

Reducing Garbage Collection Costs Using Application-Specific Information

A dissertation

submitted by

Diógenes Antonio Núñez

BA, Computer Science, Williams College, 2012

MS, Computer Science, Tufts University, 2015

In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

August 2020

ADVISOR: Samuel Z. Guyer

To Norman and Jeanne Merrill

Reducing Garbage Collection Costs Using Application-Specific Information

Diógenes Antonio Núñez

ADVISOR: Samuel Z. Guyer

Applications running on multiple environments are all supported by garbage collectors. Modern garbage collectors rely on two assumptions to perform well: the collector will reclaim enough memory to avoid frequent failed allocations and most memory allocated for the application soon becomes garbage. These assumptions are broken by modern applications. For instance, applications that rely on an application-level key-value store, or software cache, reduce run time by increasing memory use. However, that breaks the first assumption. When they are broken, garbage collector performance degrades. To reduce this performance loss, garbage collectors should be supplied application-specific information.

In this thesis, we support this hypothesis with three projects. First, we hypothesized Haskell programs violate the first assumption. The violation is thanks to the implementation of lazy evaluation which evaluates expressions only as needed. This can be circumvented with annotations, but the annotations can prove difficult to place correctly. We present Autobahn as a solution. Autobahn is a tool that uses a genetic algorithm to add strictness annotations to Haskell programs and improve their runtime.

Next, we hypothesized software caches violate both assumptions. The violation comes from conflicting goals. The goal of the cache is to maximize the amount of space used to improve runtime. The goal of the garbage collector is to minimize

the amount of space used to allow future allocations to succeed and keep time spent collection low. We present prioritized garbage collection as a solution. The prioritized garbage collector lets a programmer supply the garbage collector with data about objects to relieve memory-pressure and improve the robustness of software caching applications.

Finally, we hypothesized applications with long-lived data structures violate the second assumption. These data structures are visited every collection, resulting in repeated work and lost time. We present a design for a deferred garbage collector. The deferred garbage collector lets a programmer supply information about long-lived data structures to reduce garbage collection time.

Acknowledgments

I am deeply grateful and give great thanks to my advisor, Prof. Samuel Guyer. I am thankful for his advice and instruction over the last eight years of my graduate studies. His creativity and love for exploring new ideas for garbage collection motivated me to keep trying in spite of many technical setbacks. His patience and constant reassurances to put myself first helped me work through tough times and even take risks I would normally have avoided, especially during these unusual times.

I am thankful to the members of my committee, Prof. Kathleen Fisher, Prof. Alva Couch, Prof. Mark Hempstead, and Prof. Emery D. Berger. Their insightful, thorough, and honest comments and questions made this thesis and its work clearer and stronger.

I am thankful to my undergrad advisors and mentors from Williams College, Prof. Steve Freund, Prof. Jeannie Albrecht, and Prof. Andrea Danyluk. Prof. Freund started me on the path to systems research with our summer work on dynamic data race detection and I would not have even met Samuel Guyer if not for reading one of his papers on probabilistic calling contexts. Prof. Albrecht and her distributed systems course helped me realize how much I love working in the field of systems despite its frustrating headaches. Prof. Danyluk took the time one day to speak to me, some senior in high school who could not make the college visit day, about her computer science department out in Williamstown and without that, my career in computer science would not be anywhere near what it is today.

I am thankful to the members of the Autobahn team, Prof. Kathleen Fisher, Yisu Remy Wang and Marilyn Sun. Without them, a simple search would not have evolved into the powerful tool and research that it is today. I was, am, and forever will be honored to have worked with all of you.

I am thankful to my colleagues in the TuPL and Redline research groups for questioning, critiquing, and reviewing my work. They always forced me to pull back out and consider the bigger picture of my work. I am thankful to Matthew Ahrens, Jeanne-Marie Musca, Karl Cronburg, and Milod Kazerounian for our lunch breaks to remind me there is more to look forward to in the day than just overcoming the next technical hurdle. I am thankful to Nathan Ricci for teaching me how to lift weights, igniting a love for the activity and providing the best stress relief activity I have ever seen. I am also thankful to Raoul Veroy and Hu Moses Huang for being my lifting buddies for years after and being great conversation partners all around.

I am thankful to Prof. Ben Hescott, Prof. Mark Sheldon, and Bruce Molay for taking me on as a teaching assistant in the early years of graduate school. They helped me understand why I love teaching and education so much and why I pursue it now with more passion than ever before.

I thank the various organizations that sponsored my studies and projects throughout graduate school. Autobahn was supported in part by DARPA contract FA8750-15-2-0033. Prioritized garbage collection was supported by the Google Research Award. Deferred garbage collection was supported by the National Science Foundation.

Portions of this work have been previously published as [WNF16, NGB16]. I thank my co-authors for their permission to include our joint published work in this thesis.

Finally, none of this would be possible without my friends and family. They constantly supported me and reassured me when everything seemed to be at its worst. I thank my parents for constantly believing I could do this and pushing me to take the chance. I thank my brothers and sisters for indulging my conversations on niche Saturday morning shows or great choices in wall paint. I thank my friends for being there to help me up whenever I fell down and for laughing with me at all the natural ones I have rolled in our games. None of this here would have happened without any of you. Thank you.

DIÓGENES ANTONIO NÚÑEZ

TUFTS UNIVERSITY

August 2020

Contents

Abstract	iii
Acknowledgments	v
List of Tables	xii
List of Figures	xiii
Chapter 1 Introduction	1
Chapter 2 Garbage Collection Background	6
2.1 Memory Management Terminology	6
2.2 Memory Reclamation Problem	7
2.3 Reachability as Graph Search	7
2.4 Mark-sweep Garbage Collection	8
2.5 Mark-compact Garbage Collection	9
2.6 Generational Garbage Collection	10
Chapter 3 Autobahn: Genetic Algorithm for Strictness	11
3.1 Introduction	11
3.1.1 Definitions: Laziness and Strictness	12
3.2 Genetic Algorithm	14
3.3 Autobahn	15
3.3.1 Genes and Chromosomes	15
3.3.2 Fitness Functions	15

3.3.3	First Generation	17
3.3.4	Producing Future Populations	18
3.3.5	Termination	19
3.3.6	Summary	19
3.4	Experiments	19
3.4.1	Experimental Setup	20
3.4.2	Performance	20
3.4.3	Strictness	23
3.4.4	Case Study: <code>gcSimulator</code>	26
3.4.5	Case Study: <code>Aeson</code>	29
3.4.6	Ten-fold Cross-validation	30
3.4.7	Autobahn Performance	32
3.5	Related Work	33
3.5.1	Static Analysis	33
3.5.2	Dynamic Information	34
3.5.3	Other Approaches	35
3.6	Conclusion	35
Chapter 4 Prioritized Garbage Collection		37
4.1	Introduction	37
4.2	Problem	41
4.2.1	Existing cache implementations	41
4.2.2	Exploring cache-GC interaction	44
4.2.3	Guava performance	46
4.3	Prioritized Garbage Collection	49
4.3.1	API	49
4.3.2	Measuring Memory Footprint	51
4.3.3	Collection Algorithm	55
4.3.4	Overheads	58
4.3.5	Sache: A Space-aware Cache	58

4.3.6	Adaptive Sizing	60
4.4	Results	61
4.4.1	Experiments	61
4.4.2	Methodology	62
4.4.3	Non-adaptive SACHE	62
4.4.4	Multiple Caches	66
4.4.5	Adaptive SACHE	68
4.4.6	Web Caching Workload	71
4.5	Related Work	72
4.5.1	Reference Types	72
4.5.2	Programs Acting on Resource Limits	72
4.5.3	Using GC to Assist Running Programs	73
4.6	Conclusions	73
Chapter 5 Deferred Garbage Collection		74
5.1	Introduction	74
5.2	Proposing a Deferred Collector	76
5.2.1	DeferRef	76
5.2.2	Deferred Garbage Collection Algorithm	76
5.2.3	Implementation Challenges	77
5.3	Collecting the Benefits and Costs	79
5.3.1	Collecting Data at the Object Level	79
5.3.2	Measuring Savings	79
5.3.3	Retention	80
5.3.4	Disabling Deferral	80
5.3.5	Instrumenting a Collector	81
5.4	Experiments	82
5.4.1	Experiment	82
5.4.2	Methodology	82
5.4.3	Savings on Marking and Compacting	82

5.4.4	Cost of Retention	83
5.4.5	Benefit of Ignoring Deferral	85
5.5	Related Work	85
5.5.1	Automatic Pretenuring	85
5.5.2	Heap Subdivision	86
5.5.3	Programmer Hints	86
5.5.4	Connectivity	87
5.6	Conclusion	87
Chapter 6 Conclusions and Future Work		88
6.0.1	Autobahn	88
6.0.2	Prioritized Garbage Collection	89
6.0.3	Deferred Garbage Collection	89
Bibliography		91

List of Tables

3.1	Peak memory allocation, total run time, and garbage collection time for hand and Autobahn-optimized version of <code>gcSimulator</code> , normalized to the bare program Each colored band has two rows: the top row is the hand-optimized version, the bottom row the Autobahn-optimized version.	27
3.2	Peak memory allocation, total runtime, and GC time for Autobahn-optimized version of two Aeson driver programs, normalized to the bare program. For each band, the first row shows the results for <code>validate</code> and the second for <code>convert</code>	29

List of Figures

3.1	Comparing Autobahn-optimized programs to original programs. a) Optimized for runtime. b) Optimized for live size. c) Optimized for garbage collection time.	21
3.2	Heap profiles for original (top) and Autobahn-optimized <code>lcss</code> (bottom) program. The optimized program benefits from reduced memory usage at the call to <code>algb.algb1.algb2</code> , seen in yellow in the original and red in the Autobahn-optimized.	24
3.3	Heap profiles for original and Autobahn-optimized <code>fulsom</code> program respectively. The optimized program benefits from reduced memory usage across all cost centers.	25
3.4	Comparing NoFib benchmarks run under Strict Haskell to those optimized by Autobahn for run time.	26
3.5	Heap profiles of <code>gcSimulator</code> on $\frac{1}{2}$ of the <code>batik</code> trace.	28
3.6	Ten-fold evaluation for <code>gcSimulator</code> , showing runtime and live size performance improvements of Autobahn-optimized versions of <code>gcSimulator</code> compared to the bare program. We highlight points where the Autobahn-optimized program ran on its training trace. We also show how the hand-annotated program performed.	31
3.7	Ten-fold evaluation for <code>convert</code> , showing runtime and live size performance improvements of Autobahn-optimized versions of <code>convert</code> compared to the bare program. We highlight points where the Autobahn-optimized program ran on its training trace.	32

3.8	Graph demonstrating the run time of Autobahn itself when optimizing NoFib benchmarks for runtime and garbage collection time. . . .	33
4.1	Competing tradeoffs: as cache size increases, miss time goes down, but GC time goes up.	38
4.2	Guava cache that stores graphs and uses a weighing function to represent their size.	42
4.3	Example trace file. The number associated with each key determines the size of the data structure that is stored as the value.	45
4.4	Guava performance under three workloads: choosing a good number of entries is difficult.	47
4.5	Memory allocated under the workload of medium-sized values: Undersizing a cache (left side) incurs the cost of more misses as well as the cost of increased allocation.	48
4.6	A priority reference holds a single referent with a given priority. The application can also inquire about the total amount of memory reachable through this reference.	50
4.7	A priority space holds a set of priority references and governs their lifetime collectively under single policy.	51
4.8	Interface for Sache space-aware cache	59
4.9	Sache performance under three different workloads. While absolute performance varies, compared to the Guava cache the space-time tradeoff is relatively independent of the workload. The Sache makes the best use of the available space.	63
4.10	Comparing Sache and Guava LRU on three workloads: the performance is very similar, but the three Sache curves represent the same configuration choices. The highlighted points represent a Sache set to 40% of the heap, which easily accommodates all three workloads by using different numbers of entries.	65

4.11	Hit rates can drop dramatically when soft references are used for two caches working at different frequencies. Prioritized garbage collection keeps the hit rates of both caches relatively close by managing their resources separately.	67
4.12	Performance of Sacle vs Guava LRU cache under increasing memory pressure: our adaptive sizing algorithm shrinks the Sacle to avoid triggering massive GC overhead. At 77MB, the application with the Guava LRU cache crashes.	69
4.13	GC time over a single run: without the ability to adapt, cache and non-cache structure compete, triggering costly GCs.	70
4.14	Performance of the Sacle and Guava cache on real web traffic traces across a range of cache sizes.	71
5.1	API for the DeferRef type.	76
5.2	Ratio of time spent marking and compacting deferred objects under different cache sizes. As the cache size increases, so does the number of objects we defer on. This results in a longer compaction times for deferred objects.	83
5.3	Amount of memory retained from deferral under different cache sizes.	84
5.4	Amount of memory retained from deferral, varying the number of deferred GCs.	84

Chapter 1

Introduction

Modern programming languages come with some form of a *runtime system*. A runtime system for a programming language is a program that supports an application written in that language. The support varies between languages. However, most runtime systems include a memory manager to support applications by managing their heap memory, shortened to the *heap*.

Modern applications rely on the heap over their lifetimes to store data needed by those applications. When applications need this storage to continue, the application stops and sends a request to allocate memory on the heap to the *allocator*. The allocator is the part of the runtime system that modifies the heap through allocation and pointer modification. When the allocator fulfills the request, the application receives a pointer to an *object* to access and resumes. Over time, the application will no longer need some data and stop using that data and object altogether. Even though the object is no longer used, it remains allocated on the heap, becoming *garbage*.

This garbage must be deallocated for the allocator to reuse heap memory and complete future allocation requests¹. In some languages, garbage is explicitly deallocated. However, manual deallocation can lead to subtle memory bugs like use after free. These bugs result in application crashes or incorrect results. To avoid these bugs, modern languages with a runtime system also have a *garbage collector*

¹In the memory management field, deallocating memory is also referred to as *freeing* memory

automatically deallocate garbage.

A garbage collector is the part of the runtime system that automatically deallocates garbage for an application. By automatically deallocating this memory, an application no longer suffers from bugs related to incorrect memory reclamation. As a result, garbage collectors support applications written in many popular high-level programming languages. Garbage collectors support mobile applications written in Java and Swift, web applications written in JavaScript, as well as long running server and data-processing programs written in Java.

With the widespread use of garbage collectors, the performance of those garbage collectors is important. Garbage collectors are run when the runtime system detects memory for allocation is about to run out. Garbage collectors then must find the garbage on the heap and deallocate it. This is called a *collection*. If the allocator fails an allocation request before collection ends, the allocator cannot allocate on the heap and therefore the application cannot do any work. Therefore, the time spent on a collection affects the run time of the application. The slower the garbage collector runs, the longer the collection takes, and the slower the overall application runs. In practice, garbage collectors perform very well. This is the result of many innovations in implementation and design.

These new designs make certain assumptions about applications. One assumption is most memory in the application becomes garbage soon after it is allocated. As a result, a garbage collector can limit the amount of memory to search. This results in fast searches each time the garbage collector runs. Another assumption is the garbage collector will reclaim enough memory to avoid frequently failed allocations. Since the garbage collector runs when the application fails to allocate memory on the heap. The memory reclaimed by the garbage collector is used for future allocations. Future allocations are more likely to succeed, resulting in infrequent allocation failures and therefore infrequent collections. The assumptions are followed by most applications, resulting in good collector performance.

These assumptions can be broken by modern applications. One such application is the *software cache*. A software cache, abbreviated to cache in this document,

is a key-value store in an application that stores a fixed number of intermediate results. Caches trade the space of storing these results for the time spent creating those results. Ideally, the more space a cache can use to store more results, the more time is saved by not recreating those results. This trade-off violates a previously stated assumption: the garbage collector reclaims enough memory to avoid frequent failed allocations.

By breaking that assumption, caches exhibit bad performance when they use more space. These large caches leave less memory available for future allocations. The result is more frequent collections. Unfortunately, each collection also takes longer to complete. This increase in run time cost is possible in most modern garbage collectors. The performance of modern collectors is proportional to the amount of memory in use by the application [HB05]. As caches store more results, the amount of memory in use increases, leading to costly collections. In general, breaking the assumptions made by garbage collectors results in bad performance.

When an application must break the garbage collector's assumptions, programmers need a solution to avoid the performance loss. Common solutions include increasing available heap memory for the application or forgoing garbage collectors all together. Increasing heap memory increases the amount of total memory available for allocations and reduces the frequency of collections. However, each collection still scales with the amount of objects in use by the application. If these objects account for most of the memory in the new heap, each collection remains costly. Removing garbage collectors all together removes the cost of garbage collectors. However, memory must now be explicitly reclaimed, reintroducing subtle memory bugs. A solution should retain the benefits of garbage collection while further minimizing the costs when the assumptions fail.

We propose a solution where the garbage collector is informed by the programmer. Programmers often have information that can help the garbage collector. In the cache example, programmers supply caches with an *eviction policy*. An eviction policy determines what intermediate results should be removed when the cache is full. We say the result was *evicted* from the cache. Ideally, this evicted result is

not in use by the application. In that case, the result’s memory becomes garbage and will remain on the heap until a future collection deallocates that memory. If the garbage collector were aware of this eviction policy, the collector could deallocate the memory of an evicted result the moment it was evicted. For this reason, we hypothesize this information should be supplied the garbage collector through the runtime system by the programmer.

Any solution that supplies information to the garbage collector needs to fulfill certain criteria. One criterion is the solution must give the programmer a sufficiently expressive way to supply the information. Another criterion is the solution must tolerate imperfect information from the programmer. We will discuss each criterion separately.

Expressiveness dictates the kind of information the programmer can give to the runtime system. For instance, a common solution to supplying information is adding new annotations to the language [BOF17, NWB⁺15, NFX⁺16, Ban16, Ora15]. Annotations allow programmers to alert runtime systems about the existence of certain objects. However, they are insufficient for eviction policies in caches. Eviction policies not only need to know which objects to compare, but also the value of each object to the current application. Without that value information, the policy cannot report the least valuable object to evict from the cache.

The system must tolerate imperfect information. Programmers cannot computationally predict all possible behaviors for their application, as shown by the halting problem. One example is the eviction policy of a cache. Depending on the policy, some results in the cache might be evicted even though they are still in use by the application. If the memory storing these results is deallocated right when evicted, a memory bug can occur. This bug can subtly introduce undesired behavior into our application, like incorrect results or crashing. This undesired behavior must be prevented in the system.

Solutions that adhere to the proposed criteria improve performance when garbage collection assumptions are broken. In this thesis, we explore three problems where applications violate an assumption garbage collection relies upon. We then

present and evaluate new solutions to these problems while adhering to our proposed criteria. We present the following contributions:

First we present Autobahn. Autobahn is a tool that uses a genetic algorithm to automatically add strictness annotations to Haskell programs at the source level [WNF16]. The resulting annotations reduce the amount of memory in use and improve performance even on highly optimized programs. I am responsible for the initial exploration and proof of concept. Together with Remy Wang and Kathleen Fisher, we improved the search’s performance into the system presented in this document and explored different use cases for Autobahn.

Next we present the prioritized garbage collector. The prioritized garbage collector lets programmers to tell the garbage collector the importance of annotated objects to their application [NGB16]. The garbage collector uses this information to decide when to deallocate these objects to relieve memory-pressure.

Finally, we present the deferred garbage collector. The deferred garbage collector lets programmers tell the garbage collector what data to *not* look at during collection through annotations. These annotations are fed as hints to the collector to avoid repetitive work, saving time during each collection.

What follows is the outline of this thesis. In Section 2, we discuss relevant technical background used in the rest of the dissertation. In Chapter 3, we discuss Autobahn. In Chapter 4, we discuss the prioritized garbage collector and how it assists in applications dependent on key-value stores. In Chapter 5, we discuss the deferred garbage collector. Finally in Chapter 6, we summarize the key findings followed by possible directions of future work.

Chapter 2

Garbage Collection Background

Each chapter introduces a system that impacts memory use. Later chapter consider new algorithms for garbage collection. To fully understand the impact of each chapter, we must first explore garbage collectors in general and the problem they solve.

In this chapter, we explore garbage collection at a high level. We start by describing the problem garbage collectors solve. We then explore how garbage collection find memory to reclaim. Afterwards, we explore three modern garbage collector algorithms referenced in this thesis: the mark-sweep garbage collector, the mark-compact garbage collector, and the generational garbage collector.

2.1 Memory Management Terminology

This section briefly touches on some common terminology in the memory management field: the abbreviation GC and object headers.

In the memory management field, both a collection and the garbage collector are shortened to GC. When GC is used in text or speech, the context determines which definition to use.

An *object header* is the metadata stored on an object usually preceding its application-relevant data. The metadata is read and modified by the runtime system, including the garbage collector.

2.2 Memory Reclamation Problem

The key to understanding garbage collection is understanding the lifetime of an object. The life of an object starts when it is allocated. At this point, the object is referred to as *live*. The object remains live until it is used for the last time. The object is now referred to as *dead*. The dead object takes up memory on the heap better suited for allocating new objects. Ideally, the memory is reclaimed after its last use by a garbage collector.

Unfortunately, it is impossible for a collector to know when an object has seen its last use. Instead, garbage collectors must wait until the object is no longer *reachable*. An object o is *reachable* if

1. o is directly accessible to the application, like a local or global variable. These objects are called *roots*.
2. o is pointed to by any other reachable object

Otherwise, o is *unreachable*. This concept is used by modern garbage collectors to find dead objects.

Reachability is an approximation of the liveness of an object. If an object o is live, then o is in use by the application. It follows that any object reachable from o may also be used and therefore might also be live. In actuality, some of those reachable objects might be dead. However, only dead objects are unreachable. If the object is unreachable, then an application cannot access the object directly through the roots or indirectly through a series of pointer dereferences. This implies the object has seen its last use, is dead, and safe to reclaim. Garbage collectors consider an object dead if it is unreachable.

2.3 Reachability as Graph Search

We cast the problem of finding unreachable objects as a directed graph search problem. We consider first the representation of the heap as graph. Then we consider the new search problem.

The graph is a common representation of the heap. In this representation, objects in the heap become vertices in the graph and pointers between objects in the heap become edges between vertices in the graph. The strength of the representation is that connectivity in the heap directly translates to connectivity in the graph. This correspondence allows us to consider finding unreachable objects as a graph search problem.

The set of source vertices for the search are the objects immediately reachable from the roots. Roots are always accessible by the application, so they are always live. Furthermore, live objects are always reachable from other live objects. Live objects have corresponding vertices in the graph. It follows that all clusters of live objects on the heap will be connected components on the graph. It also follows that those connected components cannot contain vertices corresponding to unreachable objects. Therefore, to find the unreachable objects, the garbage collector needs only identify the reachable objects.

2.4 Mark-sweep Garbage Collection

Mark-sweep garbage collectors work as follows:

Phase (1): The collector enumerates all the roots in the application. These roots are the sources of the graph search.

Phase (2): The collector starts the *mark phase*. The collector performs a complete depth-first search starting from the roots, marking unvisited objects during the search. This is called the *transitive closure* from the roots.

Phase (3): Finally, the collector starts the *sweep phase*. The sweeper deallocates all unmarked objects on the heap.

Once the collector finishes, control is returned to the application.

From this breakdown, it follows the performance of a mark-sweep collector is directly proportional to the number of live objects on the heap. The more live

objects on the heap, the longer the mark phase runs, and the longer the garbage collector runs.

2.5 Mark-compact Garbage Collection

One variant of the mark-sweep garbage collector is the *mark-compact* collector. Instead of relying on a sweeper to deallocate unmarked objects, the mark-compact collector moves live objects such that all live objects are contiguous.

Mark-compact garbage collectors work as follows:

Phase (1): The collector enumerates all the roots in the application. These roots are the sources of the graph search.

Phase (2): The collector starts the *mark phase*, just like in the mark-sweep collector.

Phase (3): The collector starts the *forwarding phase*. The collector calculates where live objects will be moved to at the end of collection. This process is called *forwarding*. The destination of the object is called the *forwarding address*. This address is usually stored as metadata in the object's header.

Phase (4): The collector starts adjusting the outgoing pointers of all live objects. All live objects will be moved to their forwarding address by the collector. Prior to the move, the collector must replace all outgoing pointers of live objects with the forwarding addresses.

Phase (5): Finally, the collector starts the *compact phase*: The collector moves objects to their forwarding addresses. This process is referred to as *compaction*. At the end of this phase, the heap is neatly segregated into two contiguous ranges, the range for live objects and the range for future allocations.

2.6 Generational Garbage Collection

In practice, most objects die soon after they are allocated [LH83]. This observation is exploited by *generational* garbage collectors to improve performance [Ung84].

A garbage collector is generational if the collector segregates objects based on how many collections they survive. The objects are segregated into a *young generation* and an *old generation*. New objects are allocated into the young generation.

By segregating the heap, a generational garbage collector can choose to collect from part of the heap. When only the young generation is full, the garbage collector collects only from the young generation. Such a collection is called a *minor collection*. When both the young and old generations are full, the garbage collector collects from both generations. Such a collection is called a *major collection*. We will explore each type of collection separately.

Minor collections deallocate garbage from the young generation. Objects in the young generation may only be reachable through incoming pointers from objects in the old generation. These pointers are collected in a structure called a *remembered set*. The remembered set is then considered a root for the minor collection. After a minor collection, objects in the young generation that have survived either remain in the young generation or are moved to the old generation. The latter is called *tenuring*¹. In a minor collection, tenuring occurs only if the object has survived some number of young collections.

Major collections deallocate garbage present in both generations. In modern collectors, major collections run the mark-sweep garbage collection algorithm. At the end of a major collection, all objects in the young generation are tenured into the old generation.

¹This process is also referred to as *promotion*.

Chapter 3

Autobhan: Genetic Algorithm for Strictness

3.1 Introduction

Lazy evaluation in Haskell allows for compact and modular high level code by only evaluating expressions when needed[Hug89]. When the expression is not needed in Haskell, the runtime system allocates a *thunk* on the heap for it. A thunk is a heap-allocated unit that contains all the information necessary to evaluate a suspended expression. The number of thunks increases as the program becomes lazier. As a result, excess laziness leads to an excess of thunks cluttering the heap. This clutter results in extra garbage collections and longer runtimes.

To combat excess laziness, Haskell provides *strictness annotations* [Ban16]. Strictness annotations let programmers label expressions which will be evaluated earlier than their value is needed. However, excess annotations can change the semantics of a program or slow the program down further. These semantic changes caused by these annotations are difficult to reason about in large programs.

To avoid reasoning about the behavior, static analyses were created to place strictness annotations automatically[CPJ85, HH10]. Unfortunately, these analyses must be conservative to preserve the semantics of the original program on all possible

inputs. However, there might be an annotated version of the program that might not terminate on all possible inputs, but terminates on a subset of inputs and is faster on those inputs. In contrast, a dynamic analysis can find expressions to annotate that a static analysis will miss. Therefore, we consider a dynamic analysis on a set of inputs that represent the inputs the program is expected to run under.

3.1.1 Definitions: Laziness and Strictness

Lazy evaluation in Haskell is implemented by wrapping expressions into thunks. Thunks are *forced*, or fully evaluated, when the computation's value is needed by the program. Consider the following Haskell code, which defines the function `take`.

```
take 0 xs = []
take n (x:xs) = x:(take (n-1) xs)
```

`take` removes the first `n` items from the list parameter `xs`. Now suppose we wish to evaluate the expression `take 0 [1..]`, where `[1..]` is an infinite list. Under lazy evaluation, each argument of `take 0 [1..]` is only evaluated as needed. In this case, `take` requires the number to evaluate which case of `take` to run. The list is left unevaluated as a thunk. We will denote an expression `e` in a thunk as `[|e|]`. As a result, we get the following execution trace

```
take 0 [1..] -> take 0 [| [1..] |] -> []
```

In the base case of `take`, the function definition does not need to read into the list. Therefore, Haskell will not evaluate the thunk containing the list, saving the computation time and space needed to contain the final value of that list.

Haskell lets programmers add strictness annotations to expressions. When an expression is annotated, the expression is *strictly* evaluated, or reduced until a data constructor must be evaluated. Such a value is said to be in *weak head normal form*. For instance, `cons 1 []` and `cons 1 [| 2... |]` are both in weak head normal form. In both cases, the `Cons` data constructor must be evaluated to further reduce the expression.

To show the difference in evaluation, consider our previous `take` function and our expression `take 0 [1..]`. Recall in lazy evaluation, the list argument is never

evaluated and is left in a thunk. In strict evaluation, the list argument is evaluated to the cons cell `Cons 1 [] [2..] []`. Note that the tail of the list is a thunk and not the expansion of the list.

Note that in our example, Haskell avoids evaluating the entire infinite list, preventing non-termination. In general, strict evaluation can still lead to unintended behavior, slowdown, or non-termination. Consider the following function,

```
f x y !z = if x then y else z
```

In this example, the argument `z` has a strictness annotation and so `z` will be strictly evaluated upon entering the function. As a result, `z` will always be evaluated regardless of the value of `x`. If the evaluation of `z` does not terminate or is costly in terms of time and space, then `z` would be better left as a thunk.

INTRO TO THIS PART NEEDED Consider the following implementation of the factorial function:

```
fact 0 = 1
fact n = n * fact (n - 1)

let y = fact u
```

Assume `u` is read from user input. Then say we add a strictness annotation to `y`. If `u` is negative, then factorial will not terminate.

However, if the only possible values of `u` are non-negative, then it is safe to add that annotation. Therefore, we consider a dynamic analysis on a set of inputs that represent the inputs the program is expected to run under.

A dynamic analysis can find opportunities for strictness a static analysis will miss. Since a static analysis must be conservative, `n` in `fact` will not be annotated. However, if the only possible values of `u` are non-negative, then it is safe to add that annotation. Therefore, we consider a dynamic analysis on a set of inputs that represent the inputs the program is expected to run under.

This chapter presents Autobahn, a tool that uses genetic algorithm to infer strictness annotations. Using user-provided inputs for programs, Autobahn searches the space of annotated programs to find a program with better performance. Sec-

tion 3.2 presents an overview on genetic algorithms. Section 3.3 presents Autobahn and the genetic algorithm underneath. Section 3.4 presents the results of our evaluation of Autobahn.

3.2 Genetic Algorithm

We explain the general search problem. Consider a function \mathcal{F} that takes some argument vector \vec{x} and returns some value. The goal of the search is to find some vector \vec{x} that returns the maximum value when given to \mathcal{F} . That vector can be found by searching the space of possible argument vectors. However, that space can be too large to search exhaustively. Therefore, we must turn to heuristic search methods. One such search method is a genetic algorithm.

A genetic algorithm uses the ideas of evolution to search a space [Gol89]. The search finds argument vectors \vec{x} , called *chromosomes*. Each value in the vector is called a *gene*. The function to optimize, \mathcal{F} is called the *fitness function* as it measures how fit the chromosome is. The goal of the algorithm is to find a chromosome that returns the maximum value when given to \mathcal{F} . The algorithm searches the space over a number of rounds, or *generations*. Each generation starts with a set of chromosomes called a *population*. A genetic algorithm performs the following steps each generation:

Step (1): The algorithm measures each chromosome in the population with the fitness function. The function produces a fitness score for each chromosome.

Step (2): The algorithm then *mutates* a random selection of the population. Mutation randomly edits the genes of one chromosome to produce a new chromosome.

Step (3): The algorithm then performs splices together random pairs of the population. This action is called the *crossover*.

Step (4): The algorithm creates a new population. The new population is comprised of the chromosomes made from both mutation and crossover as well as the

best scoring chromosomes from the current population.

Step (5): Finally, the algorithm starts the next generation with the new population.

3.3 Autobahn

3.3.1 Genes and Chromosomes

In Autobahn, a gene is a program location that can be annotated. If the location has been annotated, the gene is *on*. Otherwise, the gene is *off*. Naturally, a gene is represented by a bit, where the bit is set if and only if the gene is on. For example, consider a program with only the following function

```
foo !x y z = x + y - z
```

This program has 3 genes, one for each parameter. The first parameter has a strictness annotation, so its gene is on.

In Autobahn, a chromosome is some configuration of strictness annotations in a program. A chromosome is represented by a vector of bits. For example, the above program's chromosome is 100. To find the genes and construct chromosomes for a program, Autobahn parses the source code with `haskell-src-extends` [Bro15], a parser library for Haskell programs, to find all possible locations that can be annotated.

3.3.2 Fitness Functions

To measure the fitness of a chromosome, Autobahn must compile and run many versions of the same program. When Autobahn starts, it parses the original program, producing an abstract syntax tree, or AST. Given a chromosome to measure, Autobahn modifies this AST with respect to that chromosome and the genes therein. The modified AST is then pretty-printed back to disk using `haskell-src-extends`. Finally, the file is compiled by `ghc` and run with the `ghc` profiler and a user provided input.

Autobahn gives the user three functions to choose from to measure the fitness:

runtime, garbage collection time, and peak memory use. Each function produces a *score* for a given chromosome. The runtime function returns the overall runtime of the application as the score. When used, Autobahn prioritizes finding overall faster programs. The garbage collection time function returns the amount of time spent in garbage collection time as the score. When used, Autobahn prioritizes programs that spend less time in garbage collections. Less time spent in garbage collection means The peak memory fitness function returns the maximum number of live objects on the heap, measured in bytes, as the score. This function prioritizes programs that reduce their peak memory overall. Scores are produced by running the modified program and parsing the output given by the `ghc` profiler under the `+RTS -t` option [ghc15]. This option is used to prune locations that contribute less than 1% from the profile.

To reduce the amount of time Autobahn spends measuring, Autobahn wraps each run with a time out. If the modified program times out, the modified program is given a low score. By default, this time limit is twice the original program's runtime, which we obtain by running the original program on the given inputs at the very start. This time out is chosen as a balance between reducing the search time and exploring the space of annotations.

To properly explore the space of annotations, programs that measure slightly worse than the original program remain in the population. Furthermore, these programs are valid candidates for creating the population for the next generation. This is counter-intuitive, but these programs might have annotations that can trigger performance only when combined with an as-of-yet introduced annotation.

When Autobahn introduces an annotation into the program, there is a risk that the modified program may not terminate or even compile. We will discuss each risk separately.

Non-termination Conceptually, a non-terminating program is conceptually no different than a program that runs slower than the original. This program will trigger the time out and its corresponding chromosome is given a low score to prevent

it from influencing the next generation.

Fail to compile `haskell-src-exts` allows annotations in two locations that trigger compile-time errors with `ghc`. The first location is a recursively defined variable. Consider this example found in the NoFib benchmark [Par93] which creates either an empty list or a finite list of copies.

```
copy (!n) x = take (max 0 n) xs
  where !(xs) = x : xs
```

The offending line is the definition of `xs`. This definition creates an infinite list of values with a semantically valid Haskell expression. When the annotation is introduced into the idiom as shown, `ghc` raises a parse error, citing the recursive use of `xs`. The second location is variables that appear under a typeclass instance declaration. Consider the following code snippet, an example of such a declaration.

```
instance Monad Baz where
  !m1 >>= f = ...
```

In this code, the inline operator `bind` (`>>=` in the example) is defined, which takes `m1` and `f` as inputs. `m1` is annotated in the definition. This annotation causes `ghc` to raise a parse error on `bind` itself. In both cases, `ghc` returns an error code which is caught by Autobahn and is given a low score.

3.3.3 First Generation

To start the genetic algorithm, an initial population is created. Autobahn starts by reading user-specified set of files from the user program. Those files are converted into chromosomes. The genes in the chromosome are randomly chosen and flipped to create a new chromosomes. This process is repeated to create a set of random chromosomes and together with the original chromosome make up the initial population. Note that this process can introduce new annotations or remove old annotations from the program.

3.3.4 Producing Future Populations

Autobahn passes the population with its fitness scores to the GA library [Hos11] to create the population for the next generation. To create that generation, the library also uses mutation and crossover function defined for Autobahn. Each function is explained separately. **Mutation** The GA library calls our mutation function a number of times as determined by the user. Each time the function is called, a random chromosome c is chosen. Conceptually, random genes in c are flipped from on to off or vice-versa. In our mutation function, each gene in a chromosome is flipped with some probability p . In detail, Autobahn first generates a list of random floats. The length of the list is equal to the length of the chromosome. The list of random floats is then transformed into a vector of bits of equal length where each bit is set if the corresponding float is smaller than p . The two vectors are combined with an xor, creating a new chromosome for the next generation.

Crossover The GA library calls our crossover function a number of times determined by the user. Each time the function is called, two random chromosomes $c1$, $c2$ are chosen. Our crossover function implements the Uniform Distribution [Sys89] strategy to ensure each gene in both chromosomes has an equal chance to be a part of the new chromosome. Intuitively, half of the genes from $c1$ are chosen for the new chromosome. The remaining genes in the new chromosome are taken from $c2$. In detail, the function first creates a list of random floats such that statistically about half of them are less than 0.5. This list is transformed into a vector of bits v of equal length to the length of the chromosomes. Each bit of the vector is set if the corresponding float is less than 0.5. The set bits in v will select the genes from $c1$ that will pass onto the new chromosome. Similarly, the unset bits in v will select the genes from $c2$ that will be passed onto the new chromosome. To create the new chromosome, $c1$ and v are combined with a bitwise-and, creating $c1'$. Similarly, $c2$ and the complement of v are combined with a bitwise-and, creating $c2'$. Finally, $c1'$ and $c2'$ are combined with a bitwise-or to create the new chromosome.

3.3.5 Termination

Autobahn runs for a user-specified number of rounds. At the end of the last round, Autobahn produces the survivors from the final generation with their respective fitnesses.

3.3.6 Summary

A user starts Autobahn by specifying a set of files to annotate and a fixed number of generations to run. Autobahn parses the files and runs the program to determine a time out threshold. Autobahn starts the search proper by creating an initial population using the original file as a seed. This population is then measured, producing a set of scores. The population and scores together are passed to the GA library for mutation and crossover. The results from the library and the fittest chromosomes become the new population and the next generation begins. Once Autobahn reaches the allocated number of generations, Autobahn outputs the scores of each chromosome in the final population.

3.4 Experiments

In this section, we evaluate Autobahn in the following ways:

- We run Autobahn on 60 programs from the NoFib benchmark suite [Par93] and compare the performance of Autobahn-optimized programs to the performance of the original programs. By doing so, we demonstrate Autobahn finds annotations that improve performance.
- We compare Autobahn-optimized programs to the performance of the original programs run under Strict Haskell [str15], a version of Haskell that is strict by default. By doing so, we demonstrate Autobahn is necessary to find performance enhancing annotations.
- We optimize a garbage collection simulator, which inspired this project, as a case study. By doing so, we demonstrate how Autobahn can be used to

optimize a real world program.

- We compare the annotations Autobahn finds for the Aeson library [Bry16] when the library is used by different driver programs. By doing so, we demonstrate Autobahn will find annotations that improve performance with respect to how the program is used.
- We measure the reliability of Autobahn’s results by performing a ten-fold cross validation. By doing so, we show Autobahn will reliably find performance-enhancing annotations.
- We explore the run time Autobahn when optimizing NoFib benchmark programs and our two use cases. By doing so, we show Autobahn is best used during development downtime.

3.4.1 Experimental Setup

All programs were compiled and run on a computer with four 16-core AMD Opteron 6380 processors clocked at 2.5 GHz and 128 GB of RAM. Each program was compiled with `ghc 7.10.3` with `-O2`, `-XBangPatterns`. For computing live size information, we used `-RTS -h -i0.01` to perform frequent garbage collections. For Strict Haskell, we had to compile the programs with `ghc 8.0.1` with `-XStrict` as the pragma is not available in 7.10.3.

3.4.2 Performance

We want to know how well Autobahn finds annotations with respect to our different fitness functions. We ran Autobahn on 60 programs from the NoFib benchmark suite [Par93]. We chose these 60 since `haskell-src-exts` was able to successfully parse them. We optimized each program three times, once for runtime, once for live size, and once for garbage collection time. We report geometric means for our results, following NoFib’s conventions.

Figure 3.1 shows the results of Autobahn-optimized programs normalized to the original program. The x-axis lists the benchmark program in order of increas-

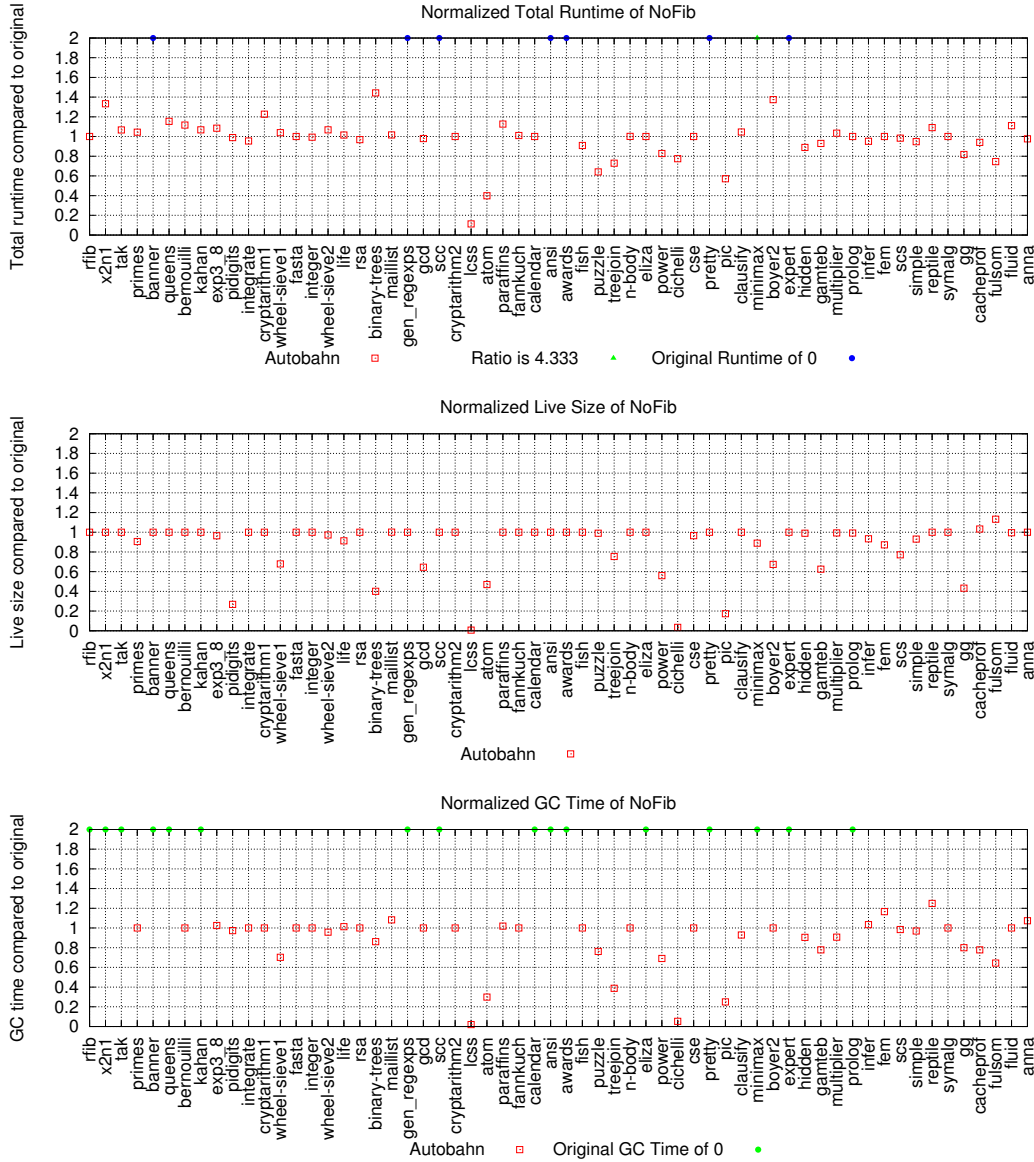


Figure 3.1: Comparing Autobahn-optimized programs to original programs. a) Optimized for runtime. b) Optimized for live size. c) Optimized for garbage collection time.

ing number of genes. The y-axis lists the normalized metric, reporting the ratio of Autobahn-optimized metric to the original program's metric. In these graphs, a lower value means the Autobahn-optimized program performed better than the original. Since Autobahn returns the original program if it cannot find a better version, we expect all values to be less than or equal to 1.0. A manual review revealed these degradations were caused by noise and are exaggerated due to how fast the programs themselves run. One of these programs, `minimax`, has a ratio of 4.333, but manually we found the difference was around 100 milliseconds. Finally, circles in the graph represent programs where the original version ran so close to zero time no measurable improvement was possible.

With each fitness function, Autobahn decreased runtime by 8.5%, live size by 7.2%, and garbage collection time by 18%. `lcss` saw the most improvement in all metrics with deltas of 89%, 99.3%, and 98%. Comments in the `lcss` benchmark state there are many opportunities for optimization. These results support the comments.

To explain the performance improvements using the run time metric, we looked at the heap profiles of twelve NoFib programs that Autobahn improved the most. We then compared those profiles with the heap profiles of their original versions. All optimized programs we reviewed had reduced heap usage. Figure 3.2 and Figure 3.3 shows the heap profiles for `lcss` and `fulsom` as two examples. The x-axis represents a specific time when `ghc` profiled the program. The y-axis details the amount of memory in 1k byte increments. Each band of color is a cost center, an expression annotated by `ghc` automatically to observe the time and space cost of evaluating that expression.

The Autobahn-optimized `lcss` benefits from reducing memory allocated at one cost center. In the original version of `lcss`, the majority of memory is allocated from one cost center, the call `algb.algb1.algb2`. In the Autobahn-optimized version, that cost center is significantly reduced and no longer dominates the heap profile. Otherwise, the shape of the profile is largely unchanged.

In contrast to `lcss`, the Autobahn-optimized version of `fulsom` has a differ-

ent shape compared to its original. The original version of `fulsom` demonstrated two peaks in its profile. In the Autobahn-optimized version, `fulsom` not only reduced its overall memory usage, but managed to only peak once.

3.4.3 Strictness

As of version 8.0.1, `ghc` has two additional language pragmas, `-XStrictData` and `-XStrict`. When compiled with `-XStrictData`, datatypes become strict by default. When compiled with `-XStrict`, functions, datatypes, and bindings become strict by default.

Do the programs perform just as well if every point of the program is strictly evaluated? To answer this question, we compiled and ran the same 60 benchmarks using Strict Haskell [str15]. Specifically, they were compiled with `ghc 8.0.1` with `-O2`, `-XBangPatterns`, `-funbox-strict-fields`, `-XStrict`, and `-XStrictData` flags. Figure 3.4 shows the results.

Seventeen programs failed under Strict Haskell. These programs failed for one of the following reasons:

- The program uses an infinite list. For example, `wheel-sieve1` and `wheel-sieve2` specify an infinite list of primes but demand only the first few. With Strict Haskell, the programs try to evaluate the infinite lists.
- The program depends on a lazy evaluation of `error` to detect a specific problem. For example, `infer` puts `error` at the end of a list; reaching this value signals an error. With Strict Haskell, the error is always triggered.
- The program contains a latent dynamic error. For example, `reptile` crashes when `nil` is passed to the `tiletrans` function, which does not occur when the program is evaluated lazily, but does occur when using Strict Haskell.

Nine programs performed worse, some significantly so, because Strict Haskell forces the evaluation of expressions that aren't needed. Autobahn did better than Strict Haskell on all of these programs. Of the programs that improved under Strict

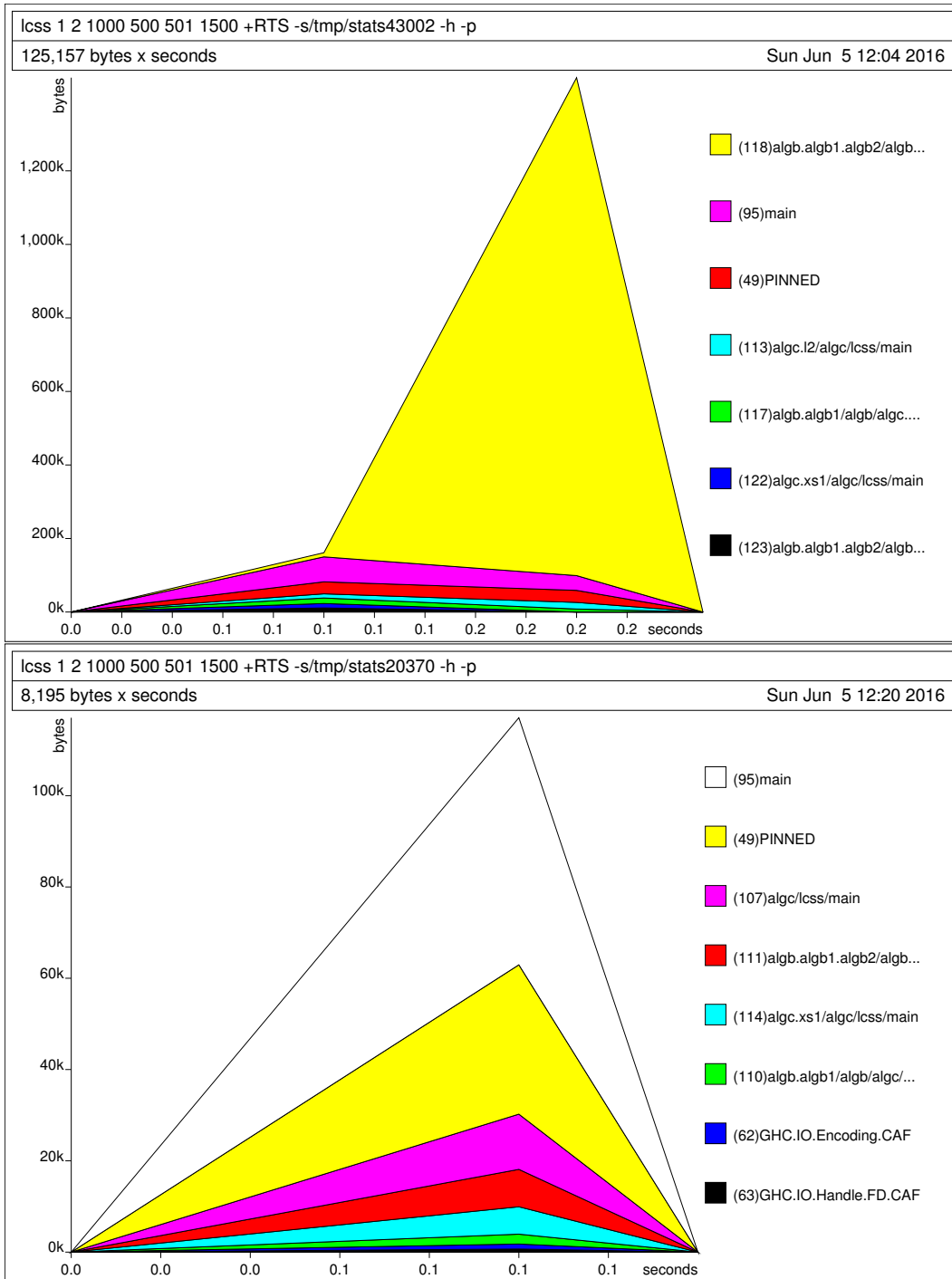


Figure 3.2: Heap profiles for original (top) and Autobahn-optimized 1css (bottom) program. The optimized program benefits from reduced memory usage at the call to `algb.algb1.algb2`, seen in yellow in the original and red in the Autobahn-optimized.

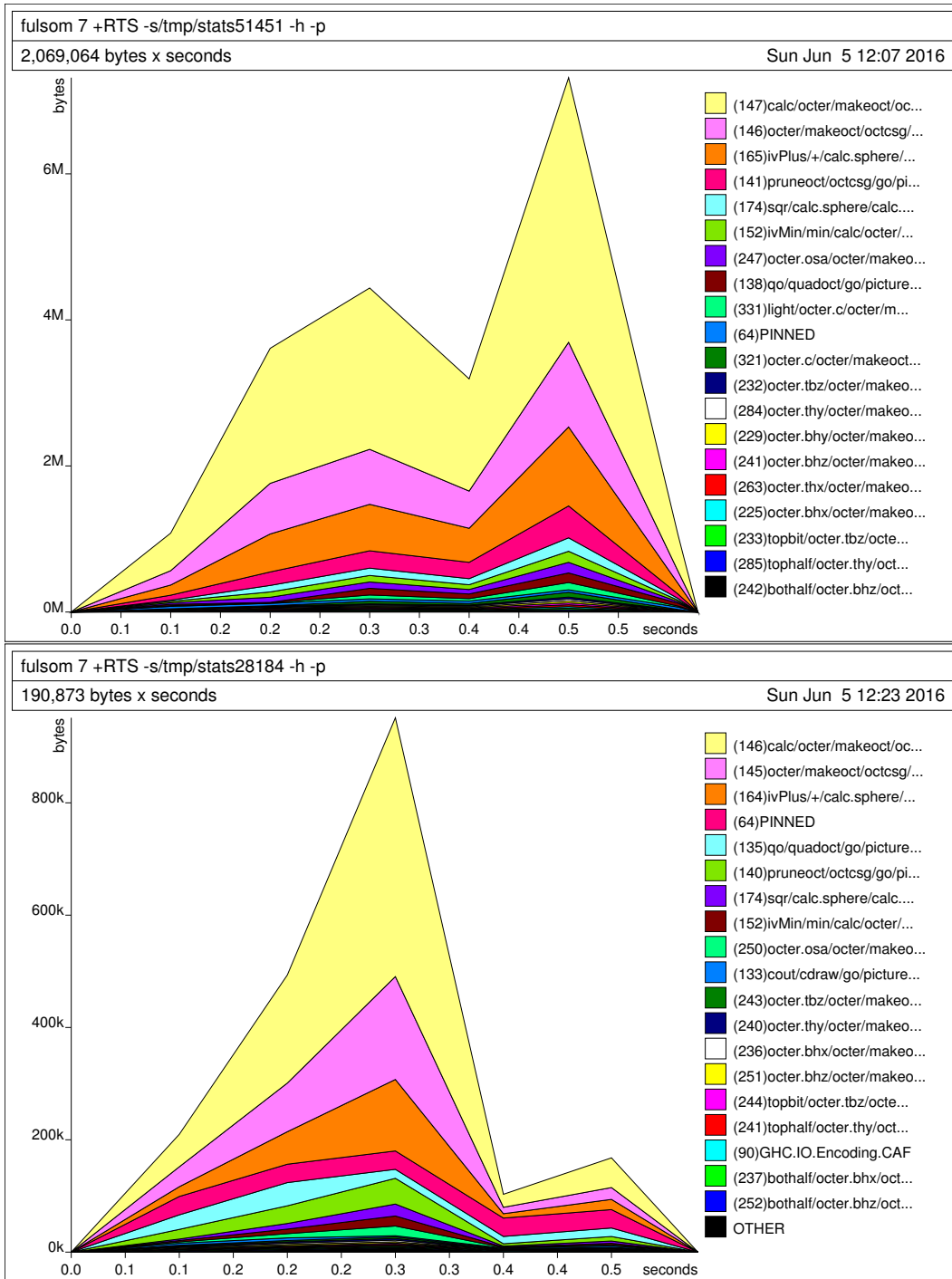


Figure 3.3: Heap profiles for original and Autobahn-optimized fulsom program respectively. The optimized program benefits from reduced memory usage across all cost centers.

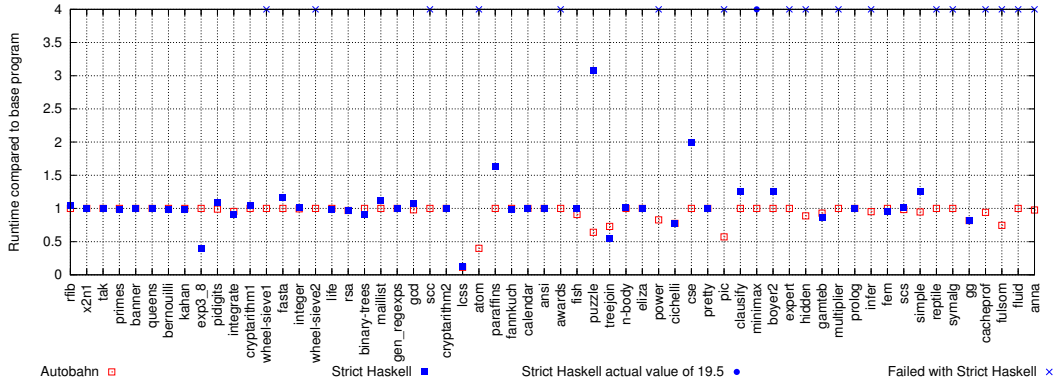


Figure 3.4: Comparing NoFib benchmarks run under Strict Haskell to those optimized by Autobahn for runtime.

Haskell, two did better than Autobahn: `exp3_8` and `treejoin`. In all other cases, Autobahn did as well as or better than Strict Haskell. One way to improve the performance of programs whose performance degrades under Strict Haskell is to add laziness annotations. Adding those annotations requires determining where to insert those annotations, which is another hard problem [CF14]. Inserting laziness annotations could also be done with Autobahn. Autobahn explores the space of programs that have one type of annotation. By replacing the type of annotation to laziness annotations, Autobahn could find opportunities for laziness in overly strict programs as well.

3.4.4 Case Study: gcSimulator

To evaluate how well Autobahn performs on real programs, we used Autobahn to optimize `gcSimulator`. `gcSimulator` analyzes ElephantTracks [RGM13] traces of Java programs to better understand the performance and behavior of novel garbage collection algorithms. `gcSimulator`'s input, the ElephantTracks traces, are on the order of gigabytes. As a result, `gcSimulator` can take several minutes to finish. This makes running Autobahn very time consuming. To optimize `gcSimulator` in a reasonable amount of time, we used the first 512KB of one of the traces as input for the program, specifically, a prefix of the trace of the `batik` program from the DaCapo benchmark suite [BGH⁺06]. We evaluated the Autobahn-optimized

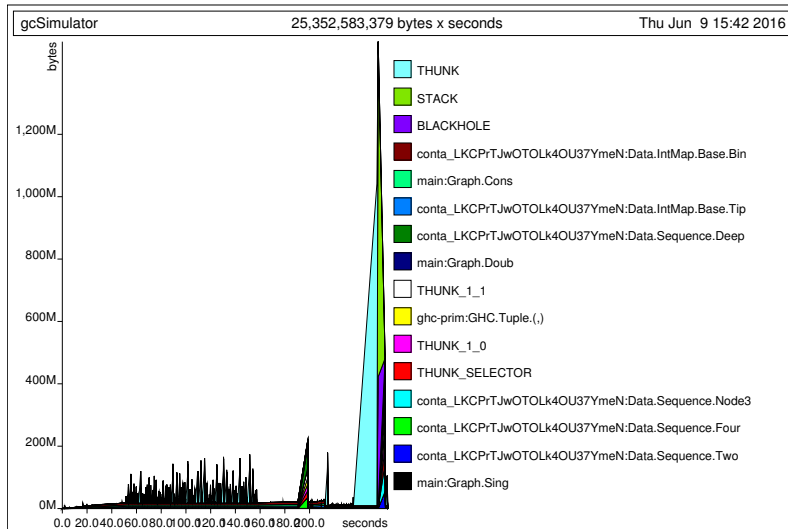
Input Data	Peak Alloc(MB)	Total Runtime	GC Time
training data	0.8	0.898	0.556
	0.8	0.905	0.417
$\frac{1}{3}$ of trace	0.140	0.616	0.094
	0.137	0.586	0.050
$\frac{1}{2}$ of trace	0.318	0.612	0.136
	0.021	0.812	0.069
full trace	0.072	0.914	0.444
	0.005	0.764	0.272

Table 3.1: Peak memory allocation, total run time, and garbage collection time for hand and Autobahn-optimized version of `gcSimulator`, normalized to the bare program Each colored band has two rows: the top row is the hand-optimized version, the bottom row the Autobahn-optimized version.

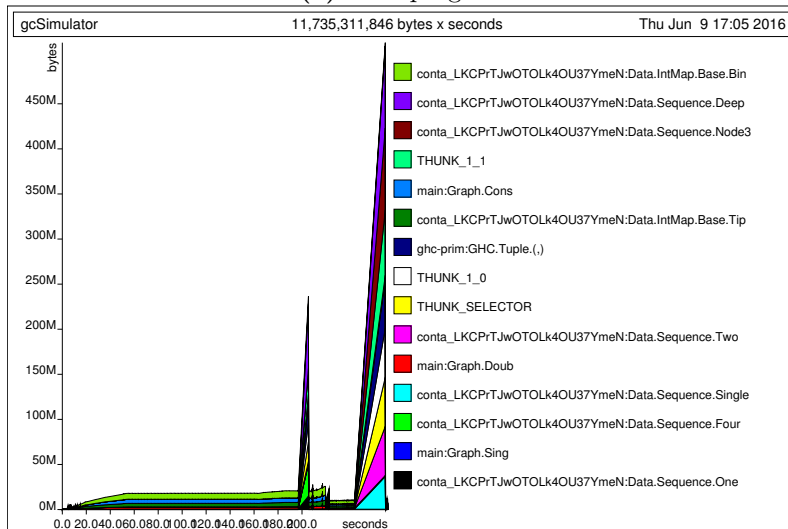
program on increasing amounts of the same trace. Of note, Autobahn started with a *bare* version of the program, a version of the program with no annotations.

Table 3.1 shows the results of this experiment. When run on the training data, both the hand-annotated and Autobahn-optimized version use 80% of the peak memory of the bare version. As the trace size increased, the Autobahn-optimized version greatly decreases its memory usage. With the full batik trace, the Autobahn-optimized version uses less than 1% of the memory of the bare version. This contrasts with the hand-annotated version which uses 7%. The reduce peak memory usage resulted in decreased garbage collection times and therefore a reduced run time. In the end, the Autobahn-optimized program ran at 76.4% of the bare version.

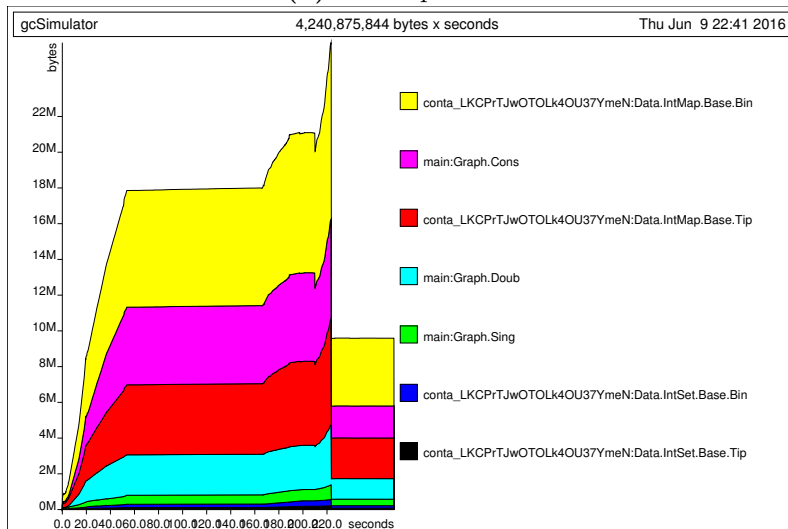
We also reviewed the changes in the heap profile. Figure 3.5 shows the heap profiles of the three programs on half of the `batik` trace. Compared to the unannotated version, the hand-annotated version does reduce the overall memory usage. The overall shape of the profiles is mostly the same, dominated by a peak at the end of the program. In contrast, the Autobahn-optimized program, while also decreasing memory usage, also changes the allocation behavior. In particular, there is no sharp rise in allocations at the end of the program. Together, Table 3.1 and Figure 3.5 show that optimizing on a small training set can provide benefits to the program even when the program is given larger inputs.



(a) Bare program



(b) Hand-optimized



(c) Autobahn-optimized

Figure 3.5: Heap profiles of gcSimulator on $\frac{1}{2}$ of the batik trace.

Input Data	Peak Alloc (MB)	Total Runtime	GC Time
A (46MB)	1.0	0.869	0.979
	0.457	0.661	0.434
B (50MB)	0.999	0.907	1.062
	0.853	0.830	0.808
C (51MB)	1.0	0.638	0.855
	0.834	0.754	0.671
D (68MB)	1.0	0.900	0.988
	0.810	0.787	0.694

Table 3.2: Peak memory allocation, total runtime, and GC time for Autobahn-optimized version of two Aeson driver programs, normalized to the bare program. For each band, the first row shows the results for `validate` and the second for `convert`.

3.4.5 Case Study: Aeson

Aeson [Bry16] is a Haskell library for parsing a JSON file containing a list of records. To test whether we could infer different annotations for different uses of this data structure, we constructed two driver programs that use the Aeson library: `convert` which fully converts a JSON file into a data structure, and `validate` which detects if a file is a valid JSON file. `convert` benefits greatly from a strict use of the library since `convert` needs to store the data in the JSON file into a data structure. Conversely, `validate` benefits from a lazy use of the library since `validate` does not need to parse the data contained in the JSON file. To limit the search space, We picked only a necessary subset of the Aeson library.

Autobahn optimized each driver for run time. To allow for improvement, the driver programs use the “wrong” version of the Aeson parser. In other words, `validate` used a strict version of the parser from Aeson and `convert` used a lazy version of the parser. During optimization, Autobahn must try correct the strictness annotations.

For inputs, we used JSON files published by the City of Chicago [jso12]. For training, we used `objects.json`, a 12MB file. For testing, we used 4 different traces of increasing size, which we call A,B,C, and D. We compiled the driver/library pairs using the same settings as NoFib except we added `-rtsopts` to gather run

time information and removed `-funbox-strict-fields` because Aeson does not use strict datatypes.

Table 3.2 shows that Autobahn optimized the two programs differently in spite of using the same fitness function and the same data. The `validate` driver saw run time improvements across all 4 files compared to its bare version, peaking at 63.8% improvement on file C. The `convert` driver not only saw improvements on its run time, but also in its peak memory usage. When run with file A, the Autobahn-optimized `convert` driver dropped its peak memory usage to 45.7%. Overall, Autobahn lowered `convert`'s peak memory usage to 71.7% of the bare version, suggesting Autobahn focused on reducing space usage to reduce the run time. In contrast, Autobahn only lowered the peak memory usage of `validate` to 99.98% of the bare version, suggesting Autobahn found other ways to try and reduce the run time.

3.4.6 Ten-fold Cross-validation

Since Autobahn runs a genetic algorithm with a randomized starting generation, we wanted to ensure Autobahn could provide reliable results when trained on different but representative inputs. To that end, we performed a version of ten-fold cross-validation on `gcSimulator` and the `convert` driver for Aeson. For each of our case studies, we chose ten input files. For each input file, we optimized the program on that input file as training data to create ten different programs. We optimized the programs for run time. Then we ran each version on all ten inputs and measured the run time and live size of these programs. We compared this data to the bare version of these programs.

For `gcSimulator`, we chose ten traces from different DaCapo benchmark programs. Due to the size of the traces resulting in long running times, we trained `gcSimulator` on the first 35 million lines from each trace. We tested the optimized programs on the full traces.

For Aeson, we chose ten different JSON data files from data made available by the City of Chicago. The sizes ranged from 32 to 68MB. We trained and tested

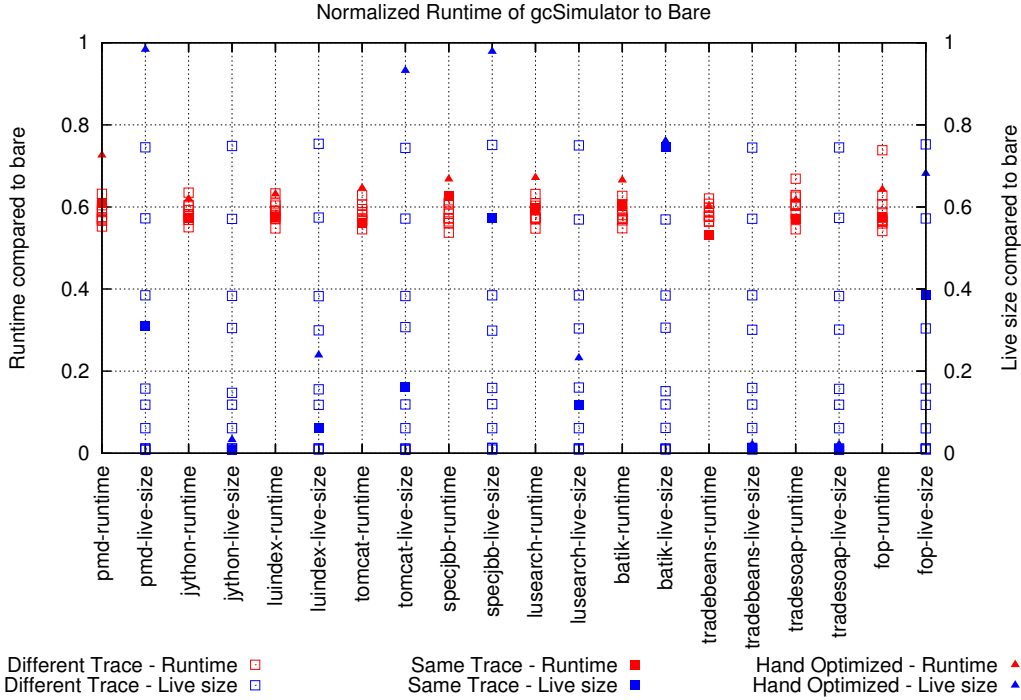


Figure 3.6: Ten-fold evaluation for `gcSimulator`, showing runtime and live size performance improvements of Autobahn-optimized versions of `gcSimulator` compared to the bare program. We highlight points where the Autobahn-optimized program ran on its training trace. We also show how the hand-annotated program performed.

the optimized programs on the full JSON files.

Figure 3.6 and 3.7 show the results of this experiment. On the x-axis, each label corresponds to the program trained on that file and the metric we measured. Each point on the graph is one of the ten runs, each on a different test input. For `gcSimulator`, the performance of the hand-optimized version on that training data is labeled with a triangle.

Figure 3.6 shows us the Autobahn-optimized version ran around roughly 60% of the bare program’s run time. In fact, the run time geometric mean is 58.6% for Autobahn-optimized programs compared to 64.8% for the hand-optimized version. We believe the live size reduction contributed to the run time. Autobahn-optimized versions reduced the live size to to 9.6% of the bare program’s live size. This is a large savings compared to the hand-optimized versions’ reduction to 22.8% over the ten programs.

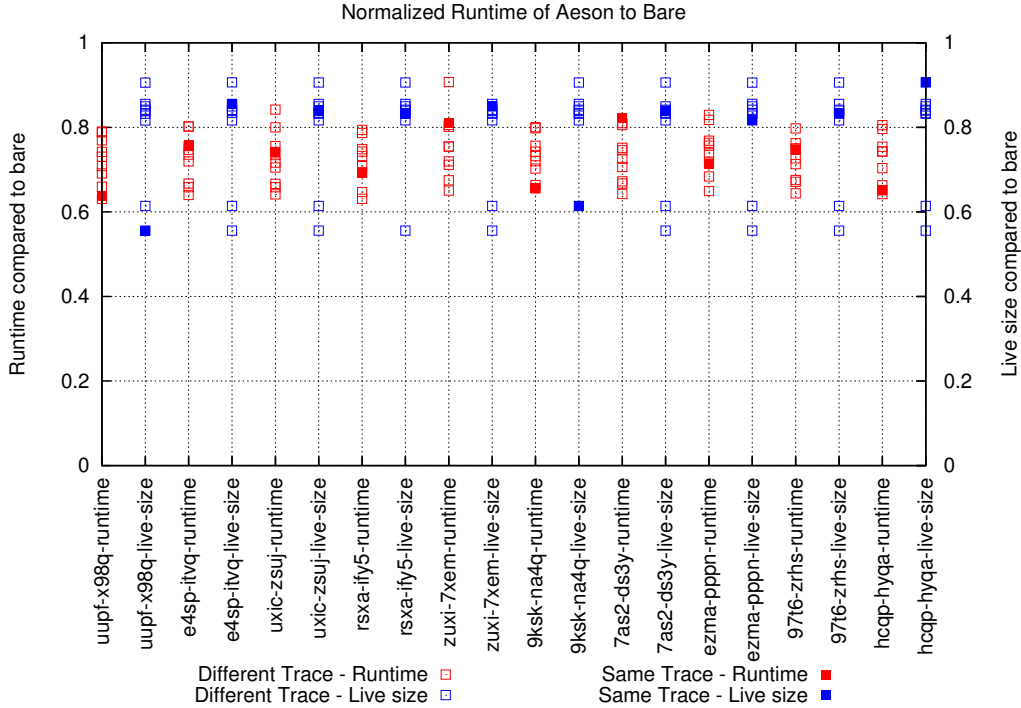


Figure 3.7: Ten-fold evaluation for `convert`, showing runtime and live size performance improvements of Autobahn-optimized versions of `convert` compared to the bare program. We highlight points where the Autobahn-optimized program ran on its training trace.

Autobahn reduced the `convert` driver’s run time to a geometric mean reduction of 65.2% of the bare program’s run time. In addition, it reduced the live size to 78.6% of the bare program’s live size. Recall `convert` must parse and store the JSON data records fully. We believe this is the reason the reduction in live size benefited run time performance.

3.4.7 Autobahn Performance

Autobahn runs for a set number of rounds before terminating and has to run the program to obtain fitness scores. For a user, it is important to know how long they will have to wait to get an improved program. We report the total runtime of Autobahn for the NoFib benchmark experiments and our two case studies here.

Figure 3.8 shows the runtime of Autobahn on the 60 programs from the NoFib benchmark. Overall, Autobahn on average ran for 14 minutes and 12 minutes to

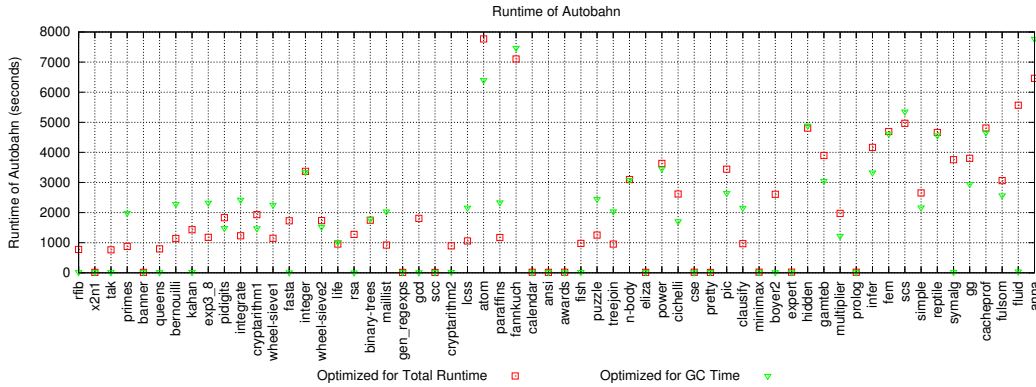


Figure 3.8: Graph demonstrating the run time of Autobahn itself when optimizing NoFib benchmarks for runtime and garbage collection time.

optimize for run time and garbage collection time respectively. Variations from the mean come from running the different versions of the program to obtain a fitness score. On a few benchmarks, Autobahn runs for almost no time. The benchmarks themselves reported running in 0 CPU seconds or spending 0 seconds in garbage collection. In those cases, Autobahn cannot optimize further and thus returns the original program without running the genetic algorithm.

As for our case studies, optimizing `gcSimulator` took 5 hours and 34 minutes and optimizing `Aeson` took 2 hours and 12 minutes. These run times were dominated by the run times of the programs, not Autobahn itself. In particular, `gcSimulator` has to process a reasonably large trace and simulate a program, complete with garbage collection cycles, in a garbage collected language.

Some of the run time for Autobahn comes from parsing, modifying, pretty printing, and recompiling the program. We can reduce the run time by using the `ghc` compiler API to parse and modify the program.

3.5 Related Work

3.5.1 Static Analysis

Using a static analysis to find opportunities for strictness is a well researched problem. Compilers use some form of strictness analysis. Strictness analysis finds expres-

sions that will be evaluated at least once and can safely be strictly evaluated [Myc82]. Some work relaxes the constraint, finding any expression so long as it is cheap to evaluate [Fax00]. Other work extends the analyses into the type system to find more opportunities [TWM95, SM10, VH15]. However, static analyses must be approximate. Furthermore, compiler analyses must consider all possible inputs. Autobahn is a dynamic analysis and not part of the compiler. Therefore, Autobahn does not have to be approximate or guarantee termination on all inputs. However, the programmer must reason about the soundness of the resulting program, which may not be easy.

3.5.2 Dynamic Information

Many have augmented compilers with dynamic information to improve optimizations. In this section, we will focus on works that use this information to find opportunities for strictness.

Ennals et. al. created an evaluation strategy that finds expressions that are cheap to always evaluate and force them with strictness [EP03]. The strategy chooses expressions to evaluate guided by an online profiler. If the program performance degrades as a result, the run time system would abort some of its findings on the fly.

Trilla et. al. added `par` calls at compile time with respect to their profiler's output [TR15]. Their profiler runs the program to determine which `par` calls are detrimental to performance and disable them. Harris et. al similarly profiled programs and identified thinks that can be expanded safely and correctly in another core [HS07]. They then rewrite the program to add those parallel tasks. Both techniques incur the overhead of profiling the program at run time to avoid detrimental behavior.

Finally, Sun et. al. reduced the annotations reported by Autobahn with `ghcprofiler` information [SF18]. The profiler output is used to locate hot spots. They use the hot spots to enhance the search space for Autobahn and reduce the number of annotations produced by Autobahn.

3.5.3 Other Approaches

Strict Haskell was introduced in `ghc` 8.0.1 as two pragmas, `-XStrict` and `-XStrictData`. These pragmas force Haskell to change its evaluation strategy from lazy to strict. As we saw in our experiments, the programmer does still have to reason about the effect a fully strict evaluation scheme has on their program or risk non-termination. The programmer can alleviate these by adding laziness annotations, an approach taken by Chang et. al. [CF14].

Chang et. al. identified locations to add laziness to a strict program [CF14]. They added laziness annotations (or delays) guided by the results of their dynamic analysis. An alternative to Autobahn’s genetic algorithm is adapting their approach to add strictness annotations. The alternate approach would still suffer from Autobahn’s soundness problem. Another approach would be to use this dynamic analysis to add laziness back into Strict Haskell programs.

StrictCheck allows programmers to specify the strictness of their Haskell functions and test those specifications on randomized input [FZL18]. StrictCheck returns counterexamples when the specification is incorrect, complete with explanation of the expected versus actual demand. StrictCheck does require the programmer to understand the semantics of demand to properly write the specification. In exchange, the counterexamples guide the user to a correct specification. In contrast, Autobahn requires no such understanding to optimize a program, only to decide whether the programmer should use the optimized program. If a programmer could translate the strictness inferred by Autobahn into StrictCheck specifications, they would have a tool to understand the classes of input that could trigger unsound behavior.

3.6 Conclusion

In this chapter, we showed strictness annotations can lead to more performant Haskell programs, but are difficult to place. As a solution, We presented a genetic algorithm to find strictness annotations for Haskell programs. We presented

the results of the more recent version of this search, Autobahn. Autobahn improves the performance of both small programs like NoFib and large programs like gcSimulator. Autobahn also infers different annotations on a library depending on the fitness function.

Chapter 4

Prioritized Garbage Collection

4.1 Introduction

Software caching and garbage collection (GC) do not play well together. The problem is that they embody conflicting goals and tradeoffs. Caching aims to achieve the highest hit rate given a particular storage budget: the larger the cache, the higher the hit rate. Unfortunately, most widely-used garbage collection algorithms have a cost proportional to live memory [HB05]. In this setting, the benefits of a larger cache are less clear because improvements in hit rate are offset, to some degree, by additional GC costs. The penalty can become particularly high if the cache starts competing with the rest of the program for resources, increasing memory pressure and GC overhead.

Figure 4.1 shows how the performance of a cache changes as we vary the number of entries while keeping the overall heap size fixed. This benchmark uses Google’s Guava caching library configured with a least-recently-used (LRU) eviction policy. We measured the time it takes (Y-axis) to serve a predefined sequence of requests under each cache configuration (X-axis). Time for the application is split into two categories, garbage collection time and *mutator time*. Mutator time is the time the application spends working and is defined by the difference between the total runtime and garbage collection time. Looking from left to right, the two opposing trends are clearly visible. Larger caches have higher hit rates, incurring

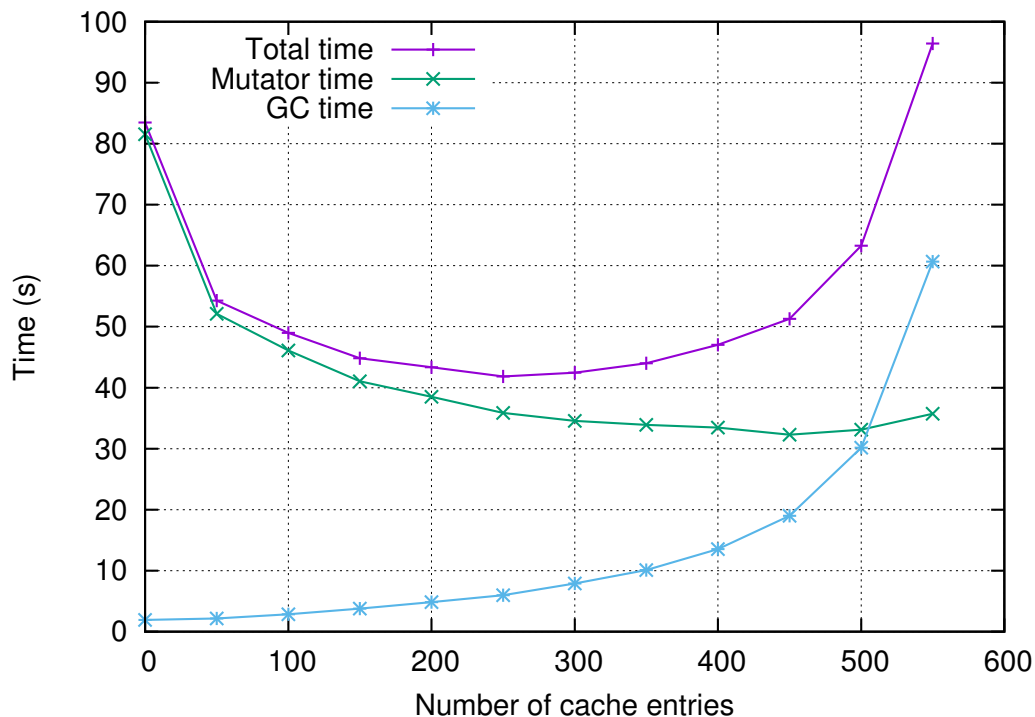


Figure 4.1: Competing tradeoffs: as cache size increases, miss time goes down, but GC time goes up.

fewer misses and lowering mutator time. At the same time, though, larger caches increase GC time. At the far right, the cache occupies almost all of available memory, increasing not only the per-GC cost, but also the frequency of collection, causing GC time to dominate total time. At the limit, the program runs out of memory and crashes. Unbounded growth of caches (and related structures, such as indexes) is a primary cause of memory leaks and performance problems in Java [MS03, XBQR11].

The scenario above is a simple and controlled experiment – real applications have much more complex behavior, including multiple caches, possibly with different eviction policies, different memory footprints, and different patterns of locality, as well as significant non-cache data structures. Getting the most out the available memory resources without triggering memory pressure poses a significant challenge.

Existing runtime systems provide some mechanisms to support memory-sensitive data structures, but they are sorely lacking in ways to configure and control the policies that govern these mechanisms. As an example, the Java runtime pro-

vides *soft references*, which the garbage collector can clear at its discretion to avoid running out of memory. A common programming strategy is to store each cache entry in a soft reference, allowing the collector to reclaim individual entries if necessary. The application, however, has little or no control over when this process is triggered, or over which soft references are cleared and in what order. Some Java Virtual Machines (JVMs), such as HotSpot, use LRU-like policies that are clearly designed for caches, but are too coarse for complex applications where a single global policy is not appropriate. In Section 4.2 we describe this problem in more detail, and in Section 4.4 we show its effect on hit rate.

Not surprisingly, soft references are widely shunned. In fact, the official documentation for the Android runtime library explicitly warns against using them for caches [Inc16]: “In practice, soft references are inefficient for caching. The runtime doesn’t have enough information on which references to clear and which to keep. Most fatally, it doesn’t know what to do when given the choice between clearing a soft reference and growing the heap. *The lack of information on the value to your application of each reference limits the usefulness of soft references.* References that are cleared too early cause unnecessary work; those that are cleared too late waste memory.” In Section 4.2 we present detailed empirical measurements showing that these are real problems.

This chapter presents **prioritized garbage collection**, an automatic memory management system designed to address the deficiencies outlined above by providing explicit support for software caches and other space-sensitive data structures. The key idea is to enable better cooperation between the application and the garbage collector. In our system, the collector provides the *mechanisms* for measuring and enforcing memory usage, while the application dictates the *policies* that drive these mechanisms. The application and the collector cooperate through a simple API that is designed around a new kind of reference object we call a **PrioReference** (short for “priority reference”). It resembles a soft reference, except that the application can specify both the *global policy* (when to trigger eviction and how much memory to

reclaim) and the *local policy* (which priority references to clear and in what order).

The chapter makes the following contributions:

1. We quantify the performance problems of existing cache implementations by driving them with a range of workloads across a range of sizes. Not surprisingly, choosing a fixed cache size, particularly in terms of number of entries, is brittle. We also demonstrate the limitations of soft references as a mechanism for implementing these data structures.
2. We present a new reference type called `PrioReference` that allows application code to communicate the relative value of not reclaiming its referent object (and transitively, reachable objects). `PrioReferences` are grouped into `PrioSpaces`, which specify the details of the total memory limit and eviction policy for each group. A common configuration is one `PrioSpace` for each cache, but the mapping is up to the application to decide.
3. We describe the design and implementation of a garbage collector that enforces these policies. The key mechanism is a modified closure phase that visits `PrioReferences` in order from highest to lowest priority, stopping when the target space bound is reached. Unmarked references are implicitly evicted, and are reclaimed immediately by the sweeper without touching them.
4. We present a space-sensitive cache, which we call a *Sache*, built on our new API. The *Sache* supports LRU and GreedyDual [CI97] eviction policies by changing the way it computes the reference priorities. Our system allows the user to set the target memory footprint in terms of available memory, so the *Sache* expands and contracts automatically to avoid memory pressure.
5. We report performance results obtained by driving a key-value store with a range of workloads. We use representative workloads to systematically explore the performance space and quantify the problems. We compare our cache to Google's Guava caching library on web traffic traces.

The remainder of this chapter is organized as follows. Section 4.2 describes the problem in more detail and explores the space of interactions between caches and garbage collection. Section 4.3 describes the design and implementation of our garbage collector mechanisms and the Sacle data structure. Section 4.4 presents Sacle performance results and compares them to traditional caches. Finally, Sections 4.5 and 4.6 review related work and conclude.

4.2 Problem

It is not easy to implement software caching in a garbage-collected language. One reason is that cache performance is governed by a space-time tradeoff that is in direct opposition to the tradeoff in garbage collection. Another reason is that garbage collected languages provide poor support for implementing *any* algorithm or data structure that is inherently space sensitive. In this section, we discuss these issues in detail, and present measurements that illustrate the problem.

The results below are obtained using JikesRVM version 3.1.2 [Jik05]. While our benchmarks can run on any JVM, we use JikesRVM for these experiments both because it can report many detailed measurements, and because it allows direct comparison with our new algorithm, which we implemented in JikesRVM. Section 4.4 contains a detailed description of the experimental setup and methodology.

4.2.1 Existing cache implementations

Google’s Guava library is a widely-used infrastructure for implementing software caches. It implements a simple `get/put` interface for keys and values, and offers a variety of eviction policies to manage the capacity of the cache. Even with this library, however, there are several significant challenges to obtaining good cache performance:

Choosing a cache size is difficult. The most straightforward eviction trigger is capacity-based: the programmer chooses the maximum number of entries (key-value pairs) that the cache will hold. When the cache grows beyond this limit, it evicts

```

Cache<Key, Graph> graphs = CacheBuilder.newBuilder()
    .maximumWeight(100000)
    .weigher(new Weigher<Key, Graph>() {
        public int weigh(Key k, Graph g) {
            return g.vertices().size();
        }
    }).build(
    new CacheLoader<Key, Graph>() {
        public Graph load(Key key) {
            return createExpensiveGraph(key);
        }
    });

```

Figure 4.2: Guava cache that stores graphs and uses a weighing function to represent their size.

entries in least-recently-used order.

The challenge of this policy is how to choose a good size: too small and the cache will underperform; too large and the program will slow significantly or crash due to memory pressure. In many cases, the cached values vary widely in size, so the same set of entries could account for wildly varying quantities of data.

One potential solution is to measure representative workloads during testing and configure the cache accordingly (e.g., by assuming an average size key and value). Unfortunately, this approach is brittle: unless the workload is extremely uniform and predictable, the number of entries is not a reliable predictor of the memory footprint of the cache. If actual workloads in deployment differ substantially, then performance will suffer.

There is no easy way to measure memory footprint. To handle these cases, Guava can manage cache capacity in terms of an application-specific “weight”. The programmer implements a `weigh()` method that can compute a weight value for any entry. The weight method could, for example, count the number elements in a container. Entries are evicted in LRU order to keep the total weight under the limit. Example code is shown in Figure 4.2.

Implementing an accurate weighing method, however, is not always easy. Ideally, we would like to know the exact size (in bytes) of each cached value. In the case of simple structures, such as strings, an accurate size is easy to compute. Measuring the size of complex data structures is more difficult. One problem is encapsulation: it might not be possible to access the hidden implementation of a

class. Another problem is structure sharing: when measured independently, the shared substructures could be counted multiple times, distorting the total weight.

One alternative is to explicitly measure the size of a data structure at runtime using reflection. This is the approach taken by JAMM, which is based on the JVM Tool Interface [Bel]. While accurate, its cost is so high that it is not practical for use in production settings. For example, measuring a data structure with 1 million objects can take 5 seconds of wall clock time, and causes the benchmark to run over 100× slower than the approach we propose here.

Eviction doesn't work. The purpose of eviction is to control cache memory use by freeing low-value entries. In a garbage collected language, however, eviction does not achieve this goal. The cache can remove entries and null out all references to them, but the memory is not actually reclaimed until a garbage collection occurs. Guava attempts to address this problem by performing eviction lazily, but we have observed cases where eviction actually makes memory pressure *worse*. If the cache misses on a recently-evicted entry, then it will create a new one, resulting in two copies in memory at the same time. A secondary effect, which we show at the end of this section, is an increase in the allocation rate. One of our observations is that it *only* makes sense to do eviction at collection time, when memory is actually reclaimed.

Soft references don't do the right thing. Recognizing that caches can be a source of memory pressure, Guava also offers the option of storing entries in soft references. The Guava documentation claims that soft references are reclaimed in LRU order when memory is tight. While this policy is not required by the Java standard, we found that Oracle's HotSpot JVM does implement such a strategy [Ora15]. It has two serious limitations, however: first, the policy is hard-wired to LRU, and second, the LRU ordering is global for all soft references. Large Java programs, such as web applications, can have multiple caches, and the global LRU order is particularly problematic if these caches are accessed with different frequencies. Entries in a less-frequently used cache all wind up at the end of the LRU queue, resulting in

the entire cache being dumped. Figure 4.11(a) illustrated this effect: the bigger the differences between the caches, the worse the impact of the soft reference policy.

4.2.2 Exploring cache-GC interaction

To study these problems in detail, we implemented a simple key-value store in Java that we can drive with a range of workloads and under a range of conditions. Our goal is to isolate cache performance and its interaction with the garbage collector. In the context of a larger application, these effects might be hard to separate from unrelated program behavior.

Trace files. The input to the driver is a trace file that specifies the workload as a sequence of key-value requests. The keys are just names, but the values represent data structures of varying sizes. The goal is to model caching of data that has a non-trivial structure (as opposed to strings, for example). Real-world examples might include a parsed XML document in tree form, or memoized computations in an optimizing compiler. The trace file itself just specifies the size of each tree in number of nodes. Figure 4.3 shows an example fragment of a trace file.

We generate each trace file according to a set of parameters: (1) number of unique keys, (2) minimum and maximum sizes of the values, (3) the distribution of value sizes, (4) the number of key requests (trace length), and (5) the temporal distribution of keys in the trace.

The set of unique key/value pairs is generated by choosing value sizes at random from a Pareto distribution. Many kinds of workloads, including web requests and file accesses, have been found to follow this kind of power law distribution [AXF⁺12, CBC95, New05]. The sequence of key requests is also drawn from a Pareto distribution, which governs the temporal locality of the trace. Here, higher alpha values create more locality, and lower values spread out the distribution more evenly. We use an alpha value of 0.1, which is on the low end and requires caches to be larger to achieve a high hit rate. The traces range in length from 10,000 to 50,000 key requests, with 2000 to 5000 unique keys.

key_8	179074
key_12	180434
key_1	150999
key_188	126021
key_2	154588
key_28	119220
...	

Figure 4.3: Example trace file. The number associated with each key determines the **size** of the data structure that is stored as the value.

Execution. Our driver initializes the cache, reads the trace and sends the cache a sequence of get and put operations. Its behavior is configured using several parameters: (1) size of the cache (number of entries for Guava), (2) the max heap size, and (3) the cost of a miss. The miss cost models the time to fetch data from a remote source or recompute it, which is proportional to the size of the resulting data assuming the miss cost for a given key-value pair is not probabilistic. In these experiments the miss cost only represents the time it would take to transmit the data over an uncongested 10MB/s network connection. Higher miss costs (for example, modeling a more expensive computation or a database query) would only exaggerate the shape of our graphs. The trace is processed as follows:

1. For each key in the trace, call `Cache.get(K)`.
2. If it **misses**, the driver uses the value number in the trace file to construct a tree of the given size. It delays execution for a time proportional to the size of the tree and the miss cost. It then stores the key and value (tree) in the cache using `Cache.put(K,V)`.
3. If it **hits**, the cache returns the associated tree data structure. The driver performs a modest computation on the tree that visits all the nodes.

Measurements. We record several measurements for each complete run of a trace file:

- Total time, mutator time, and GC time (with GC time broken down into sub-categories).

- Hit and miss rate, as well as time spent servicing misses
- Number of garbage collections
- Total memory allocation

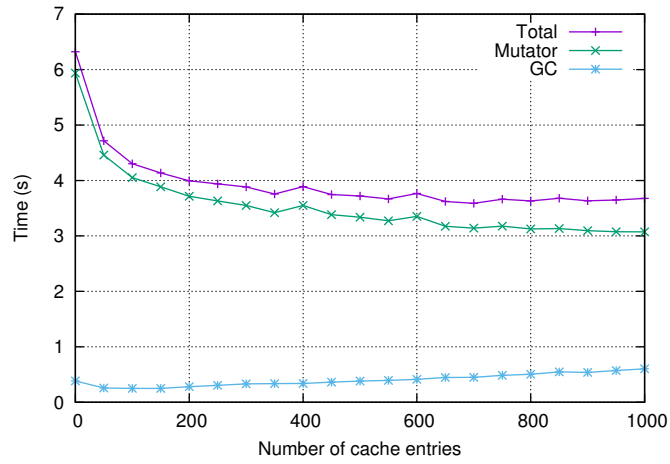
4.2.3 Guava performance

The three graphs in Figure 4.4 are typical of the performance we see for a traditional cache implementation. For these experiments, we vary the capacity of the cache from 100 to 1000 entries (the X axis) and measure the time to process the entire trace (Y axis). The heap size is fixed at 115MB. We show mutator (application) time, GC time, and total time. The three graphs differ in the sizes (in bytes) of the cached values, which affects both the memory footprint of the cache and the cost of a cache miss:

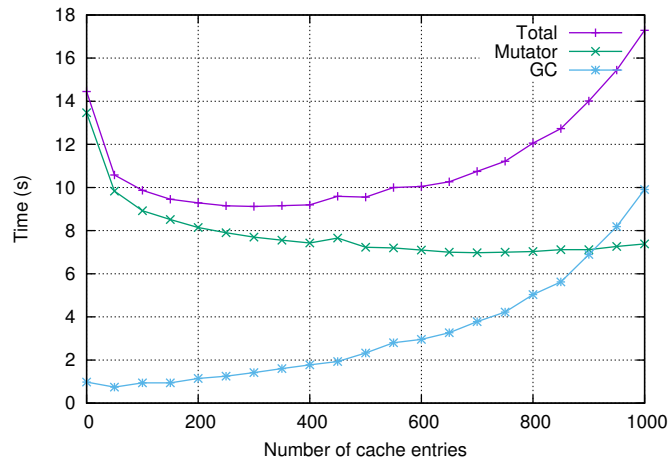
- Figure 4.4(a) shows the performance on a trace with small-sized trees (10K to 50K bytes). In this case, the total time continues to drop all the way out to 1000 entries, suggesting that the cache could probably accommodate more before incurring a memory cost.
- Figure 4.4(b) shows the same graph for medium-sized trees (50K to 100K bytes). It exhibits the typical “bowl” shape for the total time, which is explained by the opposing curves of the miss cost (going down) and GC time (going up). Miss costs are accounted for in the mutator time. GC costs at the right edge go up steeply because the cache is approaching the maximum heap size and causing memory pressure.
- Figure 4.4(c) shows the results for larger trees (100K to 200K bytes). Under this workload, only a narrow range of cache sizes is usable. Too few and the miss costs are huge; too large and the cache runs out of memory.

Looking at other metrics provides further insight into this behavior. At the left of the graph (when the number of entries is small) several factors are hurting performance.

(a) Workload of small values (10K-50K)



(b) Workload of medium values (50K-100K)



(c) Workload of large values (100K-200K)

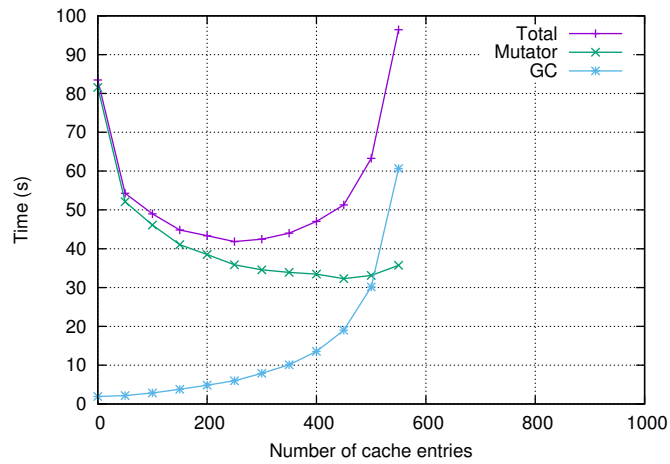


Figure 4.4: Guava performance under three workloads: choosing a good number of entries is difficult.

First, the number of misses is higher, incurring the cost of “fetching” (rebuilding) the value. Second, evictions are more frequent, filling up memory with garbage. Third, rebuilding the values increases total allocation costs. Figure 4.5 plots the total amount of allocation for a run of the trace with medium-sized trees under different cache sizes. The smaller cache sizes cause a significantly higher allocation rate.

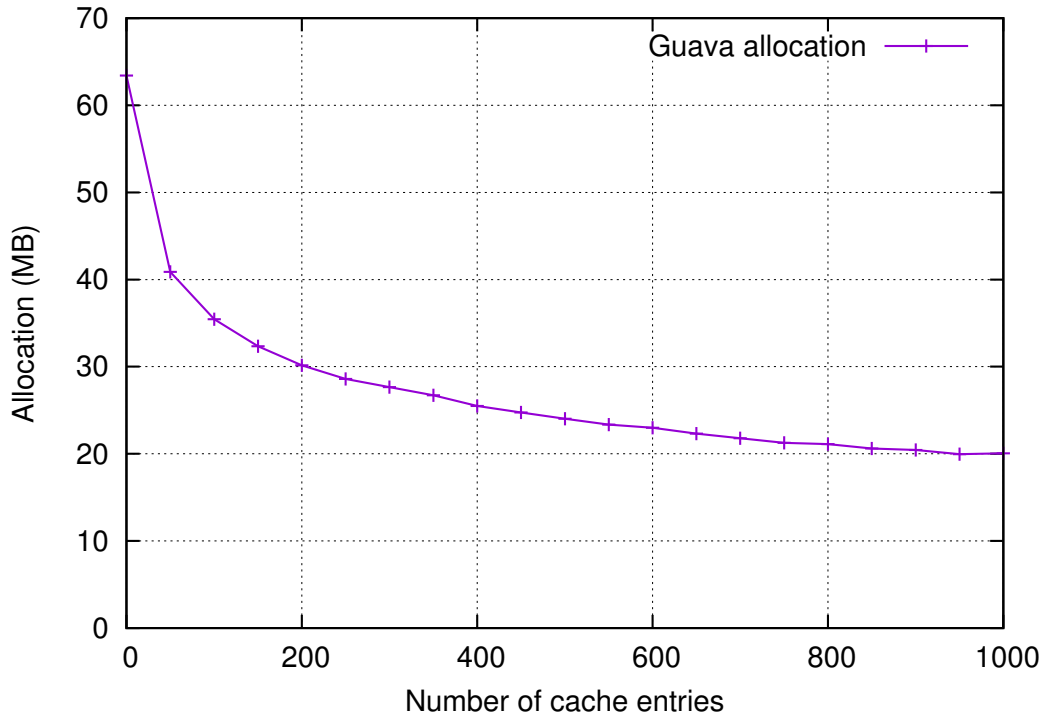


Figure 4.5: Memory allocated under the workload of medium-sized values: Under-sizing a cache (left side) incurs the cost of more misses as well as the cost of increased allocation.

These results suggest that bounding a cache unnecessarily when more memory is available can lead to performance degradations that are nearly as great as having a cache that is too large. In fact, we conclude that it does not make sense to evict entries at all if there is sufficient memory to hold them.

4.3 Prioritized Garbage Collection

In this section, we introduce new runtime support for software caches and other space-sensitive data structures. The primary goal is to provide an effective mechanism for implementing eviction policies that take into account both memory utilization (which only the garbage collector knows) and the relative value of cache entries (which only the application knows).

4.3.1 API

Prioritized garbage collection is a cooperative technique, so the central feature of our system is an API that allows application code to communicate directly with the garbage collector. Our API is modeled after Java *reference objects*, which already play a similar role. Each reference object points to a single target referent, and the particular type of reference object chosen tells the collector how to treat the referent – typically, it specifies when the referent can be reclaimed even if it is still reachable. Since the application is not aware of when collection occurs, it discovers that a reference has been cleared only when it attempts to get the referent and the result is null.

4.3.1.1 PrioReference

Following this model, a *priority reference* is a new reference type that may be cleared by the collector in order to bound memory use, but only after *all other references of lower priority have already been cleared*. The `PrioReference` class definition is shown in Figure 4.6. Each `PrioReference` has a priority value, which is simply an integer – higher values represent higher priorities. The application is free to choose and change this value in any way, allowing the programmer to implement a custom eviction policy for a collection of `PrioReferences`. In our cache implementation (below), for example, each cached value is held in a `PrioReference`, and we implement LRU eviction by ensuring that the most recently hit entry has the highest priority.

```

class PrioReference<T> extends
    java.lang.ref.Reference<T>
{
    // -- Constructor: the new reference belongs
    // to the given space
    PrioReference(T obj, PrioSpace<? super T> space);

    // -- Get the referent
    public T get();

    // -- Get and set the priority
    public void setPriority(int new_prio);
    public int getPriority();

    // -- Inquire about the memory footprint
    public boolean hasGCSize();
    public int getGCSize();
}

```

Figure 4.6: A priority reference holds a single referent with a given priority. The application can also inquire about the total amount of memory reachable through this reference.

The `getGCSize()` method allows the application to find out the total memory footprint of all objects reachable *only* through this reference. Ordinarily, this information is difficult or inefficient to compute, but our collector computes it as part of its normal marking phase, incurring little overhead. It needs to know this information in order to determine whether the size bound has been exceeded, so we opt to make it available to the application as well. The only caveat is that it is only computed at GC time, and so it is not guaranteed to be fresh or even computed at all. The `hasGCSize()` method asks whether or not the collector has computed the size. Once `getGCSize()` is called, the flag is reset until the next collection.

In Section 4.3.3, we describe the details of how size information is computed and used in the collection algorithm.

4.3.1.2 PrioSpace

In order to support multiple independent caches, priority references are grouped into *priority spaces*, each with its own memory bound. The `PrioSpace` class definition is shown in Figure 4.7. `PrioSpaces` are not spaces in the memory management sense, but rather a collection of references that are considered together. The collector considers each priority space separately, evicting the lowest priority references until

the total memory footprint reachable from the remaining references is smaller than the target bound. At the lowest level, this bound is expressed in bytes, but our API also allows it to be specified as a fraction of total memory or as a fraction of available memory, which can change dynamically as the program runs.

In a typical configuration, each cache would be managed in its own `PrioSpace`, but this one-to-one mapping is not required. For example, we can emulate `SoftReferences` by placing all `PrioReferences` in one `PrioSpace` and associating the priority of a reference with the time the program last uses a reference.

4.3.2 Measuring Memory Footprint

The job of our collector is to bound the total memory footprint of each priority space by keeping as many high-priority entries as will fit in the available space, and freeing the rest. In order to do this job the collector must be able to accurately measure the memory footprint of each entry, as well as its contribution to the total memory footprint of the priority space. There are several factors that complicate this computation. First, entries may consist of complex, pointer-based data structures, so it is not sufficient to measure only the size of the object directly pointed to by the priority reference. Second, we need to properly handle shared structures to avoid counting them multiple times. Third, we need to account for fragmentation to make

```
class PrioSpace<T>
{
    // -- Create a priority space with a specific
    //     target size in bytes
    PrioSpace(int bound);

    // -- Create an adaptive size priority space
    //     expressed in terms of percentage of the heap,
    //     either as fraction used or a fraction free.
    PrioSpace(float fraction, boolean used_or_free);

    // -- Get the actual memory footprint of the whole
    //     space after GC
    public boolean hasGCSize();
    public int getGCSize();
}
```

Figure 4.7: A priority space holds a set of priority references and governs their lifetime collectively under single policy.

sure that the sum of the sizes properly reflects the actual fraction of total memory used.

4.3.2.1 Fragmentation

Many kinds of memory allocators can suffer from *fragmentation*, in which small chunks of memory become unusable, taking away from the total available. Fragmentation is a concern for our algorithm because it could cause us to underestimate the total memory cost of a set of objects. Traditionally, fragmentation is divided into two categories: *internal* and *external* [WJNB95]. Our algorithm can easily account for internal fragmentation, but external fragmentation is more complex, as discussed below. In general, though, fragmentation has not been found to be a major problem for dynamic memory management [JW98].

Internal fragmentation is created when the allocator reserves more memory than is requested for an object in order to comply with alignment, padding, or size restrictions. For example, our current implementation uses a free list allocator with fixed size classes, so all small objects must be allocated into one of the 51 possible size denominations. Objects that do not fit perfectly are allocated in the next size up, leaving some number of bytes unused. Luckily, internal fragmentation is easy to account for: whenever the algorithm needs the size of an object, denoted $size(o)$ in this section, we use the *allocated* size, which includes unused padding, if any.

External fragmentation is created when sequences of allocations and deallocations leave unused memory in between objects. Unlike internal fragmentation, our algorithm cannot directly account for external fragmentation because it cannot determine if a given unused fragment should be charged to any particular priority space (or none at all). As a result, our algorithm views all free space as available for non-cache objects, potentially reducing the usable free space by the amount of the fragmentation. In practice, the free list allocator in MMTk has been observed to have low fragmentation [BCM04], but if it became a problem we could change the underlying allocator or collection algorithm, both of which are largely orthogonal to our technique. There is no reason we could not use a compacting collector, for

example.

4.3.2.2 Reachability

If we consider only the memory used by the objects immediately pointed to by each priority reference, computing size is easy and requires little additional collector machinery. However, many data structures have complex internal structure. Even strings typically consist of two objects: a string object and a character array. From a memory use standpoint, it makes sense for the size of a string to include the size of its character array.

We therefore define the total memory footprint of a priority space as the sum (in bytes) of the sizes of all objects transitively reachable *only* from the priority references in that space. We purposely exclude any objects that are also reachable from roots because clearing the associated priority references will not cause them to become garbage. In other words, we exclude the parts of the memory footprint that the priority space cannot control.

4.3.2.3 Structure Sharing

The size computation has algebraic properties that are crucial to ensuring we meet the target memory bound. If the size computed for a particular referent o is $size(o)$, the total size of a set of objects S should be given by the following formula:

$$size(S) = \sum_{i=0}^{|S|} size(o_i)$$

While the formula seems obvious, consider the case where two priority references share some internal structure. We need to be careful that common objects are only counted once. Otherwise, the sum of the sizes could overestimate the total memory footprint. Our algorithm guarantees this property by visiting each object only one time (see algorithm below for details), but this choice affects the measured sizes of the individual entries. For example, if two structures with roots $o1$ and $o2$ share a common object p , only one of their sizes will account for p – the one that is

traced first. In this case, o_2 will not consider any objects reachable from p . Let $o \setminus p$ represent the objects only reachable from o . Then the sizes of o_1 and o_2 are as follows:

$$size(o_1) = size(o_1 \setminus p) + size(p)$$

$$size(o_2) = size(o_2 \setminus p)$$

The sum, however, still accurately reflects how much memory is actually in use by all of the priority references together:

$$size(o_1) + size(o_2) = size(o_1 \setminus p) + size(p) + size(o_2 \setminus p)$$

The property above is crucial to our enforcement mechanism because it means we can trace a sequence of cache entries in any order, and the running sum of their sizes at any point represents how much memory would be occupied if all remaining entries were evicted.

To see why tracing the sequence in order is important, consider an extreme example in which three small entries, o_1 , o_2 , and o_3 , each of size K bytes alone, share a large common structure of size L bytes. Visiting the references in order will cause o_1 to have size $L + K$, and o_2 and o_3 to have size K . If the space bound is less than L it might be tempting to evict only o_1 . This choice will not achieve the expected space savings because o_2 and o_3 hold references to the shared state so only K bytes will be recovered. If we inspect them in order, though, we can see that the bound is reached during tracing of o_1 , and all three entries need to be evicted in order to get below target size L .

A similar issue exists for structures shared between priority spaces, although we consider this case to be more unusual. At each collection, the `PrioSpace` learns the total memory footprint of its `PrioReferences`. Consider the case where two `PrioSpaces` hold `PrioReferences` to the same data o with memory footprint of K bytes. At first, we may consider that the total memory footprint of both `PrioSpaces`

includes those K bytes. However, we use the marking bit to measure o , implying we can only measure o once per collection. While each collection processes all `PrioSpaces`, only the first `PrioSpace` to measure those K bytes and add it to their total memory footprint. Any other space will regard o as a structure in use elsewhere in the program and pass over it.

4.3.3 Collection Algorithm

The Prioritized Garbage Collector is built on a standard full-heap mark/sweep collector. The algorithms are amenable, however, to any *tracing* collector, including copying collectors and generational collectors. The reason that we focus on pure mark/sweep is that caches are highly non-generational data structures: none of the entries are short-lived, and under LRU, entries must sit in the cache for some time before they become the least-recently used and are evicted. In addition, this collector performs full-heap collections more frequently, so eviction policies built on it run more frequently.

Our collection algorithm is based on two key ideas:

- We reorder heap tracing so that we can visit specific regions of the heap graph based on their reachability – specifically, the regions reachable from priority references. No significant additional work is required, so this overhead is very low. This technique has been used by other systems to check heap properties using the garbage collector [AG09].
- We introduce *bounded marking*, in which the garbage collector traces a region until a condition is met (for example, the memory bound is reached); then it simply stops marking and nulls out potential dangling pointers. Unmarked objects are reclaimed immediately by the sweeper without being touched again.

The key to our algorithm is that it does not explicitly free low priority references to satisfy the memory bound; rather, it *protects* high priority references (by marking them) until the memory footprint grows to the bound. Once the bound is reached,

the algorithm ceases to mark any other objects in the priority space. With a small amount of fixup to avoid dangling pointers, the remaining low priority references will be reclaimed immediately by the sweeper. This approach *guarantees* that the memory bound will be respected (see reasoning below). A secondary benefit is that the collector does not need to touch the evicted objects, which might improve CPU cache performance, although we do not measure this effect here.

To ensure that the collector counts only objects reachable solely from the priority space, we reorder the phases of the collector as follows:

Phase (1): Compute exact target sizes (in bytes) for priority spaces that are specified in fractional terms. For example, if a priority space specifies its bound as 20% of the total heap, the collector uses computes $0.2 * \text{heapsize}$ as the bound.

Phase (2): Premark all `PrioReference` objects held by the `PrioSpace`.

Phase (3): Perform a complete transitive closure from the root references, marking unvisited objects during the search. This process will stop at each `PrioReference`, so the only objects that will remain unmarked are either garbage or are reachable *only* through a priority reference.

Phase (4): Revisit each `PrioReference` in priority order, from highest to lowest, and perform a transitive closure starting at its referent object. During this phase, the collector computes a running sum of the object sizes. If the total size hits the memory bound for the space then this phase ends immediately. Since it visits the priority references in order, the remaining unmarked instances must all be lower priority.

Phase (5): Any `PrioReferences` with unmarked referents are nulled out (the pointer to the referent is set to null).

Phase (6): If Phase (3) ended early, it could leave *part* of a data structure marked, with outgoing pointers to unmarked objects. To preserve memory safety

the collector also nulls out all potentially dangling pointers as well as the priority reference containing this partial structure – in effect, evicting the entire structure.

Phase (7): Evicted objects are garbage, and can be immediately reclaimed by the sweeper.

Notice that since the collector is doing the work, eviction only occurs at GC time. But this makes sense: we cannot assess the global memory situation until GC time, and we don't have an effective way to recycle memory in between GCs.

Partial eviction. Phase (5) of the algorithm handles the case in which bounded marking stops part-way through a data structure, leaving pointers from marked objects to unmarked objects. In order to preserve memory safety, this phase nulls out all of these references. In addition, we null out the `PrioReference` itself, which makes even the partially marked portion of the structure unreachable.

There is a case, however, in which a program could observe a partially evicted structure. We believe this case would be rare, however, since it only happens under very specific conditions. If the program creates a weak reference to an object in the cache, and that object is part of the marked portion of the partially evicted structure, then our JVM will preserve the weak reference and the object it points to. This weak reference will be cleared at the next GC, since the strong reference from the cache entry is now unreachable, but there is a brief period where the program could follow the weak reference and find a structure with null fields in unexpected places.

We have not been able to find a satisfactory and performant solution to this problem. If this problem manifests in a program, however, our implementation provides a switch that forces eviction only at cache entry boundaries – that is, when the space limit is reached, the mark phase continues until the current entry is completely marked. The downside of this option is that it allows the cache to grow beyond the size limit. In practice, however, we find that it only hurts performance

when cache entries are very large (100's of KB each) and keeping the extra objects is a substantial burden on memory resources.

4.3.4 Overheads

4.3.4.1 Runtime Overhead

Phase (3) of the algorithm above requires the collector to visit `PrioReference` objects in priority order. In our current implementation, the `PrioSpace` class keeps its priority references in a max heap, so that common operations are $O(\log N)$ time. This cost, however, is paid every time the priority of a reference changes: the priority space must re-insert it into the heap. It is possible that in some configurations the total cost of these inserts would exceed the cost of simply sorting the list immediately before garbage collection. In practice, we have not observed a significant performance penalty.

4.3.4.2 Space Overhead

Each `PrioSpace` stores its `PrioReferences` in a max heap, which is implemented as an array. Furthermore, we arrange the `PrioSpaces` into a linked list in the VM, using an extra reference in each `PrioSpace` to point to the next one. So, if we have N `PrioReferences` spread across K `PrioSpaces`, then the total space overhead is $N + K$ references, in addition to the space required for each `PrioReference` and `PrioSpace` instance.

4.3.5 SACHE: A Space-aware Cache

Using these mechanisms, we implemented a space-aware cache that we call a *SACHE*. The interface to the *SACHE* is essentially the same as the Guava LRU cache, and it can be used as a drop-in replacement. An overview of the *SACHE* class is shown in Figure 4.8.

As with a cache built around a `HashMap`, we can `put` key-value pairs in, `get` a value using a key, and `remove` a value using a key. The *SACHE* stores all

```

class Sacle <K,V> extends HashMap<K, PrioReference<V>> {
    protected long highest_priority;
    protected PrioSpace<V> priospace;

    public Sacle(long maxSize);

    public boolean put(K key, V value);
    public V get(K key);
    public V remove(K key);

    private void update();
}

```

Figure 4.8: Interface for Sacle space-aware cache

values in `PrioReference` objects, so that the collector can measure and evict them as necessary. The `Sacle` increments the `highest_priority` value as necessary to ensure that the most recently hit value has the highest priority. In this way, the `highest_priority` value acts as a clock to mark recently hit values. The methods work as follows:

Constructor: create an empty hash map and an empty `PrioSpace` with the given space bound `maxSize`. We also support a version that specifies the size as a fraction of available memory.

get(K): If there is an entry in the map for the given key, update the priority on its `PrioReference` to `highest_priority` (incrementing if necessary) and return the referent value. If not, return null.

put(K, V): Create a new `PrioReference` in the `Sacle`'s priority space with the value `V` as the referent and give it the highest priority. Store the pair of key `K` and priority reference in the hash map.

remove(K): If there is an entry in the map for the given key, remove it from the map and remove the corresponding `PrioReference`.

update: Periodically scan the hash map looking for entries that have null values in their `PrioReferences`, indicating that they were evicted by the garbage collector. This method runs when the program accesses the `Sacle` after a collection and does not interact with the `PrioSpace`. Since the collector already evicted the `PrioReferences` from the `PrioSpace`, we can complete this operation in $O(N)$,

where N is the number of entries in the SACHE.

It is possible to add collector support to remove an entry from the SACHE whenever the collector decides to evict its corresponding `PrioReference`. In particular, we can manipulate the SACHE's outgoing pointers directly. This would improve the performance of the `update` operation. However, we decided against this to provide a more general reference type versus a cache-specific reference type.

4.3.6 Adaptive Sizing

The SACHE is an efficient and effective bounded-size cache supported directly by the garbage collector. The problem remains, however, of how to choose its size. As we show in Section 4.4, choosing a fixed size, measured as a fraction of total bytes in the heap, yields good performance across a range of workloads for our simple key-value store. Many applications that use caching, however, are not just key-values stores – they have other computations going on that are competing for resources. For example, looking at the results we might choose a SACHE sized to occupy about half of the heap. If other parts of the program need the other half, however, the resulting memory pressure will cause massive GC overhead.

Our solution is to adaptively size the SACHE according to available memory. Our current policy is simple: at each garbage collection we choose a SACHE size that ensures a minimum amount of free memory (if possible). Other policies are certainly possible, and we discuss some of them in the future work section.

To ensure a free memory reserve of size R bytes, we need to know the size of the heap (H bytes), and the total live size of all the data *not* in the SACHE (L). Using this information, the maximum size of the SACHE is set to $H - (L + R)$.

We can efficiently recompute this value at every garbage collection, growing or shrinking the SACHE adaptively. Phase 2 of our algorithm visits all objects not in the SACHE, so we can augment it to also compute their total size, which is L . Before Phase 3 starts, we compute the target bound for the SACHE, and the algebraic properties of the bounding mechanism guarantee that evicted entries will leave at least R bytes free.

4.4 Results

In this section, we evaluate how prioritized garbage collection helps overcome the conflicting space-time tradeoffs of software caching and automatic memory management. Note that there is nothing particularly innovative about our core caching algorithm or data structure. The Sache is essentially just a hash map. The difference is in how it manages provisioning – specifically, how it manages the number of entries in the hash map. Given a *fixed* configuration, it performs almost identically to the Google Guava cache on a given workload. What we show here is that with support from the collector, the Sache can automatically *vary* its configuration to make the best use of available memory regardless of the characteristics of the workload. In addition, it can adapt online to changes in memory use or workload.

4.4.1 Experiments

We evaluate our system using two caching applications: a key-value store driven by the same synthetic workloads presented in Section 4.2, and a web caching implementation driven by real traces of web traffic. Since the Sache API is almost identical to the Guava LRU cache API, we can easily switch between cache implementations in each benchmark system. We show results for the following experiments:

- First, we repeat the experiments shown in Section 4.2 using a Sache configured with a range of fixed sizes (non-adaptive). Specifying the size in terms of memory footprint instead of number of entries is robust across a range of workloads.
- Second, we test the adaptive sizing algorithm for the Sache with a driver that dynamically increases and decreases memory demand. The algorithm can dynamically shrink or grow the Sache in response to changing memory demands in other parts of the program.
- Third, we test the two cache implementations under a real workload of web traffic. Software caches perform the work of the central data structure in a

web cache (something similar to `memcached`). This application is challenging to implement in a garbage-collected language, since the goal is to cache as much as possible.

4.4.2 Methodology

We implemented our GC mechanisms in JikesRVM version 3.1.2 as a modification to the stop-the-world mark-sweep collector. The Sacle itself is implemented at the application level. We have also incorporated these mechanisms into a generational collector, but it provides little benefit for cache-heavy applications because caches are so strongly non-generational. In addition, the generational collector delays computation of the size information and enforcement of the eviction policies, since these mechanisms cannot be implemented properly for partial collections.

Building and running. We build JikesRVM in the `FastAdaptive` configuration (for performance). The Guava experiments are run on an unmodified version of JikesRVM to avoid incurring any possible penalty related to prioritized GC. All experiments are run on a machine with dual 2.8GHz Xeon X5660 processors (X64) with a total of 12GB of main memory running Ubuntu Linux kernel 3.2.0.

We run each configuration only one time, but due to the length of the traces (tens of thousands) small fluctuations in the cost of any individual cache operation or garbage collection are averaged out. In addition, the performance differences we highlight are orders of magnitude greater than experimental noise, and in many cases it is the difference between running to completion or crashing with an out-of-memory error.

4.4.3 Non-adaptive Sacle

Figure 4.9 shows the performance of a Sacle under the same workloads as the Guava cache. We measure the size as a percentage of the heap instead of a maximum number of entries. Each graph has three curves: one for mutator time, one for GC time, and another for total time.

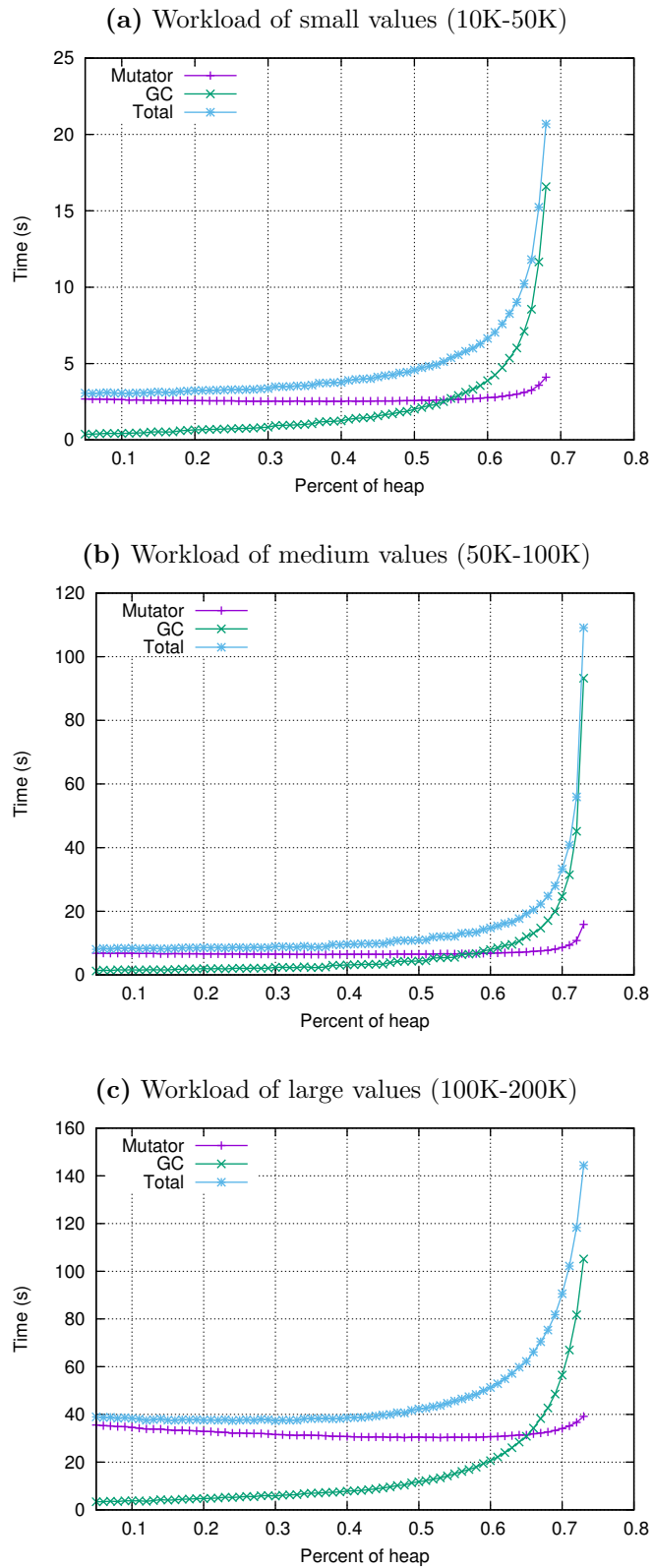


Figure 4.9: SACHE performance under three different workloads. While absolute performance varies, compared to the Guava cache the space-time tradeoff is relatively independent of the workload. The SACHE makes the best use of the available space.

- Figure 4.9(a) shows the performance on the trace with small-sized values. The total time is a bowl shape, but the minimum occurs early on at a limit 10% of the heap size. Afterwards, the mutator time remains steady until 65%.
- Figure 4.9(b) shows the performance on the trace with medium-sized values. The minimum of the bowl is not present on the graph. Just like the graph with small-sized values, the Sacle peaks heavily towards the end, dominated by garbage collection cost.
- Figure 4.9(c) shows the same graph for larger values. The Sacle continues to function even when using 75% of the heap. This comes at the cost of increased garbage collections.

All three graphs have the same shape. The mutator time is about the same until after 60% of the heap is reserved for the Sacle. Recall that the Sacle limit is only enforced at a collection. Therefore, the Sacle holds more values and more hits occur. The application does not rebuild values, lowering the mutator cost. On the other hand, the Guava caches have to rebuild many items at when we have a low limit on the number of entries.

Despite the Sacle using a lot of memory prior to GC, the GC curve starts low and peaks towards the end. The prioritized GC frees elements of the Sacle as soon as it observes the limit would be exceeded. This leaves much less garbage in the heap than simply evicting the value from the Sacle outside of GC.

Figure 4.10 compares the above results with the graphs for Guava in Section 4.2. To directly compare them, we plot the Sacle with the average number of entries after a garbage collection. Recall that prioritized GC only enforces the bound at a collection, so the Sacle actually grows larger in between collections. These numbers are approximately the memory bound of the Sacle divided by the average size of the values in the trace.

We see that the shapes of the curves between Guava and the Sacle are about the same. The key difference is that Sacle curves represent the *same set of configuration options* – 10% of the heap on the low end and 80% of the heap on

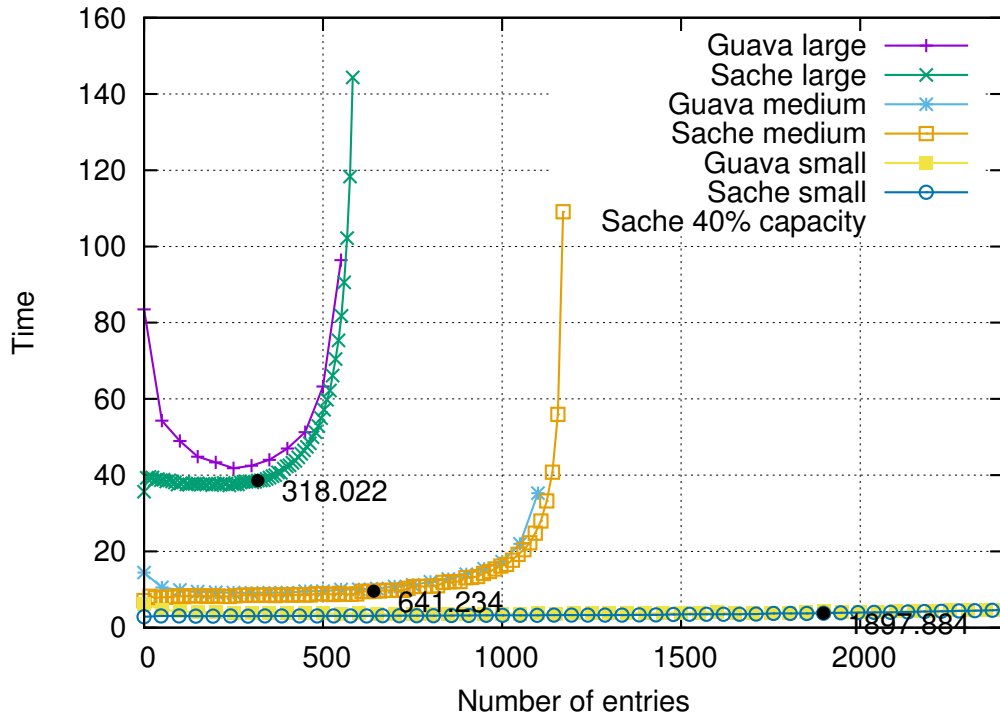


Figure 4.10: Comparing Sache and Guava LRU on three workloads: the performance is very similar, but the three Sache curves represent the same configuration choices. The highlighted points represent a Sache set to 40% of the heap, which easily accommodates all three workloads by using different numbers of entries.

the high end. A programmer could choose a Sache capacity of 40% and be able to achieve good performance regardless of the workload. The three labeled points on the graph show how this choice leads to different numbers of entries under the different workloads.

Looking at the small workload, the Guava’s curve is about flat while the Sache’s extends further out and eventually starts to move upward. The Sache utilizes as much memory as it can before the limit is enforced. This allows the lower end limits of 5% to perform better than the hard limit of 1500 entries on Guava.

Looking at the larger values, the Guava cache fits less than 600. The size limit on the Sache and prioritized GC allow the Sache to handle a mix of sizes. In particular, the collector frees values that the Sache will not keep because of the size limit. This also allows the Sache to hold more items than the Guava cache can, by prioritizing smaller values.

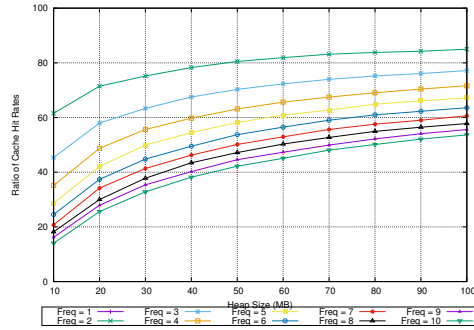
4.4.4 Multiple Caches

One of the problems with soft references is that they are managed by the JVM using a single global policy. Even if that policy happens to be the right one (e.g., LRU) treating all soft references as equal can lead to very bad performance. Consider, for example, a program with two caches. If one cache is accessed less often than the other then its entries will tend to appear towards the end of the global soft reference LRU queue. When memory is tight, many more of its entries will be reclaimed regardless of their value to the application (i.e., regardless of the miss cost).

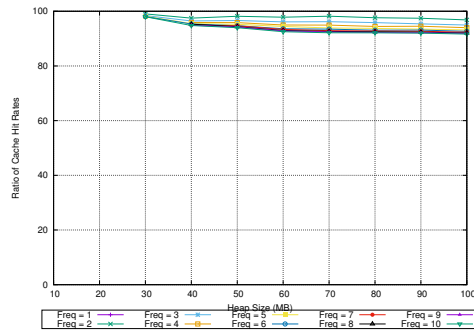
We measure this effect directly using the following experiment: we run two caches simultaneously and have both serve requests from our largest trace, but at different frequencies. One cache processes N requests for every request the other cache processes. We also use a larger heap size to measure the effect of having more memory available. We run this experiment on the HotSpot VM with Guava caches that wrap their cache values with soft references and neither an explicit size limit nor an eviction policy. We do this to measure the effectiveness of HotSpot's soft reference eviction policy alone. We also run the experiment on our modified VM with two Saches, each configured to use 20% of the heap. We use the LRU policy for each Sache's `PrioSpace` to match HotSpot's policy for removing soft references. Hit rates are reported as a number between 0 and 1. We report the difference between hit rates on the same scale. Figure 4.11 presents the results of this experiment.

Figure 4.11a shows the results for soft references running on HotSpot, which clearly reflect the global LRU policy. The X-axis shows the heap size; the Y-axis shows the ratio of the measured hit rate to the maximum hit rate. As the difference between the access frequencies of the two caches grows, the hit rate of the slower cache drops significantly. Its entries appear to be less valuable because they are less frequently accessed, so the soft reference eviction policy removes them first. The effect is more pronounced in smaller heaps because the soft reference policy is more aggressive. In the worst case (10-to-1 frequency difference), the hit rate is only 1/4 of its potential, but the degradation at just a 2-to-1 difference is very significant as

(a) Guava Caches with Soft References



(b) Saches with PrioReferences



(c) Maximum Hit Rate of the Caches

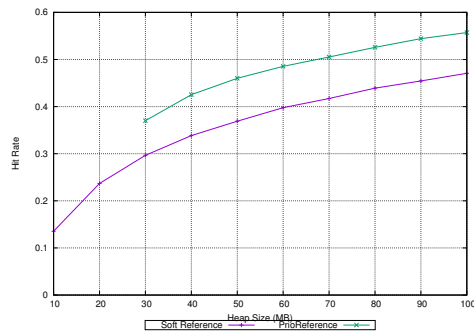


Figure 4.11: Hit rates can drop dramatically when soft references are used for two caches working at different frequencies. Prioritized garbage collection keeps the hit rates of both caches relatively close by managing their resources separately.

well.

Figure 4.11b shows the same results for prioritized garbage collection. The hit rates of the two Saches differ by at most 5% because the PrioSpaces manage their references separately, so the VM does not clear the PrioReferences in the less frequently used cache regardless of what is going on in the higher frequency cache.

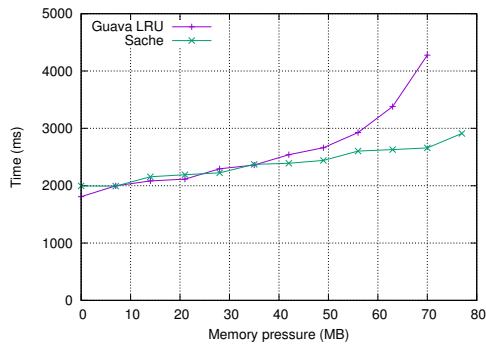
Finally, Figure 4.11c shows the maximum hit rate (in absolute terms) for both the Guava cache with soft references and the Sache with PrioReferences. As expected, with no competing memory demands, the two systems perform almost identically. Note that for Figures 4.11b and 4.11c, the Sache does not have data for 10MB and 20MB. Since VM objects share heap space with Java applications in JikesRVM, we needed 30MB to start running the experiment without running out of memory.

4.4.5 Adaptive Sache

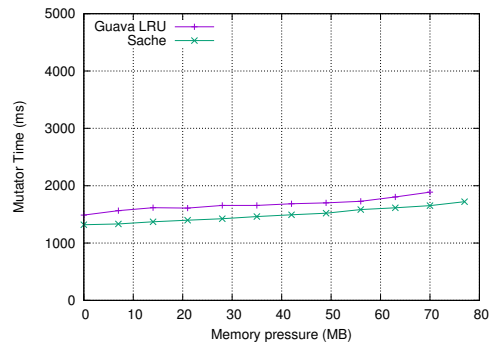
The purpose of the adaptive sizing algorithm is to allow the cache to respond to changes in the available resources. Our goal is to prevent the cache from competing with other application data structures, causing unnecessary memory pressure.

For these experiments, we modified our benchmark to build a separate large data structure that grows as the trace is processed. Each experiment is divided into three phases: during the first 1/3 of the trace, no extra memory is used; during the middle 1/3, the program starts growing the non-cache data structure, consuming more and more memory; during the last 1/3 of the trace, the program slowly dismantles the structure, allowing the collector to alleviate the pressure.

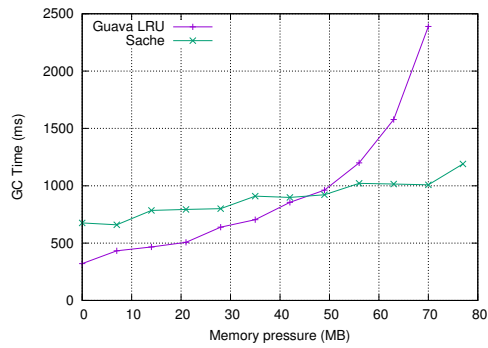
We ran our medium-sized-objects trace through the key-value store using both the Sache and Guava LRU cache. For the Sache, the adaptive algorithm is configured to target a 50% memory reserve. This value corresponds to a heap two times the live size, which is a good target for performance [HB05]. We size the Guava cache using the data collected in Figure 4.4(b): the best size for this workload appears to be around 350 entries. We fix the heap at 115MB, as in the earlier experiments.



(a) Total time



(b) Mutator time



(c) GC time

Figure 4.12: Performance of Sacle vs Guava LRU cache under increasing memory pressure: our adaptive sizing algorithm shrinks the Sacle to avoid triggering massive GC overhead. At 77MB, the application with the Guava LRU cache crashes.

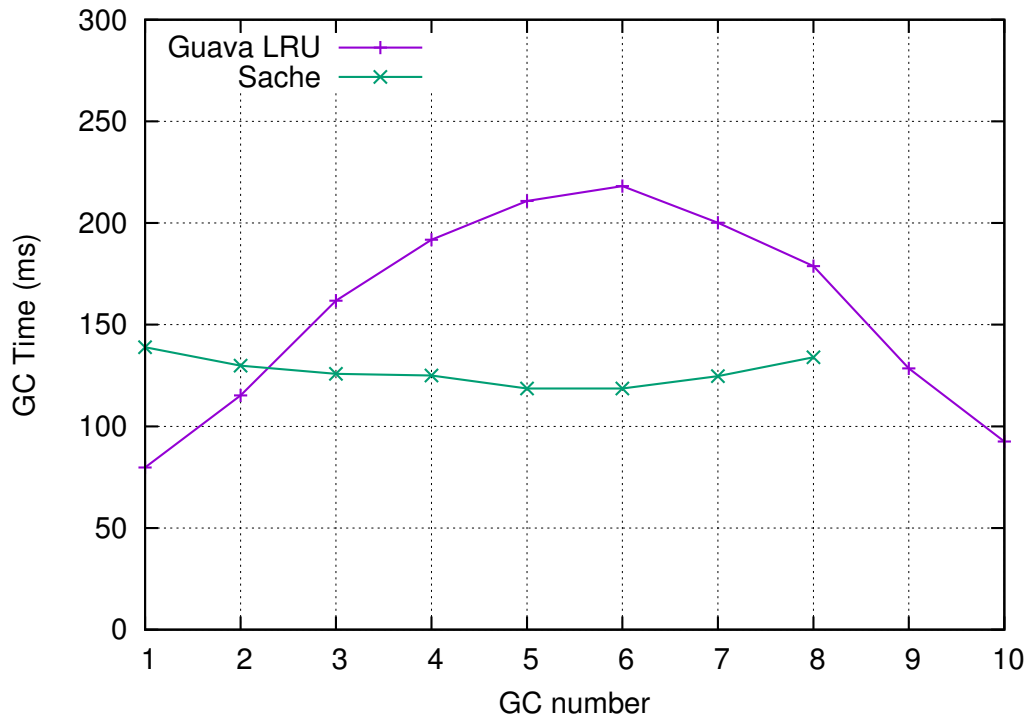


Figure 4.13: GC time over a single run: without the ability to adapt, cache and non-cache structure compete, triggering costly GCs.

Figure 4.12 shows the total time, mutator time, and GC time. Unsurprisingly, the Sacle and the Guava cache exhibit similar performance as long as memory is plentiful. As memory pressure increases, however, the Guava cache competes with non-cache structures in memory, and GC costs skyrocket. When memory pressure exceeds 77MB, the Guava implementation crashes. The Sacle automatically shrinks to ensure sufficient free memory, resulting in a smooth curve and no crashes.

Figure 4.13 provides some insight into this behavior. It shows the GC time for each collection during a single run of the benchmark. As expected, in the Guava implementation, once the non-cache structure begins to grow, each GC becomes much more expensive. In addition, memory scarcity triggers more frequent GCs. With the adaptive Sacle, the GC time is flat: the algorithm guarantees that the cache will not cause the live size to exceed the target reserve.

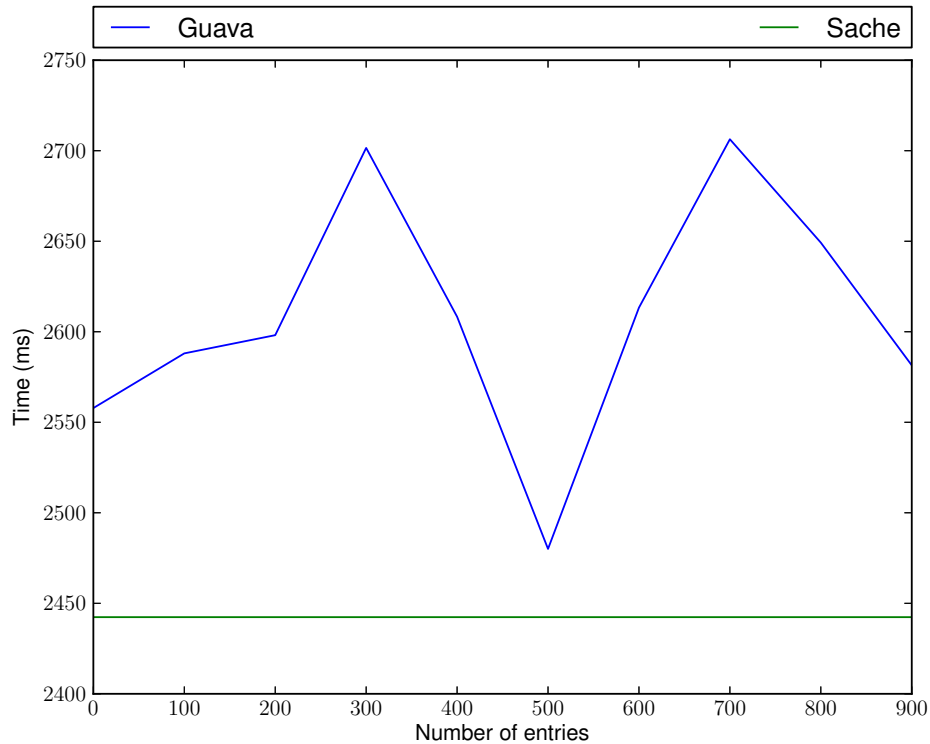


Figure 4.14: Performance of the Sacle and Guava cache on real web traffic traces across a range of cache sizes.

4.4.6 Web Caching Workload

The heart of a web caching application, such as `memcached`, is a key-value cache like the ones we describe here. We adopted the techniques for testing web caches and applied them to our cache implementations. We use the BU 272 trace, a record of real web traffic, to drive the caches, and measure performance as above[CBC95]. It consists of 15K entries requesting a total of 72MB of web data. Figure 4.14 shows the performance of the Guava cache across a range of numbers of entries. The Sacle is a flat line, since it chooses its own size.

4.5 Related Work

4.5.1 Reference Types

Hayes introduced Ephemérons to determine unreachable objects in key-property lists instead of using a list with a weak key and strong value [Hay97]. This capability allows entries with dead keys to be removed from the list and properly deleted. In contrast, eviction from the Sacle depends on the value.

4.5.2 Programs Acting on Resource Limits

There has been considerable work on caching for web traffic. The Greedy-Dual algorithm takes into account the amount of time to obtain a page as well as the size of the page to determine its eviction [CI97]. The cached objects are text documents, so their size can be easily measured. Our system allows these kinds of eviction policies to be used in general software caches, where measuring the size of a cached structure is non-trivial.

Yang and Mazières describe resource containers for Haskell that bound the memory usage of untrusted code [YM14]. Exceeding the limit of the container kills the accompanying thread. A Sacle is a more general structure which allows the program to choose what action to take when memory bounds are exceeded. In principle, it could also be used to implement a similar security policy.

Czajkowski and von Eicken introduced an interface for programmers to monitor the resources used by threads in a Java program [CvE98]. Furthermore, they allow programmers to implement their own reaction to threads exceeding their resource limits. They use bytecode rewriting to track heap memory usage for each thread. In contrast, we track a list of known objects and use the garbage collector to both track memory usage and enforce the limits. In addition, our technique uses general heap reachability to define bounded structures, rather than thread ownership.

JAMM uses JVMTI to traverse the heap and compute the size of data structures [Bel]. This approach is flexible and powerful, but very slow. A large data

structure can require seconds of runtime to size (according to the documentation). By piggybacking on the garbage collector, we can perform the same measurement with almost no performance overhead. The tradeoff, however, is that we cannot compute sizes at arbitrary points during execution.

Price, Rudys, and Wallach divide a process into tasks and used the garbage collector to track how much memory is attributed to each task [PRW03]. Our work expands on this by allowing arbitrary data structures to be tracked and by providing a way to enforce a size limit.

4.5.3 Using GC to Assist Running Programs

O’Neill and Burton presented simplifiers as a way to improve the performance of a program [OB06]. Objects can add a `simplify()` method that the garbage collector invokes when the collector traces over it. The Data Structure Aware Garbage Collector lets the program denote which objects are internal nodes for data structures [CP15]. It uses this information to improve garbage collection for these structures and therefore overall performance. Our work uses the GC to allow programs to run when they would run out of memory in normal execution. Furthermore, we can traverse and modify the structures without editing those structures’ code.

4.6 Conclusions

This chapter presents a new approach to managing the conflicting tradeoffs between software caching and garbage collection. The key to our approach is widening the interface between the application and the runtime system in order to increase cooperation.

Chapter 5

Deferred Garbage Collection

5.1 Introduction

Long-lived objects slow down garbage collectors. To explore this, consider the PageRank algorithm. In order to properly rank a given page, the algorithm needs to keep a graph of web links in memory, which is generated by input data. This graph remains in memory for the duration of the program. However, this graph slows down collectors. Each time the collector is triggered, the collector will mark each object in the graph as live, even if the graph has not changed. This repetitive work leads to long GC pauses and an overall slower application. The garbage collector can avoid this work if it was aware of those long-lived objects. If the collector was aware, then it can process only the objects of the graph that have been edited and not visit the unedited objects. In this case, we say the unedited objects have been *deferred* on by the collector. By deferring, the collector visits less objects during one collection, which speeds up that collection.

The most common deferral tactic is to segregate the object by age with a generational garbage collector. However, generational garbage collectors can still run major collections which must traverse the whole heap. If that heap's old and young generations are constantly full, then major collections will be more frequent and more expensive.

These long and frequent collections can be reduced with *pretenuring*. Pre-

tenuring is the act of allocating an object into the old generation or some other rarely visited region of the heap. These objects can be chosen automatically through profiling at runtime [UJ92, SZ98, Har00] or profiling at build time [BSH⁺01]. All of these works consider each object or allocation site separately. Furthermore, they have a discrete measure of lifetime, placing objects into specific bins, regardless of their relation to each other. Objects that just barely miss the cut to pretenure are still subject to the frequent collections.

These objects can be identified manually using programmer annotations. In prior works, annotations are added to long-lived objects or their allocation sites at compile time and then the objects are pretenured to rarely visited regions at runtime. [BOF17, NWB⁺15, NFX⁺16]. However, they rely on the programmer to successfully annotate all relevant allocation sites in the application. This annotation burden can be alleviated by exploiting the connectivity of these long-lived objects. Instead of allocating all objects that create a long-lived data structure, those objects can be found dynamically by exploiting their connectivity.

In this chapter, we propose a new cooperative garbage collection algorithm, the *deferred garbage collector*. This algorithm depends on programmer annotated objects. Programmers identify long-lived objects by annotating the root of data structures. These data structures are then used to identify new deferred objects by exploiting the connectivity of these marked roots.

This work provides the following contributions:

- We propose the deferred garbage collector as a modification of the mark-compact collector.
- We explore how to instrument a mark-compact collector to estimate the savings and costs of the deferred garbage collector.
- We present the results of deferring on a caching microbenchmark, using an instrumented mark-compact collector to gather the data.

5.2 Proposing a Deferred Collector

In this section, we propose a collector that permanently defers on objects. The goal of the collector is to improve the performance of each collection by reducing the amount of time spent operating on long-lived objects.

5.2.1 DeferRef

Just like prioritized garbage collection in Chapter 4, deferred collection is a cooperative technique between the application code and the garbage collection. The API is also modified after Java reference types. The API in deferred collection consists of the DeferRef. The class definition of DeferRef is shown in Figure 5.1.

A DeferRef is the communication point between the deferred collector and the application code. In the application code, the programmer identifies an object or data structure that should be deferred. The programmer then wraps the object or data structure root around this reference type.

```
class DeferRef<T> {
    private T obj;

    DeferRef(T obj);

    // -- Get the referent
    public T get();
}
```

Figure 5.1: API for the DeferRef type.

5.2.2 Deferred Garbage Collection Algorithm

The deferred garbage collection algorithm is a modified version of the mark-compact collection algorithm. In this collector, DeferRefs and all objects reachable from them are deferred. This is *dynamic deferral*. With dynamic deferral, the deferred collector can mark objects to be deferred in the same way mark-compact collectors can mark objects as live.

The deferred collector performs the following phases:

Phase (1): First, the collector marks all newly found defer roots as deferred.

Phase (2): The collector performs a modified mark phase, a transitive closure from the traditional roots. If the collector encounters a deferred object, marking stops.

Phase (3): The collector performs the defer phase, another transitive closure from defer roots. The collector marks objects as deferred instead of live and does not stop if an object was marked as live in the mark phase.

Phase (4): The collector performs a modified forwarding phase. If object was deferred and not in a deferred region, they are forwarded into the deferred region. Objects in deferred regions are not visited in this phase and therefore do not get a forwarding address.

Phase (5): The collector adjusts the pointers to all objects, including deferred objects. If a deferred object points to a non-deferred live object, the deferred object's pointer must be updated for correctness.

Phase (6): Finally, the collector performs a modified compact phase. Newly deferred objects are moved to the deferred region. Objects already in the deferred region are not visited and therefore not moved.

5.2.3 Implementation Challenges

Implementing any garbage collection algorithm correctly is challenging and takes many person-hours. Deferred collection is no exception and in fact has its own specific set of challenges to overcome. Each challenge is discussed separately for the remainder of this section.

Moving objects to deferred region quickly The collector now needs to move a object to one of two places: the regular heap or the deferred region. At first glance, the collector could perform the forwarding phase twice, once for regular forwarding and another for forwarding deferred objects. However, the second phase can be just as costly as the first. Just like the first pass, the collector must visit the whole

heap to find the newly deferred objects. Implementers must interweave forwarding regular and deferred objects.

Deferral during minor collection This is only an issue with generational collectors. Full collections visit the whole heap, so the collector design can work easily. However, young collections visit a subset of the heap. If an object in that subset is to be deferred, then all objects reachable must be deferred as well. If those objects are actually in the old generation, the collector must traverse the boundary to properly defer them. This is counter to the mechanisms of most collectors, which stop processing objects if they are in the old generation.

Finding new deferred roots Over the course of an application's lifetime, deferred objects can be modified. A deferred object might suddenly point to a non-deferred object. This non-deferred object will need to be deferred in the next collection and to do so, the deferred object must be considered a deferred root. These deferred roots can be tracked with a remembered set. Remembered sets are a perfect fit since they track pointers between different regions of the heap. However, the size of this remembered set impacts the defer step in the collector. The larger the set, the more objects that can be moved and the longer the collection. If the set continues to be large over the course of the program, this can have a negative performance impact.

Retention The collector is designed to defer on long-lived objects. However, long-lived objects can eventually die. If those objects die after being deferred, they will remain in the defer region as garbage for the rest of the application's run. These objects are labeled as *retention*. Retention can cause more frequent collections. The cost of those extra collections can eclipse the savings of deferral.

Ricci proposed periodically performing an unmodified full GC [Ric16]. In other words, allow the collector to ignore the deferral property on all objects and collect the heap as normal.

5.3 Collecting the Benefits and Costs

Implementing the proposed collector will be difficult. This difficulty comes from introducing subtle bugs into the program. Before any programmer implements the collector, they must first know how the collector will affect a running application. In particular, we want to know how much time is saved by deferring objects and how much time is lost to retention. This information can be found by simulating deferred collections in an existing garbage collector.

The rest of this section explores how to instrument a mark-compact collector to collect this data. The collector is not changed semantically, so the collector is still memory-safe while gathering the relevant data.

5.3.1 Collecting Data at the Object Level

To start, we need to differentiate between deferred and non-deferred objects. To detect if an object is deferred, the collector stores additional metadata per object. The metadata keeps track of which objects are deferred. To ensure all deferred objects are found, this deferred objects are not ignored at any point in the collector.

Deferred metadata is calculated during and after the mark phase of the mark-compact collector. The mark-compact collector is modified to perform Phase 3 of our proposed deferred collector right after its mark phase. Recall Phase 3 finds all objects reachable from a set of defer roots premarked in Phase 1. Phase 1 assumes all the defer roots are known and knowing these roots can mean modifying other parts of the runtime system. Instead, the mark-compact collector's mark phase is modified to find these defer roots. The defer roots are precisely the `DeferRef` objects introduced by the programmer. These objects are recorded as deferred and set aside as a defer root.

5.3.2 Measuring Savings

- Measure time spent marking deferred objects (A) - Measure time spent compacting deferred objects (A)

Now that the deferred objects are known during each collection, the savings of deferred collection can be measured. Time is saved in deferred collection by not visiting deferred objects during collection. Therefore, the amount of time spent visiting deferred objects is measured in the modified mark-compact collector. The modified collector focuses on the mark and compact phases. Measuring the savings in the mark phase is as straightforward as measuring the newly added Phase 3 of the deferred collector. Measuring the savings in the compact phase is not as straightforward

5.3.3 Retention

As discussed in Section 5.2.3, retention needs to be considered in any deferred collection implementation. Retention can cause extra collections and possibly cleared. Therefore, retention after each collection must be measured. However, the mark-compact collector does not actually suffer from this retention.

This instrumented collector does not suffer from retention because it does not actually defer on objects it marks for deferral. The objects marked for deferral become garbage when the collector detects they are unreachable. This garbage has a specific trait: the object is marked for deferral, but not marked as live. To find these objects, the collector adds a phase right before the forwarding phase, where objects have been marked accordingly, but the heap has not yet been reorganized. This new phase visits each object of the heap and checks for that trait. If found, the size of that object is output to a file. The accumulated count of the size of all dead deferred objects is exactly the amount of retention from a deferred collection.

5.3.4 Disabling Deferral

Retention needs to be cleared and so the effects of different retention clearing strategies must be measured. One strategy is periodically ignoring deferral. The effects of this strategy on retention can be simulated by post-processing the retention information gathered earlier. The amount of retention is periodically reset. This reset simulates a non-deferred collection removing all the retention up to that point.

5.3.5 Instrumenting a Collector

The data was collected by instrumenting the Garbage-First Garbage Collector, shortened to G1 [DFHP04]. G1 is a generational mark-compact garbage collector that splits the heap into contiguous sized spaces called *regions*. G1 can choose which regions to visit. This means G1 can choose not to visit our deferred regions, making it a preferred target to modify into our deferred collector. For this reason, G1 was instrumented to collect our data.

When instrumenting G1, we focused on measuring the savings during full collections. Full collections must traverse the whole heap and can be costly with large heaps with a large number of live objects. If those objects are also long-lived, deferring on those objects can save time in costly collections.

To collect the object level data, we added two header words to every object, a *count word* and *deferral word*. The *deferral word* is set if the object is deferred and unset otherwise. The *count word* is split into two halves. The lower half of the count word tracks the number of times the object has been marked throughout its lifetime. The upper half of the count word tracks the number of times the object has been marked since it was deferred.

Timing compaction in G1 requires some ingenuity. When G1 performs a full collection, objects are copied one at a time and in order of address. This means for an object o to be properly copied, every object in all addresses before o must also be copied to ensure the data at o 's new address can be safely overwritten. By that same reason, we cannot just replace o 's copy with a no-op (i.e. copy into its old address for a pass) as o 's old address can be the target for another object's copy. Therefore, we must measure the total cost of compacting both non-deferred and deferred objects in one pass.

To measure the total cost of compacting non-deferred objects and the total cost of compacting deferred objects, we chose to measure the cost of compacting each object. The time for compacting *each* object is reported. In a post-processing pass, we accumulate these times to calculate the total costs of compacting non-deferred

and deferred objects.

5.4 Experiments

In this section, we estimate the savings and costs of a deferred collector. We verify that the collector can provide savings and those savings increase proportionally with the amount of deferred data. However, retention also increases and must be cleared throughout the program run.

5.4.1 Experiment

We run the singular key-value store application from Section 4.4. The application is a cache driven by the synthetic workload with medium values from Section 4.2 under a fixed heap size. In the application, the root of the cache data structure is deferred on, reflecting how we expect programmers to annotate their structures. This experiment is repeated on a range of fixed sizes to evaluate how costs and savings change when larger structures are deferred on.

5.4.2 Methodology

We implemented the instrumentation on the G1 collector in OpenJDK and built under the slowdebug configuration. We chose this configuration to ensure the new object header was properly cleared after compaction.

All experiments were run on a machine with dual 2.8GHz Xeon X5660 processors (X64) with a total of 12GB of main memory running Arch Linux version 5.3.1. We used `-XX:-UncompressedOops` option and set the maximum heap size to 320 MB to ensure the same failure rate as the example in Chapter 4. The increased heap size is to accommodate the increased size of the header every object.

5.4.3 Savings on Marking and Compacting

Figure 5.2 shows the potential savings on skipping marking and compacting deferred objects under an increasing number of maximum cache entries. Unsurprisingly, the

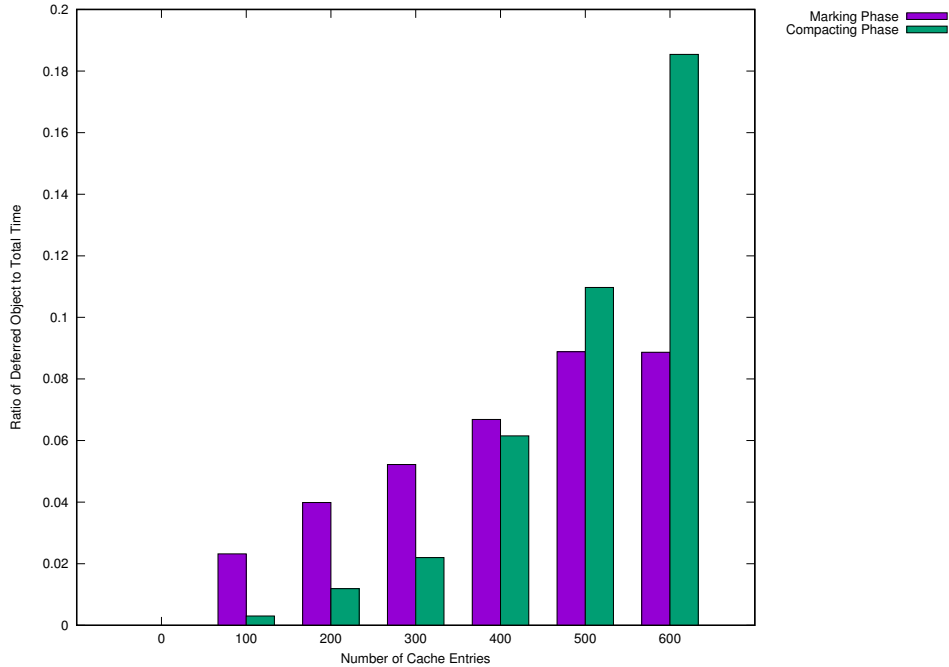


Figure 5.2: Ratio of time spent marking and compacting deferred objects under different cache sizes. As the cache size increases, so does the number of objects we defer on. This results in a longer compaction times for deferred objects.

savings on compacting objects increases as the cache limit does. The cache stores more values, each of which the GC defers. However, the marking costs plateau at about 10% of the total marking time. The results suggest that most of the savings will come from not compacting instead of not marking.

5.4.4 Cost of Retention

Next, we wish to estimate the cost of retention added by deferred collections. Figure 5.3 shows the potential retention in this experiment. The y-axis now represents the maximum number of megabytes taken up by retention. Retention increases to 84MB when the cache is configured to hold a maximum of 600 entries. In that particular run, this retention accounts for over a fourth of our heap. This result supports our hypothesis that this deferral collector will need to clear the retention occasionally.

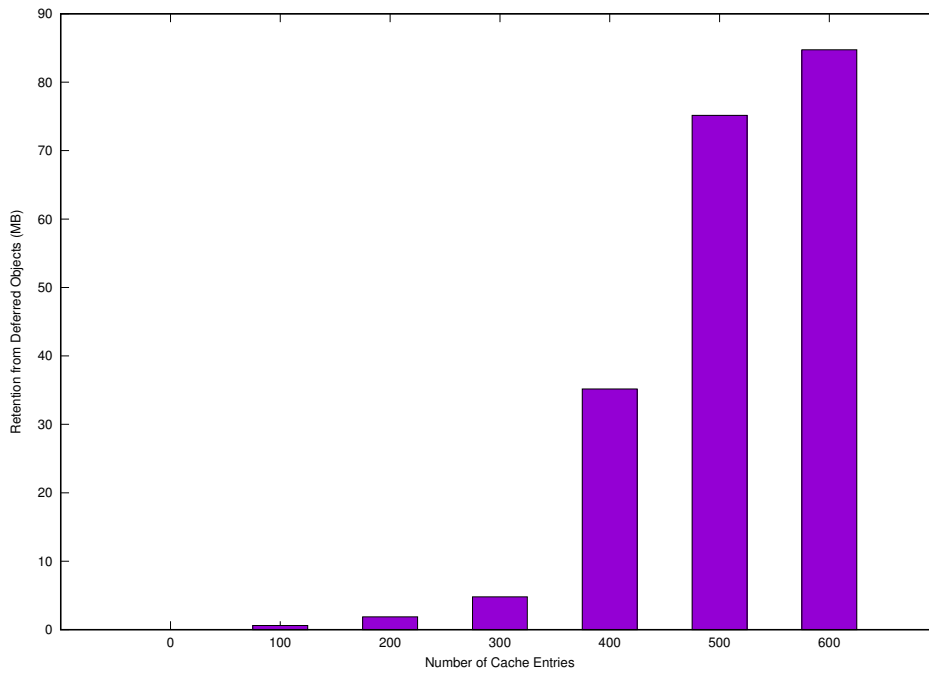


Figure 5.3: Amount of memory retained from deferral under different cache sizes.

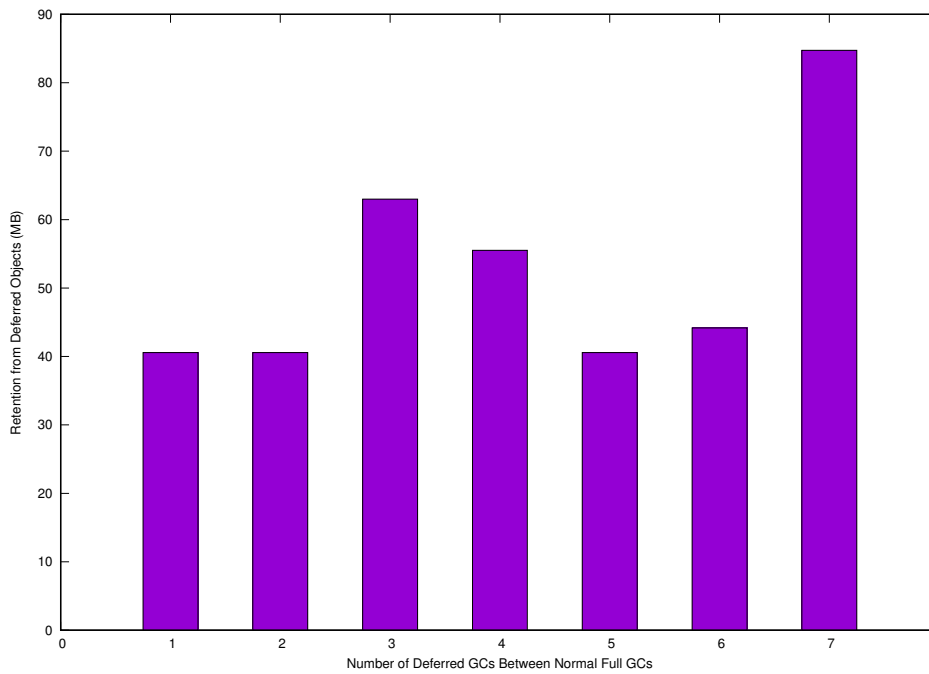


Figure 5.4: Amount of memory retained from deferral, varying the number of deferred GCs.

5.4.5 Benefit of Ignoring Deferral

In Section 5.2.3, non-deferred collections run between deferred collections were proposed to clear the retention. For this experiment, the period between non-deferred collections was set to a fixed number of collection, k . k was varied from 0 to the total number of collections in the run. A value of 0 means no deferred collections occurred. A value equal to the total number of collections means only deferred collections occurred.

The results of this simulation are shown in Figure 5.4. The x-axis represents how many deferred collections occur before an unmodified collection. The y-axis represents the maximum retention on the heap. Of note, the first entry represents a GC that never defers and the final entry represents a GC never clears retention. In most cases, the retention is cut to a little over 40 MB. In the cases where 3 or 4 deferred GCs occur prior to an unmodified GC, this strategy fails to clear retention from two deferred GCs at the tail-end of the program. These two deferred GCs contribute to most of the 84MB retention seen earlier.

5.5 Related Work

5.5.1 Automatic Pretenuring

Pretenuring is a popular way to avoid frequent collections caused by long-lived objects. Ungar et. al. focused on the survivor rate of collections to determine if young objects should be pretenured to the old generation [UJ92]. Seidl et. al. focused on creating custom pretenuring allocators created by profiling applications [SZ98]. Harris focused on sampling objects in the heap and monitoring them and their allocation sites to find pretenuring targets [Har00]. These works considers only particular objects for pretenuring. In contrast, deferred garbage collection considers an object and every object reachable from it for deferral.

Blackburn et. al classified objects as having short, long, or immortal lifetimes and pretenuring the old and immortal objects [BSH⁺01]. Immortal objects are

objects that survive until the end of the application. In deferred collection, deferred objects are made immortal manually. This allows objects to transcend the bins, allowing very long objects to be considered immortal.

5.5.2 Heap Subdivision

Subdividing the heap is not a new paradigm in garbage collection. Generational garbage collection, was one of the first to break the heap apart [Ung84]. G1 took this a step further and subdivided the heap into regions while keeping the generational distinctions [DFHP04]. Dynamic deferral is a logical extension of these ideas, but uses programmer intuition instead of program lifetime to decide what and when to promote objects.

Clustered Collection defers on objects as well identifying clusters of rarely mutated objects before full collections [CM15]. The clusters are re-identified as necessary. This re-clustering can harm the performance of applications that change phases instead of remaining in a steady-state. Our exploration touches on the cost of deferring on mutating objects in a single deferred cluster.

Write-rationing garbage collection automatically identifies objects that are not written to as often and moves them over to non-volatile memory [ASME18]. It observes writes in the mature space via the write barrier and moves these objects from DRAM to non-volatile memory and vice-versa on full collections. This exploration focuses solely on heaps in DRAM.

5.5.3 Programmer Hints

NG2C allows programmers to create their own generations, allowing objects of similar lifetimes to die together [BOF17]. N2GC uses scoping to determine the generations thanks to the use of a `new` keyword. Yak allows programmers to wrap program paths under epochs, which allocate all objects to a data space region [NFX⁺16]. We explore dynamic deferral, which allows for data structure mutated outside of a single static scope to be deferred.

5.5.4 Connectivity

Others have also taken advantage of connectivity to improve garbage collection. In particular, connectivity-based GC proposed promotion through connected objects. Data-structure aware garbage collection moves data structures to promote spatial locality [CP15]. However, they require the data structure internals to be annotated by hand. We explore using the root of a data structure to propagate deferral. By using only the root, programmers can avoid annotating already existing libraries.

Connectivity-Based Garbage Collection (or CBGC) exploits object connectivity to determine what objects should die together and in doing so, eliminates the need for a write barrier [HDH03]. They use a static analysis to partition the heap. These partitions are then monitored and collected by CBGC. A collection algorithm based on this exploration will rely on programmer hints to perform well.

5.6 Conclusion

In this chapter, we explored how dynamic deferral can benefit G1. With deferral, G1 can reduce the compaction phase of full GCs in our cache microbenchmark by up to 20% under memory pressure. While retention constituted up to 25% of our heap in the worst case, we saw adding unmodified full GCs can cut that retention in half.

Chapter 6

Conclusions and Future Work

We described three works that use programmer given information to improve garbage collection behavior. Representative inputs allow Autobahn to place strictness annotations to improve program runtimes. Eviction policies allow prioritized GC to bound the memory usage of caches to reduce memory pressure. Programmer annotations on long-lived data structures allow deferred collection to reduce time spent in garbage collection. We conclude by looking into future work for each project presented.

6.0.1 Autobahn

Autobahn produced numerous strictness annotations. Many times, the annotations were overly numerous, too many for a user to digest the consequences of. Sun et. al. improved Autobahn by removing some annotations that did not produce a tangible benefit to the program [SF18]. An avenue for future work is to continue to reduce the number of annotations, from more post-processing of the annotation to using more sophisticated search algorithms. Another avenue is to automatically determine what inputs do not trigger undefined behavior in Autobahn-optimized programs. One way to find those inputs is to translate the annotations inferred by Autobahn into StrictCheck [FZL18] specifications and process the results returned by StrictCheck.

6.0.2 Prioritized Garbage Collection

Prioritized garbage collection’s adaptive sizing algorithm is rudimentary, relying on a simple heuristic. A more sophisticated heuristic can be created by introducing statistics like the current miss and hit rates of the SACHE. This new information will allow the prioritized garbage collector to better determine how to resize not just one SACHE, but multiple SACHES. Furthermore, not every eviction policy a programmer can write is safe to run during a garbage collection. For instance, no policy that allocates objects should be run. Finally, the current API can be more expressive. The API does not make obvious how to assign an object a particular score for the collector to properly prioritize them. To better understand the needed level of expressiveness, a proper user study testing the expressiveness of the current API is required. With the results of that study, we can craft a domain-specific language for `PrioSpace` eviction policies. This language can help lower the difficulty of writing complex high-level policies and allow for more use of the Prioritized GC’s functionality outside of caching.

6.0.3 Deferred Garbage Collection

We presented a limited study of deferred garbage collection. This study can be expanded to properly evaluate the collection algorithm on real world programs. Similar algorithms were evaluated by modifying big data programs written in frameworks like GraphChi [KBG12] and Hyracks [BCG⁺11] and running applications written in those frameworks.

We presented methods to estimate the savings and some of the costs of using deferral garbage collection, but there are more costs to consider, especially with retention. Retention can lead to more collections. These extra collections can override the savings from deferral. While we explored adding unmodified major collections to reduce retention, we need a way to approximate how much time these collections can save. Measuring these savings accurately will require that the retention be present and therefore an implementation of dynamic deferral. The exploration

only looked at propagating deferrals during major collections. The next step in the investigation is propagating deferrals during minor collections. Another step forward is implementing garbage collection using G1 as a starting point.

Bibliography

- [AG09] Edward E. Aftandilian and Samuel Z. Guyer. GC assertions: Using the garbage collector to check heap properties. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–244. ACM, 2009.
- [ASME18] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. *SIGPLAN Not.*, 53(4):6277, June 2018.
- [AXF⁺12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.
- [Ban16] Bang Patterns. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/bang-patterns.html, 2016.
- [BCG⁺11] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE 11, page 11511162, USA, 2011. IEEE Computer Society.
- [BCM04] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceed-*

ings of the International Conference on Measurements and Modeling of Computer Systems, pages 25–36, 2004.

- [Bel] Jonathan Bellis. Jamm. <https://github.com/jbellis/jamm>.
- [BGH⁺06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.
- [BOF17] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. Ng2c: Pretenuring garbage collection with dynamic generations for hotspot big data applications. *SIGPLAN Not.*, 52(9):213, June 2017.
- [Bro15] Niklas Broberg. The haskell-src-exts package. <https://hackage.haskell.org/package/haskell-src-exts-1.17.1>, 2015.
- [Bry16] Bryan O’Sullivan. The Aeson Package. <https://hackage.haskell.org/package/aeson>, 2016.
- [BSH⁺01] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 01, page 342352, New York, NY, USA, 2001. Association for Computing Machinery.

- [CBC95] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of WWW client-based traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, Boston, MA, USA, 1995.
- [CF14] Stephen Chang and Matthias Felleisen. Profiling for laziness. POPL '14, 2014.
- [CI97] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 18–18, 1997.
- [CM15] Cody Cutler and Robert Morris. Reducing pause times with clustered collection. *SIGPLAN Not.*, 50(11):131142, June 2015.
- [CP15] Nachshon Cohen and Erez Petrank. Data structure aware garbage collector. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015*, pages 28–40, New York, NY, USA, 2015. ACM.
- [CPJ85] Chris Clack and Simon L. Peyton Jones. Strictness analysis—a practical approach. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 35–49, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
- [CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '98*, pages 21–35, 1998.
- [DFHP04] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management, ISMM '04*, pages 37–48, New York, NY, USA, 2004. ACM.

- [EP03] Robert Ennals and Simon Peyton Jones. Optimistic evaluation: An adaptive evaluation strategy for non-strict programs. ICFP '03, 2003.
- [Fax00] Karl-Filip Faxén. Cheap eagerness: Speculative evaluation in a lazy functional language. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 150–161, New York, NY, USA, 2000. ACM.
- [FZL18] Kenneth Foner, Hengchu Zhang, and Leonidas Lampropoulos. Keep your laziness in check. *Proc. ACM Program. Lang.*, 2(ICFP):102:1–102:30, July 2018.
- [ghc15] Profiling —Glasgow Haskell Compiler 8.10.1 User’s Guide. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/profiling.html, 2015.
- [Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.
- [Har00] Timothy L. Harris. Dynamic adaptive pre-tenuring. *SIGPLAN Not.*, 36(1):127136, October 2000.
- [Hay97] Barry Hayes. Ephemérons: A new finalization mechanism. *SIGPLAN Not.*, 32(10):176–183, October 1997.
- [HB05] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 313–326, New York, NY, USA, 2005. ACM.
- [HDH03] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. *SIGPLAN Not.*, 38(11):359373, October 2003.

- [HH10] Stefan Holdermans and Jurriaan Hage. Making "strictness" more relevant. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 121–130, New York, NY, USA, 2010. ACM.
- [Hos11] Kenneth Hoste. The GA Package. <https://hackage.haskell.org/package/GA>, 2011.
- [HS07] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. ICFP '07, 2007.
- [Hug89] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989.
- [Inc16] Google Inc. *SoftReference — Android Developers*, 2016 (Accessed March 23, 2016).
- [Jik05] Jikes RVM. IBM, 2005. <http://jikesrvm.sourceforge.net>.
- [jso12] City of Chicago Public Datasets. <https://www.opensciencedatacloud.org/publicdata/city-of-chicago-public-datasets/>, 2012.
- [JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the 1st International Symposium on Memory Management*, pages 26–36, 1998.
- [KBG12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI12, page 3146, USA, 2012. USENIX Association.
- [LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419429, June 1983.

- [MS03] Nick Mitchell and Gary Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, pages 351–377, 2003.
- [Myc82] Alan Mycroft. Abstract interpretation and optimising transformations for applicative programs. 1982.
- [New05] M. E. J. Newman. Power laws, Pareto distributions and Zipf’s law. *Contemporary Physics*, 2005.
- [NFX⁺16] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI16*, page 349365, USA, 2016. USENIX Association.
- [NGB16] Diogenes Nunez, Samuel Z. Guyer, and Emery D. Berger. Prioritized garbage collection: Explicit gc support for software caches. *SIGPLAN Not.*, 51(10):695710, October 2016.
- [NWB⁺15] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. *SIGARCH Comput. Archit. News*, 43(1):675690, March 2015.
- [OB06] Melissa E. O’Neill and F. Warren Burton. Smarter garbage collection with simplifiers. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness, MSPC ’06*, pages 19–30, New York, NY, USA, 2006. ACM.
- [Ora15] Oracle. *SoftReference (Java Platform SE 6)*, 2015.

- [Par93] Will Partain. The `nofib` benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer-Verlag, 1993.
- [PRW03] David W. Price, Algis Rudys, and Dan S. Wallach. Garbage collector memory accounting in language-based systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy, SP '03*, pages 263–, Washington, DC, USA, 2003. IEEE Computer Society.
- [RGM13] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. Elephant tracks: Portable production of complete and precise gc traces. In *Proceedings of the 2013 International Symposium on Memory Management, ISMM '13*, pages 109–118, New York, NY, USA, 2013. ACM.
- [Ric16] Nathan Ricci. Determining when objects die to improve garbage collection, 2016.
- [SF18] Marilyn Sun and Kathleen Fisher. Autobahn 2.0: Minimizing bangs while maintaining performance (system demonstration). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell 2018*, page 3840, New York, NY, USA, 2018. Association for Computing Machinery.
- [SM10] Tom Schrijvers and Alan Mycroft. Strictness meets data flow. In *Proceedings of the 17th International Conference on Static Analysis, SAS'10*, pages 439–454, Berlin, Heidelberg, 2010. Springer-Verlag.
- [str15] Language options —Glasgow Haskell Compiler 8.10.1 User’s Guide. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghc_exts.html#extension-Strict, 2015.
- [Sys89] Gilbert Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, page 29, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

- [SZ98] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. *SIGOPS Oper. Syst. Rev.*, 32(5):1223, October 1998.
- [TR15] José Manuel Calderón Trilla and Colin Runciman. Improving implicit parallelism. Haskell '15, 2015.
- [TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA '95*, pages 1–11, New York, NY, USA, 1995. ACM.
- [UJ92] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Trans. Program. Lang. Syst.*, 14(1):127, January 1992.
- [Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGSOFT Softw. Eng. Notes*, 9(3):157167, April 1984.
- [VH15] Hidde Verstoep and Jurriaan Hage. Polyvariant cardinality analysis for non-strict higher-order functional languages: Brief announcement. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM '15*, pages 139–142, New York, NY, USA, 2015. ACM.
- [WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, pages 1–116, September 1995.
- [WNF16] Yisu Remy Wang, Diogenes Nunez, and Kathleen Fisher. Autobahn: Using genetic algorithms to infer strictness annotations. *SIGPLAN Not.*, 51(12):114–126, September 2016.

- [XBQR11] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 270–282, 2011.
- [YM14] Edward Z. Yang and David Mazières. Dynamic space limits for Haskell. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 588–598, New York, NY, USA, 2014. ACM.