considering the most accurate estimate to be one which minimizes the 'cost' in terms of memory times run time. Since I assess accuracy of the garbage collection prediction alone, I ignore the limitations imposed by estimated framework sizes on the memory size that would ordinarily be part of recommendations.

Accuracy by this metric for several example programs in shown in Table 7.1. Values are shown in terms of the portion total worker memory consumption. Ranges represent the region in which the 'most accurate' configuration ought to occur based on measured runs. For framework configurations, this is between the largest-memory configuration with recomputation and spilling and the next highest memory configuration (with no recomputation). For garbage collector configurations, this is the configuration closest to the recommendation within 10% of the lowest observed cost. The margin should compensate for inconsistency in runtimes from minor changes in scheduling and I/O performance.

The results show that framework recommendations accurately identify when extra work is performed due to insufficient memory across programs which make different uses of Spark's abstractions. That additional work occurs does not indicate that this is the point of best performance; instead, it shows that the configurations are successfully in meeting the goal of keeping the computation in memory, effectively providing an upper-bound on memory requirements.

Garbage collector recommendations, on the other hand, are much more variable given that they lack such bright line criteria. But they consistently seem to be more conservative than necessary. Consistent with the observation that Spark can overestimate the amount of memory required for data it manages (Section 5.2.1), the largest source of error is the constraint the tenured space is large enough to hold Spark's estimated size. These estimates were consistently conservative but generally not by a large more than 20%. Given the variety of programs, many types of data (graphs, genomes, sensor readings, matrices, etc.) and a mix of programs dominated by transient or persistent data sets that were iterated over repeatedly, this shows that the garbage collector model is not limited to a particular kind of program that might be written for Spark.

For most of the programs, I observed, Spark required 10–30% extra space over what garbage collector measurements showed was actually used. I observed this misestimation go up to 50% in the case of an iterative graph computation which is likely to have lots of shared references (perhaps to graph labels) between different iterations. As noted in Section 5.2.1, this extra space may be fragile, in that it may become required space in unlucky cases such as where partitions are transferred between workers.

7.4 Summary and Discussion

By carefully combining models for different components in in-memory analytics, I developed SLAMR, a tool that produces consistently conservative memory recommendations for analytics programs. Converting the models into recommendations is non-trivial, with two problems: first, making sure the interaction of the components is properly dealt with, and second, using estimates of garbage collection overhead to drive configuration choices.

The primary constraint in handling the interaction between components is that it is difficult to infer how configuration changes in the framework affect the load on the language runtime. I used two strategies to compensate. First, I was careful about observing the language runtime workload. I provide extra space for the language runtime when the framework had a very different configuration rather than assuming that my tool was presented a realistic garbage collector workload. Second, I restrict the language runtime configurations based on the framework's measurements of memory usage. In practice, this restriction is often determines the memory requirements more than the actual garbage collector demands.

This indicates that, although I rather precisely evaluate what the framework measures as its memory occupancy, the framework's accounting often leaves something to be desired. The most likely cause for this is sharing between multiple stored items, such as in strings. If one was not concerned about failures from cases where this sharing does not occur, then one could simply rely on the language runtime estimates to configure and save a substantial amount of space. Unfortunately, this is risky because the sharing might easily be broken, for example, if the strings are serialized to the network due to a different computation pattern.

Another interaction between the language runtime model and framework model is turning overhead estimates into recommendations. My strategy for this problem was to produce a total runtime estimate and use it estimate a user's 'utility'. By accounting for the framework's scaling in way biased towards small sizes, this has a similar effect to choosing a minimum size. Because of the steep increase in overhead that occurs with small language runtime configuration parameters, this approach is not very sensitive to exact choice of accounting for runtime — important since I do not collect enough information to accurately estimate scaling.

Overall, I showed that I can quickly arise at a good memory configuration by observing normal executions of an analytics program. My procedure chooses these example runs such that they are conservatively, rather than needing to exhibit the bad behavior I wish these recommendations to avoid. Thus, they are suitable for running the program normally during development. Given the high accuracy of the framework model and how it often constrains the language runtime configuration, it is likely that every run after the first working example will be substantially better configured than any user guess — without risking failures that discourage users from tweaking memory configurations.

Chapter 8

Future Directions

Analytics frameworks and language runtimes provide incomplete support for modeling their memory usage. Most of the challenges are ones of visibility: instrumentation focused on performance modeling often provides little information about the decisions leading to that suboptimal execution. Increased visibility could make both writing memory analysis tools easier and allow them to provide more detailed and more accurate recommendations.

In the case of my memory models, there are several opportunities where my memory model could be generalized to improve recommendations in this way. These center around relaxing assumptions. I chose to model a simplified garbage collector, but the same approach could be extended to better reflect the wider options users have to configure modern garbage collectors. Also, in modeling my simple garbage collector, I relied on relatively coarse-grained observations of program behavior. More detailed observations should decrease the dependence on observing the program in a 'close enough' execution.

Similarly, when modeling the behavior of the Spark framework, I chose to measure the memory usage needed to keep the computation in memory, not judging any user choices about what to allow to be stored in memory. Evaluation of the performance tradeoffs of the example programs I explored revealed that this is not always a good choice. For many programs, the benefits of avoiding recomputation or filesystem I/O are minimal. The problem is that sometimes it matters substantially; the main challenge is obtaining enough information to determine when. Since the framework does not naturally encounter this behavior in a typical run, this requires special instrumentation.

In general, improving models of memory usage in analytics programs is limited by the visibility into those programs. Gaps in visibility come from where the underlying language runtimes and frameworks do not expose the right kind of information. Examining these gaps provides insight into both how to improve tools for users to configure analytics and how frameworks might make the work of these tools easier.

In this chapter, I review several avenues for generalizing my memory recommendation tool and designing future analytics stacks to make resource requirements more visible. I start by examining the state of instrumentation, both in analytics frameworks and in language runtimes, including more invasive but complete ways of extracting information than I used.

I then discuss several extensions to the models of garbage collector and framework memory usage I presented. For the garbage collector model, I discuss generalizing my model to handle more types of garbage collectors. For my framework model, I look at the tradeoffs involved in extending the model to provide users insight about what to keep in memory. I conclude with a summary of these challenges and opportunities for improving visibility into memory usage in analytics stacks.

8.1 State of Framework Instrumentation

Spark's instrumentation is extensive but matches imprecisely with monitoring its memory requirements. Spark's monitoring is driven by a combination of operational needs — such as deciding when to use secondary storage — and a desire for user feedback, usually through a web-based monitoring interface. This latter seems the primary reason for its 'event log' I use to collect data for analysis. The event log stores the same network messages used by the web interface and permits it to be archived to provide this interface for completed programs.

Much of the information Spark collects in the event log represents items that are inevitably communicated to the Spark driver and were thus trivial to log. This includes when tasks are started and stopped, since these are decisions made on the master in Spark's current architecture. It also includes what RDD partitions were updated by tasks, which the master needs to track to schedule tasks locally to their input partitions. Also included in this data is the size of each of these partition as stored in memory or elsewhere, which is tracked internally at the Spark driver for all workers.

After a task is scheduled on a worker, the decision about whether to read its data from local cache or recompute it is local. These decisions were not always apparent from the centralized event logs. It was from the logs at the workers but not in a form intended for machine parsing. Generally, this information was never sent to a central location, outside of external log collection systems.

Higher-level information used by the scheduler is also included in Spark's machine-parseable log. Spark groups tasks into jobs, representing the computation of a single output. These jobs are in turn divided into stages, representing all the computation up to completion or some shuffle step. For each stage, Spark records the RDDs it depends on and the dependencies between stages caused by shuffles. From debugging, this also includes a stack trace of where the stage was triggered, which might be useful for providing user feedback — although this often occurs deep inside libraries, such as implementations of machine learning algorithms.

This instrumentation is not quite enough to reconstruct computation dependencies. Spark programs often create RDD, then perform some operation that forces it to be computed, then use the result of this operation to construct new RDDs. For example, a typical implementation of a sort over the Spark core might compute a sample of the input to the sort, then use this to construct a partitioning for the actual sort operation. The dependencies tracked by Spark could omit the relationship between the final sort operation and the sampling operation.

Some other parts of Spark's instrumentation are driven by performance monitoring. Much of this seems due to [66]'s performance diagnosis work. This includes breaking down how tasks spend their time — between garbage collection overheads, computation and I/O wait. It also includes measuring how much I/O tasks do and from what source — which will show the effects of a failure of caching, but not identifying when the cause is a failure to cache data.

To support displaying these measurements and the web interface and perform the analyses like in [66], this information is collected and sent to the driver. The overhead is limited by piggybacking on existing task completion messages. To enable interesting statistics to be recorded anywhere, Spark internally allows look-up of the task the current thread is responsible for to add information to its future task completion messages. While this ensures that the effects of performance affecting decisions are accounted correctly, this decouples decisions to log when I/O or computation occurred without from the operations which trigger that work.

Generally, Spark's instrumentation of data sizes is opportunistic, so memory sizes are only recorded when Spark uses the size to make memory usage decisions. This avoids overheads, since size estimation involves a potentially expensive traversal of an object and its references, but it meant that some sizes I required for my analysis were unavailable. Spark did generally record sizes when they were easy to measure, such as sizes of serialized data or the amount of data read from or written to files.

8.1.1 Estimating Managed Data Sizes

To assess memory usage, it is important to have actual measurements of memory usage. In a framework like Spark that takes advantage of a managed language, this information is neither free nor exact. For cases where data is serialized, sizes are easy to obtain and often available centrally. For example, in implementing its shuffle, Spark sometimes centrally collects the size of each map output — one for each mapper/reducer pair — which has low overhead since this is the size of serialized data. But when data is not already in a serialized form, size estimation requires traversing trees of references, so Spark samples the data structure to control overhead even to measure the size of one item.

Such size estimation is inexact, sometimes significantly. For one example program I examined, the total gap between Spark's size estimate and the actual heap occupancy was around half of the total size. The most likely cause was overlap between objects. Given multiple items whose total size must be tracked, it is natural to add up the sizes of distinct items to tell if enough space is left. When these items overlap internally, however, this produces a number that is too high. Such overlap is a natural consequence of one data structure being transformed into another. Frequently such transformations will cause the resulting data structure to refer to the exact sample objects as the original — for example references to the strings may be copied rather than the strings themselves. Accounting for these shared objects is difficult.

Sharing breaks the normal expectations for storage management math. If objects A and B share space, then removing A and B together may save substantially more space than removing A or B alone. Eviction policies that account for this sort of sharing reasonably seem challenging, especially since systems like Spark lack the freedom to explicitly page out the unshared parts only like operating systems which have a superficially similar tradeoffs in managing shared memory can.

Frameworks like Spark could measure sharing by taking advantage of lineage. By comparing a value before and after a transformation, the framework could determine how much is shared in memory between the two. By only doing this for a sample of values, the overhead could be kept to a reasonable limit. A challenge for this approach, however, is if the individual values are large, containing many references.

To determine the total sizes, frameworks can randomly sample references to get a good overall sample of size of a data structure. But taking two random samples of references to measure the amount of overlap between two data structures introduces considerably more sampling error, thus requiring more samples. One could attempt to sample the two values related by lineage in the 'same way', but this seems likely to break if, for example, one is an array produced by filtering out values that match a predicate from the other, so every reference is at a different index.

Besides measuring sharing, it is not clear how useful such information would be since sharing might be disrupted by small changes to where partitions are computed. In particular, this makes estimates of sharing risky for making memory configuration recommendations — small changes to executions might erase the memory savings from sharing. If, instead of computing a partition on the worker node where its dependency was, it is computed on a different worker, then there will be no sharing. Even if later the dependent partition is recomputed on the new worker, there is still likely to be no sharing. Instead, a new, entirely separate copy of the value will be created. Similarly, 'no-ops' like saving a partition to the local filesystem and restoring are likely to create new copies of data, destroying any savings from sharing.

8.2 Gaps in Garbage Collector Monitoring

The garbage collector models of Chapter 6 use simple measurements and a model a simple garbage collector. These simplifications make it practical to bound the garbage collector overhead using measurements obtained with low overhead. More sophisticated measurements and more complete models might make these bounds tighter, especially for applications where low measurement overheads are not a priority. These tighter recommendations would be enabled both by increased precision of the model and the ability to account for additional garbage collector features.

The measurements of garbage collector activity in Chapter 6 avoid requiring invasive instrumentation of the language runtime. Without this limitation, it may be possible to obtain much better information about allocation and garbage collector activity. In particular, this could allow measurements of garbage collector performance to be better separated from measurements of allocations.

For example, an eden collection consists both of object scanning costs, object relocation costs, and actual copying costs. Some of these costs are related to the number of references or number of objects to scan or relocate; some are more related to the amount of data contained in these objects. Lacking easy visibility into the mix of objects, I captured the total time as a function of promotion size for a particular program — including both the time taken for object scanning versus copying and what particular mix of time was necessary for that program. An alternate approach would be to instead capture the object scanning time, object relocation time, and copying costs explicitly, then separately measure the object counts, sizes, and number of traversed references.

I experimented with obtaining some of these measurements from OpenJDK. The instrumentation is fairly lightweight — counting the number of objects or references encountered in a garbage collection pass in addition to the number of bytes. Generally, it appeared to provide little benefit to accuracy versus predicting from total size. The likely reason is that most of the programs I examined have roughly the same average object size, so the added information did not substantially improve precision.

This approach should be more useful when measuring the distribution of objects is decoupled from sampling garbage collection overhead. There are two sorts of notable scenarios where this might happen. First, this would allow one to use object size distribution measurements on one system to infer the behavior of another. This, notably, would allow one to make inferences about large changes in language runtime, such as estimating the effect of switching from 32-bit to 64-bit pointers. Second, it would compensate for cases where garbage collection time measurements might be available but would not cover the actual region the garbage collector would operate in — or would not cover it accurately. Generally, more direct measurements of object lifetime distributions should be less sensitive to 'noise' from other activity on the machine or apparently random variation.

One way to obtain such direct measurements is using the Java Virtual Machine Tool Interface (JVMTI) [65]. Unlike aggregate garbage collection statistics, this mechanism allows very fine-grained tracking of object sizes. JVMTI permits an instrumentation agent to 'tag' an object and learn when tagged objects are freed. Combined with instrumentation of object allocations, this allows tracking of object lifetimes and the distribution of object sizes with each particular lifetime. Example results are shown in Figure 8.1 for a partial run of PageRank (using the LiveJournal dataset) on Spark. In addition to obtaining information about the distribution of object sizes, this technique can obtain useful data over a wider variety of lifetimes than simply observing young collections would allow given eden or survivor space size settings.

Unfortunately, this instrumentation has very high overhead, even if it only tags a small sample of objects (in this case biased by object size to avoid undersampling large arrays that

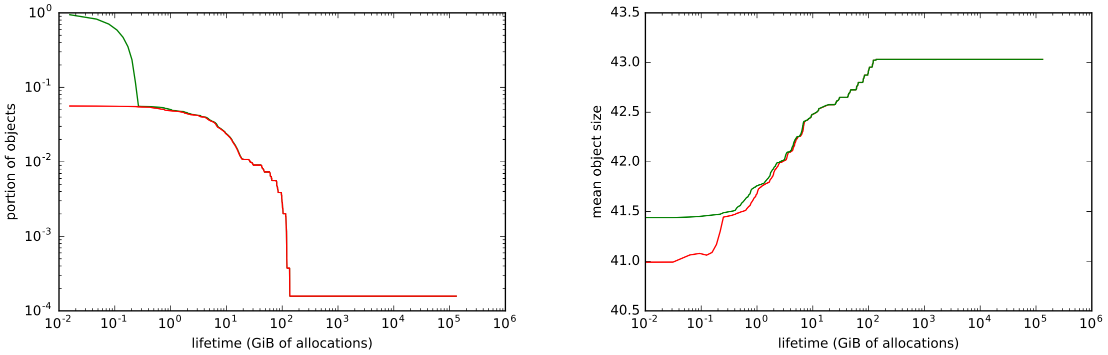


Figure 8.1: Portion of objects alive (left) and mean size (in bytes) of objects alive (right) after a given amount of allocations have passed for a partial execution of PageRank on Spark. The two lines represent an upper and lower bound. Note that below the eden size used in this execution (1/8th of a gigabyte) there are very poor bounds on the object lifetime, reflecting that information is only obtained after a garbage collection.

account for most memory usage). In addition to the actual cost of scanning objects on each collection, to obtain good bounds on lifetime, one needs to periodically force full collections and/or force the eden size to be small — a very large source of overhead. The instrumentation also has the danger of breaking escape analysis, an optimization where a heap allocation is transformed into a stack allocation. The instrumentation relies on transforming Java bytecode to call a native function to 'tag' an object. So to the optimizer, it will appear that the heap allocation is always necessary. With more support for the language runtime, this problem could be avoided.

By combining this type of detailed measurement of the lifetime distribution of objects with detailed microbenchmarks of a garbage collector, one might obtain more precise predictions of garbage collector performance. In particular, since this measurement approach, unlike mere measurements of the amount of live memory after a particular garbage collection, gives upper and lower bounds on object lifetimes, so it could more accurate bounds on garbage collection overhead.

8.3 Handling More Garbage Collectors

Another reason to investigate detailed monitoring of program allocation activity is that it would aid in extending my approach to other garbage collectors. This is especially true if one were to be less concerned about having conservative bounds on garbage collection compute time than about its performance more generally. Most notably, alternate garbage collector designs may allow garbage collection to be overlapped better with I/O or avoid long pauses that are inconvenient for streaming or distributed computation.

Two notable variations that real garbage collectors use that are not captured by the two-generation model of Chapter 6. One is garbage collectors with a different generation structure. Typical generational garbage collectors use survivor spaces, so there is more opportunity to filter out objects before they reach tenured space. There are also collectors with a more dynamic generational structure, like the OpenJDK's Garbage-First ('G1') Collector, which can divides memory into regions which it can process in any order. Separate from region structure, many garbage collectors are concurrent. Instead of stopping a program to perform garbage collection, they perform it asynchronously, which is beneficial if the program has spare compute-cycles, as, for example, analytics programs which spend a lot of time waiting for misbalanced stages to complete might.

8.3.1 Survivor spaces

The simplest way to add survivor spaces to the two generation model I propose is to continue the 'steady state' assumption. Under this assumption, one assumes that the distribution of object lifetimes is fixed. To be more precise, following the approach of [88], one can model the lifetime of objects as continuous function $L(x)$ representing the portion of objects which will still be reachable after x more bytes are allocated. Given this, the portion of objects surviving an eden collection for an eden size of E is $E \int_0^E L(x)dx$, since the first byte allocated after the previous eden collection needed to survive E bytes being allocated and the last before the eden collection 0 bytes.

To keep things simple, I will start by assuming that there is one relevant parameter for the survivor space: the threshold number T of minor (eden and survivor) collections. An object must survive before it is promoted to tenured space — the objects 'age'. This will implicitly determine the survivor space size. In practice, OpenJDK tries to adjust this parameter based on the survivor space size, and any collector may need to promote objects earlier if the survivor space fills up.

The objects that will be promoted in one eden collection are objects that survived T eden collections. The size of objects in survivor spaces is then $\int_E^{(T-1)E} L(x)dx$ (neglecting cases where the program ran out of room the survivor spaces). There are two approaches to applying this formula given actual measurements: one can use measurements of the amount promoted, and the amount surviving to approximate the function $F(y) = \int_0^y L(x)dx$. In particular, in addition to obtaining a value of $F(y)$ from an collections at an eden size y, garbage collectors, including OpenJDK, can indicate how much of the survivor space is used by objects of each age.

This provides an estimate of the amount of data in survivor spaces and the amount promoted, but does not indicate the amount of time spent on survivor space collection. To adjust the model of Chapter 6 to handle that, one can modify its time estimate for eden collections. Instead of the time spent being proportional by the size of promoted objects, it should be proportional to the size of objects either promoted or remaining in survivor spaces. In generational collectors, surviving objects are usually copied after every minor collection rather than incurring the additional complexity of handling fragmentation.

A problem with the approach above is that it assumes a steady state — an assumption that is not true as illustrated in Section 6.3.1. Variation from steady state is very problematic for modeling survivor spaces. In particular, it is common for analytics programs to have periods when a significant portion of objects are long-lived — some dataset is being loaded — and periods when few objects are long-lived — a loaded dataset is being processed. This pattern decreases the impact of survivor spaces. During these 'loading' periods, the survivor spaces will be exhausted, and during the 'quiet' periods, the survivor spaces will be comparatively underutilized.

One simplifying assumption that would aid in modeling these affects is that the amounts remaining after different number of minor collections are highly correlated. Namely, the times when the largest quantity of objects survives K eden collections are the same as the times when largest quantity will survive after $(K + 1)$ collections. Given this assumption, one can segment the distributions of the amounts of objects surviving each number of eden collections based on the how many minor collections could occur before survivor spaces would be exhausted. The garbage collection overhead for each these segments could then be estimated separately.

This segmentation assumes variation in the maximum age allowed in survivor space will match the variation in the amount surviving rather than this maximum age, for example, being set statically. For OpenJDK configured with a static survivor space size, this behavior would depend on the control loop in [88] closely approximating this behavior as it attempts to optimize throughput. If one also allows that control loop to dynamically change the survivor space size (the default behavior), one could similarly assume that it is behavior will approximate the optimal size for each region of portions promoted.

8.3.2 Concurrent collectors

Probably the most common concern about garbage collectors commercially is pause times, that is times when application code is not allowed to run to let the garbage collector proceed. For things like web services and online transaction-processing systems, these have the most user visible effect on performance. To reduce pause times, many garbage collector designs try to run concurrently with the application. This not without cost. One cost in additional synchronization required to ensure that application activity does not interfere with the collector and vice-versa. Another cost is from what is collected: where after a traditional garbage collector completes, there will be no dead objects in memory, a concurrent collector will miss some of the objects made unreachable while it was performing its collection pass. Consequently, a concurrent collector is likely to have somewhat increased memory requirements to achieve the same average overhead as a non-concurrent collector.

There are two problems for modeling garbage collector overhead for these collectors. One is the different amount of objects they collect. To model this conservatively, one can consider a simple form of concurrent collector — one imagines a two-generation collector, but whose full collections only collect objects that were free immediately after the prior collection. This would imply that the collector takes a 'snapshot' of live objects after it finishes doing a collection and immediately start the next collection. In practice, concurrent collectors that take snapshots like this often start later, but this approach would seem consistent with the conservative overhead estimation approach.

When concurrent collectors are most beneficial, it is because it is possible to overlap time when useful computation cannot be done with garbage collection activity. So, to decide how to use concurrent collectors, one should estimate how helpful this overlap is. The simplest way to estimate this would be to measure the compute usage of the underlying program to identify the aggregate number of compute cycles that are free. Problematically, this is another case where the actual performance gains are likely to be sensitive to different phases of program operation — only if the times when the analytics program does not pin the CPU coincide with times when it needs garbage collection is the concurrent collector going to be helpful. To account for this, one could group the measurements memory allocation activity with corresponding measurements of compute demand.

For example, one could split the program execution into several fixed-sized segments, and model each segment separately to limit the effect of the mismatch between where free compute time is and where garbage collection is required. But generally, this does not lend itself to an approach where one can reasonably say that overhead prediction was a upper-bound on the actual overhead.

8.4 Interactions with Shared Storage

My focus on analytics programs has assumed that an analytics program uses memory independently of other programs, with its memory managed within its own language runtime. Some designs for in-memory analytics try to share memory between otherwise independent queries. There are two common ways this is achieved: either by running one instance of a framework, with a common language runtime, that has sessions with multiple users, or by using an in-memory filesystem that allows frameworks to access the in-memory file directly (such as via Alluxio [2] or HDFS with short-circuit local reads [8]).

These approaches make it more efficient both when the same data is being analyzed by multiple users and when a single user has an analysis pipeline that combines multiple tools built on different, otherwise incompatible frameworks. For it to work well, the data must be manipulated without being transformed, likely requiring frameworks use the data in a form other than what is most natural for user-defined functions.

The techniques I use to estimate framework memory requirements can largely estimate requirements for these shared filesystems, which act similarly to a analytic's framework internal data store. Problematically, however, except in cases where the user allocates resources for a single pipeline, it is not clear how a user would use a recommendation — current practice does not have users explicitly request resources, and it is not clear what form requests should take when data is shared between multiple users.

8.5 Space/Time Tradeoffs

My configuration tool is premised on users making good decisions about what to make eligible to keep in memory. As my evaluation illustrated, for some programs, the benefit of keeping extra data in memory is minimal — and users would often be better off having some data be recomputed or reread from disk.

The primary difficulty in accounting for this is estimating the cost of recomputing or rereading data. Enabling caching is likely to the order of computations. When data is produced on demand each element is likely to be computed or deserialized just before it is used in the next stage of the computation. When it is cached, it is likely to be read or written sequentially from or to an array.

Adding instrumentation to measure the computation time or deserialization time of small slices of computation is likely to have high overhead, but is certainly possible. BigDebug[36] does this for Spark for the purpose of identifying values that are very slow to process, and reports around 20–120% (depending on the program) overhead for this type of monitoring. But it is not clear whether assessing the cost of the slice of computation will produce accurate estimates because of the changes to data locality and variation in I/O costs. To compensate for this, one could test changing the caching policy at the granularity of full tasks or similarly large units of work to estimate the difference in execution time.

Given an estimate of difference between caching and not caching — additional computation or serialization and I/O costs, a second problem is deciding which items to cache or not cache. A simple way to apply them would be a greedy strategy, finding which caching decisions results in the best reduction in estimated cost. Then, after each decision, one could update the costs of new decisions based on corresponding changes in the times to compute derived data from what would be cached. Doing this requires more examination of the execution dependencies than my trace-based analysis uses. From these dependencies, one can determine how the execution trace would change if a particular dataset were or were not cached. The greedy strategy might not produce the best results: for example, if two derived datasets have a common ancestor, caching that ancestor might be the locally best decision (starting from no caching) while caching the derived datasets individually might be globally best.

After obtaining a recommendation for what caching decisions to make, an important matter of practical difficulty is the separation between the framework-level decision and the user's action. This is a place where the universal interface of the framework's tasks and datasets is not helpful. For example, it is common for user's of Spark to use the framework through a SQL layer or a machine learning library, etc. Telling a user where a dataset was created that should or should not be cached is thus not actionable — this will often be asking them to make a change within a library. Even when it is contained entirely within their own code, the point to change may be part of a loop — either one needs to suggest which iterations matter or be sure that recommendations apply to all iterations of the loop.

8.6 Conclusion

Visibility into memory usage is imperfect. Even when one is only considering memory usage within an specific analytics framework like Spark, there are challenges due to the layers of abstraction within the it. My model relied on observing the lower level of Spark's abstraction and therefore is not so well connected to the actual computation specified by the user. Because of this gap, I needed to add instrumentation to connect higher-level decisions, like recomputation, to lower-level instrumentation. Frameworks would benefit from being designed with making such abstraction breaking easy and more general, especially as additional layers of abstraction, like shared in-memory storage systems, query engines, and libraries are required.

Similarly, for reasoning about garbage collection, while modeling more complicated garbage collectors is certainly challenging, there is distinct lack of visibility into program memory demands. Some of this seems simple to fix — for example, that is hard to get counts of objects instead of just sizes. Large benefits in modeling garbage collectors without simply trying a variety of garbage collector configurations might come from being able to better understand a program's memory demands. Unfortunately, techniques for measuring this well has high overheads because of the fundamental limitation that, beyond some sort of complicated static analysis, the only way to determine lifetime bounds is to force garbage collections.

The need to do extra work to force measurements also exists for framework configuration issues, but seems less problematic. A major gap in my model is that the assumption that users want to keep everything in memory does not appear to be appropriate, based on the actual memory tradeoffs I observed. With some assistance from the framework and certainly noticeable overheads, it seems likely one could estimate the effects of decisions about what to actually keep in memory.

Generally, through coarse-grained observations and simplified models of analytics stacks, I have provided useful memory recommendations. There is promise to improving these recommendations by dropping some of these simplifications or relying on more detailed instrumentation. In addition to expanding my work, the gaps in instrumentation I observed suggest how future analytics systems could improve visibility into their memory usage. A common theme was problems due to layering — true sizes and memory usage decisions were not known at the layer from which instrumentation was recorded. Through more complete monitoring and more complete models, future systems should be able to help users with more memory usage decisions compensating for the increasing complexity of their memory management.

Chapter 9

Conclusion

When users deploy in-memory analytics programs, they face some unfortunate choices. When they run their program on shared system, they are asked ‘how much do you want?’. And whatever they answer, they pay for, especially for relatively constrained memory capacity. In this process are users are not likely to receive feedback about whether their choice was good — except in the form of bad behavior when their guess is too low.

In this dissertation, I demonstrated that, with a negligible compute overhead and space overhead easily eclipsed by the resulting memory allocation savings, analytics frameworks can provide useful feedback about memory requirements. By taking advantage of the high level of abstraction of analytics frameworks, I can estimate memory requirements with confidence without, as is typical of prior configuration suggesting systems, needing to obtain a large number of samples to calibrate a model. Rather than focus on obtaining the most accurate cost/performance configuration, I emphasized making sure estimates were conservative. By doing this, I reduce user friction to using these estimates — users can start doing so immediately and should not feel they need to inflate estimates ‘just in case’. By reducing this friction, my system effectively addresses the main challenge of user memory configurations in practice — users have no clue and it is easier for them to request excess resources than spend a significant amount of time fixing it.

9.1 User Demands

I investigated how users misconfigure resource requests by examining a trace from a varied workload on a Google cluster. Comparing the aggregate allocated and used resources, it seemed around half the cluster’s memory and compute power was reserved and wasted. A more detailed investigation, acknowledging that the resources need to be allocated to support maximum rather than actual usage, suggests that the gap is somewhat smaller but still large — at least a third of resources ‘wasted’.

This gap occurred in spite of apparent efforts to correct it by the cluster scheduler. It appeared that resource usage was largely stable, and therefore perhaps predictable from past usage. The scheduler could and did overcommit resources, but did so very modestly. Presumably, this behavior avoided inconveniencing users by preventing their programs from running successfully by second-guessing their resource requests. This modesty, however, did not prevent the scheduler from imposing high costs in terms of injected failures in the form of evictions to make room for higher priority workloads.

These high costs of resource misestimation have become more of a problem as large-scale data analytics software has evolved. Originally the developers of analytics frameworks focused almost exclusively on scale and taking best advantage of spinning disks, the medium of choice for large online datasets. Then, with the decreased price of DRAM and increased use of analytics frameworks for problems that were not among the largest datasets in existence, later framework developers shifted to using memory. This increased interactivity and improved performance, especially for less straightforward computation.

One cost was increased sensitivity to memory allocations. No longer were the primary performance tradeoffs based upon distributed filesystem performance or core count. Instead, users needed to supply enough memory to store their working set of their program in whatever (often virtual) machines they allocated for their program. If they missed, performance would be very poor; if they did not, there was relatively little gain. On account of memory not before being such a bottleneck in before these in-memory analytics frameworks, little prior work had examined this problem.

9.2 Recommendation Goals

Faced with evidence of how poor memory configurations were, I built and evaluated a tool to recommend memory configurations for these in-memory analytics programs, for which these decisions are particularly important. Although in the cluster scheduler trace I examined users nominally configured a single memory size, a real recommendation involves accounting for several sizes, reflecting the several settings the user must configure. Configurations must account for different regions within the user's allocation with different purposes and how a user's memory allocation should be divided among multiple concurrent workers.

Underlying the configurations my tool SLAMR produced was the hypothesis that memory configuration has a sharp tradeoff. Below some size, users experience very poor performance from extra computation or I/O compensating for a lack of memory, but above that size, users experience little benefit. This hypothesis largely matched the results of my sensitivity analysis. Sometimes the tradeoff was not so severe, especially for memory usage that supported aggregation and many-to-many data transfers, where may be more dependent on network performance.

The assumption of this extreme drop off simplified data collection when modeling framework requirements. My tool did not need to measure the precise costs of computations, model filesystem performance and serialization performance — all of which would likely require considerably more invasive instrumentation. The assumption also made using coarse estimates of ‘utility’ a practical approach; the steep increase in costs alleviates concerns about sensitivity to small changes in cost estimates.

It also controlled my instrumentation overhead. As a result, the time overhead from that instrumentation was minimal, which it may not have been if I needed to take detailed performance measurements or compare different caching strategies. The main source of overhead was extra space to support the more finicky garbage collector instrumentation. But this should usually be made up for by the added memory-efficiency of my configurations over naive user estimates.

Ultimately, by focusing on safety and low overhead, I believe I provided configurations in a way which better meets user goals than a system that found the configuration with the best possible performance. I provide users perhaps more wasteful configurations, but quickly and with confidence about their potential negative consequences. In this way, my system avoids getting in the way of users whose real goal is to perform some computation.

9.3 Analyzing Program Traces

To make configuration recommendations, I recorded traces of the behavior of frameworks and language runtimes. These traces exposed data sizes and access patterns of the real program. Using these traces, I replayed the program’s behavior in a way that could account for a multitude of configurations. Key in supporting this analysis was carefully choosing metrics that were mostly independent of the configuration, reflecting the demands of the program and not the resources with which it happened to run.

By relying on these relatively low-level traces, I avoided depending on the detailed behavior of the analytics framework or a particular program’s memory allocation patterns. The cost was that I relied on executing programs to obtain low-level traces, rather than sampling the program’s execution. In addition avoiding dependencies on implementation details in the layers underlying the program, using actual traces limits concerns about sampling errors that hard to reason about when user-defined functions are involved.

9.3.1 Framework Analysis

For framework recommendations, I examined the places where the framework, Apache Spark in my case, maintained large amounts of data. In addition to identifying these large data accesses, I also made sure to distinguish which of these accesses caused by insufficient memory. Thus, I ensured my analyses had the information to infer what would happen when there was sufficient memory.

To make recommendations, I analyzed accesses to abstractions kept by the analytics framework in question. This let me handle differences in where data was stored (e.g., once on each worker versus on one worker) and what configuration parameters controlled whether its storage would be attempted. Given this separation, I could account for memory’s contribution to different configuration parameters, and the results of changes to the degree of parallelism likely to accompany larges changes in memory allocations.

To handle the very common behavior of frameworks and other systems acting like caches, I used 'stack algorithms' based on classic work from the processor cache design literature. These techniques permit efficient estimation of how much of a cache is 'necessary'. Such measurements are likely to be useful outputs for a variety of systems where users currently wold be likely to empirically sample cache hit rates instead of performing this post-hoc analysis.

This form of memory was by far most important to determining performance. It was important overall to model other sources of memory usage, where the framework constructs some large temporary data structure for each task or stores a large amount of data on each worker. But these programs were either relatively easy to configure for memoryA represented by pathological cases. Some programs either had very small memory requirements overall, and these dominated — they were essentially disk-to-disk transformations. Alternately, they were programs where the memory usage was only large because of a low partition count. In this case, experimental results indicated, that at least on my tests systems with fast (SSD-backed) disks, there was no performance benefit to avoiding higher partition counts. SLAMR's main benefit in this case would be identifying the space savings of a higher partition count.

9.3.2 Garbage Collector Analysis

Unlike in the above analysis, for garbage collectors, the abstractions provided by the language runtime do not nicely compartmentalize memory usage. Instead, I largely rely on measuring garbage collection activity and the aggregate sizes that the garbage collector sees. As with framework measurements, I took some effort to do this in a way that would efficiently capture activity relevant to a variety of potential configurations.

Capturing appropriate information required some more active intervention. Most notably, to obtain reasonable measurements, I varied the size of the eden (youngest generation) region as the program ran. This parameter permitted very fine grained sampling, avoiding problems from changing configurations that would arise from different 'phases' of program behavior. My examination showed such phases were common, making it unwise to simply rely upon measurements of average sizes and times. Exploring different eden sizes effectively measured the low end of the lifetime distribution. Using this, I could estimate the effectiveness of the generational assumption that more recently allocated objects are more likely to be garbage.

Otherwise, I focused on measuring values that would be roughly constant for a particular program, even if the garbage collector parameters changed. My choices were validated by checking that the parameters were constant for a variety of programs despite configuration changes. Similar to measurements based around varying eden sizes, these values reflected the mix of object lifetimes and sizes in the program rather than the garbage collector's configuration.

While my goal was to avoid dependence on configuration choice when measuring garbage collector activity, I was not entirely successful. My estimates of garbage collection time depended on the total memory allocated to a worker, which I could not effectively vary of a single execution. Instead, I needed multiple measurements to accurately account for this effect. Fortunately, this was rarely an important factor in overall recommendations. This is especially true for natural scenarios where the main issue is the number of workers to use rather than the size of each of those workers.

Changing the number of workers, however, was more of a concern. The ‘short-term’ measurements of program behavior were very stable, and even the total allocations performed across each worker was close to constant. But the total amount of space stored on each worker certainly changed. Fortunately, I could estimate this effect using framework-level information.

9.3.3 Comparing and Combining Models

I produced complete memory recommendations by combining the framework model with the garbage collector model. In actually applying these combined models, I needed to merge the minimum sizes for framework storage along with overhead estimates for the garbage collector. How this should be done varied depending on how the user pays for resources. Although looking at precise amounts of memory is more informative for evaluation, under common models of cluster provisioning the question is more of how many (virtual) machines.

Before looking at the overall recommendations my tool produced, my expectation was that the framework model would be the more accurate model and the garbage collector one less accurate. But there were surprising sources of inaccuracy with the framework model. One was that while the extra work was required due to insufficient memory, it had much less performance impact than I expected, at least in my testing environment with its relatively fast storage. More unexpected was that though configuration parameters that were close the minimum that avoided extra work, the usage explicitly tracked by the framework sometimes did not match actual memory usage.

One likely reason for this rational paranoia — similar to that of users when making their memory configuration decisions. To avoid crashes from memory exhaustion, it is better to overestimate how much memory one is using than spend any effort that would risk underestimating it instead. Another is that there is sharing between different sets of stored data that are hard for frameworks to account for when data is stored as ‘natural’ objects in a language runtime.

Except for the differences between framework accounting and language runtime accounting, the garbage collector estimates generally were less accurate. Part of this is because they are about overhead and not a Boolean decision, so mismeasurement of compute times can affect results. Another part is that there is less direct measurement of program activity available at language runtime level, where one must somehow deal with a large number of small objects.

9.4 Summary

Through instrumentation with at most negligible time overheads and space overheads easily recovered by the improved memory configuration, SLAMR efficiently provides users with recommendations about their memory usage. I focused on providing recommendations which would avoid memory exhaustion problems rather than finding the optimal price/performance point. This goal made constructing recommendations more efficient and better matched user incentives.

Failures and very poor performance encourage users to dramatically overprovision resources. I saw this sort of overprovisioning empirically in actual mixed cluster workloads typical of where analytics programs are run. Similar incentives would encourage them to add 'slack' on top of not-so-reliable recommendations. To compensate for this, I focused on constructing my recommendations to represent an upper bound on the necessary memory allocation. To lower friction for users and accommodate more ad-hoc analyses newer analytics frameworks allow, I also made these recommendations with minimal instrumentation overhead, piggybacking on the user's normal program executions.

My technique for producing recommendations was focused around collecting traces of the program behavior during these executions. By observing an actual execution, I did not need to account for much of the machinery of analytics frameworks and their underlying language runtimes. I measured the workload placed upon the framework and its underlying language runtime avoiding measurements that would simply reflecting its current configuration. I used approximate models of cache behavior and of garbage collector behavior, making approximations that may overestimate sizes to account for variable or hard-to-measure behavior.

These multilayered analytics stacks do not need to be black boxes. By observing a summary of the demands a program places on some layer of the stack, one can estimate the behavior of the stack in a variety of configurations. Thus, using observations generally sharing infrastructure with the mechanisms they use to make runtime decisions, these systems can demystify their resource requirements. This is particularly useful for memory requirements, which are increasingly important for end users to understand as large-scale data analysis becomes more accessible. Rather than making the deployment of analytics applications a guessing game, the layers of abstraction for managing data in these relatively new systems can be a boon to helping users understand their resource requirements.

Bibliography

[1] Rackspace® Public Cloud.

[2] Alluxio, Inc. Alluxio. http://www.alluxio.com/.

[3] Amazon Web Services. Amazon Elastic Compute Cloud Documentation.

[4] Amazon Web Services. Auto Scaling. https://aws.amazon.com/autoscaling/.

[5] Amazon Web Services. Elastic MapReduce. https://aws.amazon.com/elasticmapreduce/.

[6] Apache Foundation. Apache Whirr. https://whirr.apache.org/.

[7] Apache Software Foundation. Apache Hadoop. https://hadoop.apache.org/.

[8] Apache Software Foundation. HDFS Short-Circuit Local Reads. https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, and others. Above the clouds: A berkeley view of cloud computing. Technical report, 2009.

[10] Azul Systems. Azul Pauseless Garbage Collection: Providing continuous pauseless operation for Java applications.

[11] Lawrence Barsanti and Angela C. Sodan. Adaptive Job Scheduling via Predictive Job Resource Allocation. In *12th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2006.

[12] Bryan T Bennett and Vincent J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

[13] J. Bergstra, N. Pinto, and D. Cox. Machine Learning for Predictive Auto-Tuning with Boosted Regression Trees. In *Innovative Parallel Computing*, San Jose, May 2012.

[14] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

[15] J. Bilmes, K. Asanovic, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347. ACM, 1997.

[16] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *ACM Sigplan Notices*, volume 36, pages 353–366. ACM, 2001.

[17] Chris Bunch. *Automated configuration and deployment of applications in heterogeneous cloud environments*. PhD thesis, University of California, Santa Barbara, 2012.

[18] Callum Cameron, Jeremy Singer, and David Vengerov. The judgment of forseti: economic utility for dynamic heap sizing of multiple runtimes. pages 143–156. ACM Press, 2015.

[19] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 363–375, Toronto, Ontario, Canada, 2010. ACM.

[20] Christine H. Flood and Roman Kennke. JEP 189: Shenandoah: An Ultra-Low-Pause-Time Garbage Collector.

[21] Walfredo Cirne and Francine Berman. Using Moldability to Improve the Performance of Supercomputer Jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, October 2002.

[22] Cloudera. Crunch.

[23] E. G. Coffman, M. Yannakakis, M. J. Magazine, and C. Santos. Batch sizing and job sequencing on a single machine. *Annals of Operations Research*, 26(1):135–147, 1990.

[24] Databricks. spark-perf: Spark Performance Tests.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[26] P.A. Dinda. A prediction-based real-time scheduling advisor. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 10 –17, 2002.

[27] M. Frigo and S.G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[28] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the Cloud. *22nd International Conference on Data Engineering Workshops*, pages 87–92, 2010.

[29] Archana Ganapthi, Kaushik Datta, Armando Fox, and David Patterson. A Case for Machine Learning to Optimize Multicore Performance. In *HotPar*, 2009.

[30] Archana Ganapthi, Harumi Kuno, Umeshwar Daval, Janet Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting Multiple Performance Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE 2009*, 2009.

[31] Swapnil Ghike. Garbage Collection Optimization for High-Throughput and Low-Latency Java Applications. April 2014.

[32] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[33] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[34] Google, Inc. Google BigQuery Documentation.

[35] Google, Inc. Google Compute Engine Documentation.

[36] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 784–795, New York, NY, USA, 2016. ACM.

[37] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*, pages 630–637. IEEE, 2009.

[38] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 18. ACM, 2011.

[39] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R.H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011.

[40] M.R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda. Applications Know Best: Performance-Driven Memory Overcommit With Ginkgo. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 130–137. IEEE, 2011.

[41] Urs Hölzle. A fast write barrier for generational garbage collectors. In *OOPSLA/ECOOP*, volume 93, 1993.

[42] Hunt, Charlie and John, Binu. *Java Performance.*

[43] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72. ACM, 2007.

[44] Stephen A. Jarvis, Daniel P. Spooner, Helene N. Lim Choi Keung, Junwei Cao, Subhash Saini, and Graham R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Gener. Comput. Syst.*, 22(7):745–754, August 2006.

[45] Terence Kelly and Daniel Reeves. Optimal Web Cache Sizing: Scalable Methods for Exact Solutions. In *Fifth International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[46] M.A. Kozuch, M.P. Ryan, R. Gass, S.W. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. López, M. Stroucken, and G.R. Ganger. Tashi: location-aware cluster management. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, pages 43–48. ACM, 2009.

[47] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz. Napsac: design and implementation of a power-proportional web cluster. *ACM SIGCOMM Computer Communication Review*, 41(1):102–108, 2011.

[48] Palden Lama and Xiaobo Zhou. AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud. In *ICAC '12: International Conference on Autonomic Computing*, 2012.

[49] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[50] Henry Lieberman and Carl Hewitt. A Real-time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM*, 26(6):419–429, June 1983.

[51] Jimmy Lin. MapReduce is Good Enough? If All You Have is a Hammer, Throw Away Everything That's Not a Nail!, September 2012.

[52] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[53] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent memory and linux. In *Proceedings of the Linux Symposium*, pages 191–200, 2009.

[54] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, André Schumacher, Anthony D Joseph, and David A Patterson. Adam: Genomics formats and processing patterns for cloud scale computing. Technical Report 2013-207, University of California, Department of Electrical Engineering and Computer Science, 2013.

[55] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[56] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.*

[57] Microsoft Corporation. Microsoft Azure.

[58] Barzan Mozafari, Carlo Curino, and Samuel Madden. Resource and Performance Prediction for Building a Next Generation Database Cloud. In *CIDR 2013*, 2013.

[59] Netflix. Netflix Prize.

[60] G.R. Nudd, D.J. Kerbyson, E. Papaefstathiou, S.C. Perry, J.S. Harper, and D.V. Wilcox. PACE—A toolset for the performance prediction of parallel and distributed systems. *International Journal of High Performance Computing Applications*, 14(3):228, 2000.

[61] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14, New York, NY, USA, 2013. ACM. Oliner:2013:CCE:2517351.2517354.

[62] Frank Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Lab., CA (USA), 1981.

[63] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[64] OpenStack Foundation. OpenStack Documentation.

[65] Oracle. JVM Tool Interface: Version 1.2.

[66] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[67] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, November 1999. Previous number = SIDL-WP-1999-0120.

[68] Paul Menage. CGroups.

[69] J. A. Peacock. Two-dimensional goodness-of-fit testing in astronomy. *mnras*, 202:615–627, February 1983.

[70] Alexander Rasmussen, Michael Conley, Rishi Kapoor, Vinh The Lam, George Porter, and Amin Vahdat. ThemisMR: An I/O-Efficient MapReduce. Technical Report CS2012-0983, Computer Science and Engineering Department, University of California at San Diego, July 2012.

[71] G. Reig, J. Alonso, and J. Guitart. Prediction of job resource requirements for deadline schedulers to manage high-level slas on the cloud. In *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on*, pages 162–167. IEEE, 2010.

[72] C. Reiss, J. Wilkes, and J. L. Hellerstein. Obfuscatory obscanturism: Making workload traces of commercially-sensitive systems safe to release. In *2012 IEEE Network Operations and Management Symposium*, pages 1279–1286, April 2012.

[73] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[74] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, 2011. http://goo.gl/5uJri.

[75] Reynold XIn and Josh Rosen. Project Tungsten: Bringing Spark Closer to Bare Metal, April 2015.

[76] Reynold Xin, Parviz Deyhim, Ali Ghodsi, Xiangrui Meng, and Matei Zaharia. GraySort on Apache Spark by Databricks. Technical report.

[77] RightScale, Inc. RightScale. `https://www.rightscale.com/`.

[78] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, Cascais, Portugal, 2011. ACM.

[79] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3R: increased performance for in-memory Hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, August 2012.

[80] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[81] Stanford Large Network Dataset Collection. LiveJournal social network.

[82] David Terei and Amit A. Levy. Blade: A Data Center Garbage Collector. Technical report, 2015.

[83] James Gordon Thompson. *Efficient Analysis of Caching Systems*. PhD thesis, EECS Department, University of California, Berkeley, September 1987.

[84] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[85] Beth Trushkowsky, Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, 2011.

[86] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. *ACM SIGOPS Operating Systems Review*, 36(SI):239–254, 2002.

[87] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, and others. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[88] David Vengerov. Modeling, Analysis and Throughput Optimization of a Generational Garbage Collector. In *International Symposium on Memory Management*, Dublin, Ireland, 2009.

[89] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[90] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. Technical Report HPL-2011-58, HP Laboratories, 2011.

[91] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[92] Marc Via, Christopher Gignoux, and Esteban González Burchard. The 1000 Genomes Project: new opportunities for research and social challenges. *Genome medicine*, 2(1):1, 2010.

[93] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[94] John Wilkes. Omega: Cluster management at Google. July 2011.

[95] John Wilkes and Charles Reiss. Details of the ClusterData-2011-1 trace. 2011. https://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1.

[96] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proc. NSDI*, volume 7, pages 11–13, 2007.

[97] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. HacigümüS. ActiveSLA: A profit-oriented admission control framework for database-as-a-service providers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 15. ACM, 2011.

[98] Lijie Xu, Jie Liu, and Jun Wei. FMEM: A Fine–grained Memory Estimator for MapReduce Jobs. In *ICAC*, 2013.

[99] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX workshop on Hot Topics in Cloud Computing (HotCloud '10)*, 2010.

[100] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joesph, Randy Katz, Scott Shenker, and Ion Stoica. The Datacenter Needs an Operating System. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, page 17, Berkeley, CA, USA, 2011. USENIX Association.

[101] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In Rudolf Fleischer and Jinhui Xu, editors, *Algorithmic Aspects in Information and Management: 4th International Conference, AAIM 2008, Shanghai, China, June 23-25, 2008. Proceedings*, pages 337–348. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[102] Xiaoyun Zhu, Zhikui Wang, and Sharad Singhal. Utility-Driven Workload Management using Nested Control Design. In *Proceedings of the American Control Conference*, Minneapolis, MN, USA, June 2006.