2019

Subheap-Augmented Garbage Collection

Benjamin Karel
University of Pennsylvania

Subheap-Augmented Garbage Collection

Abstract

Automated memory management avoids the tedium and danger of manual techniques. However, as no programmer input is required, no widely available interface exists to permit principled control over sometimes unacceptable performance costs. This dissertation explores the idea that performance-oriented languages should give programmers greater control over where and when the garbage collector (GC) expends effort. We describe an interface and implementation to expose heap partitioning and collection decisions without compromising type safety. We show that our interface allows the programmer to encode a form of reference counting using Hayes' notion of key objects. Preliminary experimental data suggests that our proposed mechanism can avoid high overheads suffered by tracing collectors in some scenarios, especially with tight heaps. However, for other applications, the costs of applying subheaps---in human effort and runtime overheads---remain daunting.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Jonathan M. Smith

Keywords

Garbage Collection, Memory Management, Subheaps

Subject Categories

Computer Sciences

SUBHEAP-AUGMENTED GARBAGE COLLECTION

Benjamin Karel

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2019

Supervisor of Dissertation

Jonathan M. Smith, Olga and Alberico Pompa Professor

Graduate Group Chairperson

Rajeev Alur, Zisman Family Professor

Dissertation Committee

Steve Zdancewic (Professor of CIS; Committee Chair)
Andreas Haeberlen (Associate Professor of CIS)
Joe Devietti (Assistant Professor of CIS)
Alex Garthwaite (External Member, VMware)

ACKNOWLEDGMENTS

I am indebted, in ways large and small, to many whose paths have crossed with mine over the years. My advisor, Jonathan Smith, has not only imparted the wisdom of experience in matters academic, but has also demonstrated by example a more exalted life: unfailingly kind to all who deserve it and many who don't, buttressed by a stubborn positivity in which any day that starts with waking up is a good day. André DeHon has demonstrated admirable patience and tireless dedication as a collaborator and surrogate advisor. My committee—Andreas Haeberlen, Joe Devietti, Steve Zdancewic, and Alex Garthwaite—have provided helpful feedback on my drafts and ideas and have been universally generous with their time. In words and actions alike, Benjamin Pierce has demonstrated the power and importance of precise thought and clear writing. Milo Martin and Stephanie Weirich both provided sage advice (which, to my detriment, I did not fully heed) and constructive feedback in my first years at Penn.

Many fellow graduate students have influenced my time at Penn. Perry Metzger encouraged me both in my transition into the doctoral program and my pursuit of—as he put it—writing the Great American Compiler. Adam Aviv and Katie Gibson welcomed me into Jonathan's group. Michael Greenberg, Benoît Montagu, and Cătălin Hriţcu provided friendship, support, and guidance over our respective stints on the SAFE project. But the single biggest fixture of my time at Penn has been Nikos Vasilakis. He has been a fine friend and a generous collaborator. It has been an honor and a privilege to witness his skills—in research and in writing—far exceed my own.

Last but not least, I owe much to my family. To my parents, for their unwavering support and all the countless things they have done to help me get to where I am. And to my wife Lauren, last on the page but first in my heart, for being the other half of my orange, and remaining so as the years accumulated past my initially planned graduation date.

ABSTRACT

SUBHEAP-AUGMENTED GARBAGE COLLECTION

Benjamin Karel

Jonathan M. Smith

Automated memory management avoids the tedium and danger of manual techniques. However, as no programmer input is required, no widely available interface exists to permit principled control over sometimes unacceptable performance costs. This dissertation explores the idea that performance-oriented languages should give programmers greater control over *where* and *when* the garbage collector (GC) expends effort. We describe an interface and implementation to expose heap partitioning and collection decisions without compromising type safety. We show that our interface allows the programmer to encode a form of reference counting using Hayes' notion of key objects. Preliminary experimental data suggests that our proposed mechanism can avoid high overheads suffered by tracing collectors in some scenarios, especially with tight heaps. However, for other applications, the costs of applying subheaps—in human effort and runtime overheads—remain daunting.

Chapter Contents

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1 : Introduction

Most computer programs allocate memory as they run, but memory is a finite resource. Reclaiming unused memory safely and efficiently motivates the study of garbage collection (GC). Functionally correct GC algorithms date to the early years of computer science [McC60, JHM11]. Yet *correctness* is not enough: effort over subsequent decades has focused on various facets of *performance*, such as low pause times, high space efficiency, and productive use of available hardware resources. Research in garbage collection has produced designs that work well for most programs, but every GC embodies tradeoffs and heuristics that may be ill-suited for certain programs. Thus the goal for GC design is not merely to produce collectors which are efficient on average, but those which perform well for the widest range of programs.

The quest for efficient execution that avoids the performance pitfalls of GC has also led to the study of alternatives to GC. Examples include region-based memory management [TBEH04] and substructural type systems [Tov12], as well as the continuing use of unsafe manual memory management techniques. Manual techniques afford programmers the flexibility to use the most suitable disciplines for their particular program, boosting efficiency. While unsafe manual techniques can confer speed, their lack of memory safety contributes to a software ecosystem featuring widespread exploitation of vulnerabilities. Losing safety is too high a price for performance.

This dissertation investigates *subheaps*, a novel scheme for allowing programmers to divide the heap and identify profitable collection points in order to improve the performance of garbage collection. These elements respectively constitute the "where" and "when" of memory reclamation. Subheaps preserve the safety of automatic collection while seeking to gain some of the benefits of programmer input.

1.1. Motivation: Bridging Language Users and Implementors

The literature has shown that different garbage collectors perform markedly better or worse with particular programs [FT00, SK07, SBWC07, JCMM16]. One can think of different GC algorithms as having "rough edges"[1] that snag on particular allocation patterns. These rough edges are problematic for both language implementors and language users. Each algorithm can also be tuned in various ways, but optimal configurations are often input- or machine-dependent.

The language implementor is faced with the choice of selecting a GC design to suit the needs of all future users. One approach is to provide several GC implementations and let the user choose between them. For example, the HotSpot JVM (version 7.0) shipped with four separate GCs: serial, parallel, concurrent mark-sweep, and garbage-first [JDK]. This complexity is a large burden in engineering, documentation, and testing effort. Most language environments provide a single GC implementation, and rely on heuristics to get good performance for most programs. Many problem domains can see significant gains in performance with customized GC heuristics [Har00, SKB04, NFX+16, DEE+16, GGS+15, MHAK15], but language implementors cannot anticipate the needs of all current and future problem domains.

Meanwhile, users must make do with the tools provided by their language implementation. A topic near and dear to the hearts of systems programmers is the issue of **control**. GCs rely heavily on heuristics, especially for when, where, and how much to collect. With such heuristics, sophisticated GC implementations can offer excellent performance to most programs in most circumstances. The cost of a sophisticated collector comes in opaqueness and loss of predictability. As the GC becomes more

[1] Credit to Alex Garthwaite for this turn of phrase.

complex, users lose insight into the GC's rough edges and how to overcome them.

What can be done when pre-chosen heuristics end up poorly suited to one's program? Performance problems cropping up due to memory management leave the user in a tight spot, facing unappealing choices. Using more powerful hardware is expensive at scale. Tweaking GC configuration parameters can be a slow process because engineers generally have an indirect understanding of how each setting will interact with their program. Furthermore, tuning GC parameters often has fragile results. Rewriting in a different language is expensive and often infeasible. Many developers end up "fighting" the GC, via dubious tricks like manual object pooling (which undermines temporal safety) or using "off-heap" allocations (which throws out the benefits of GC entirely). In most environments, programmers have little recourse.

This dissertation envisions and explores subheaps: a tool to bridge the gap between language implementors and language users. For users, subheaps provide a way to "customize" an off-the-shelf GC to one's particular program. For implementors, subheaps represent a simple mechanism that can obviate the necessity of providing multiple redundant GC implementations in order to provide acceptable GC performance to a wide range of user programs. One lens on subheaps is as an argument for *programmer control over GC* as a language survival characteristic [Gab].

1.2. The Core Idea Of Subheaps

Subheaps give the programmer an API to guide the garbage collector's effort. The programmer's goal is to divide the heap into multiple pieces which can be collected independently and efficiently. Each piece is called a subheap. Programmers modify their programs to dynamically create, activate, and collect subheaps. A key assumption behind the idea of subheaps is that *humans* can (sometimes) make careful choices

for when and where to collect a subheap, simultaneously increasing efficiency and efficacy of GC.

To be clear, subheaps are not a new garbage collection algorithm or implementation in the usual sense. Most designs for new GCs apply to all programs automatically. Subheaps differ in this aspect. If a particular program does not make use of the subheap API, the program will not see any benefits from running against a subheap-enabled GC, and its garbage will have to be collected with some existing GC design.

Subheaps offer two potential benefits to the world at large, at least for those programs which can make productive use of subheaps. First, increasing program efficiency can save money by reducing hardware costs and/or software engineering effort. Second, subheaps offer the potential to eliminate technical factors favoring unsafe languages stemming from the flexibility and performance potential of manual memory management, thereby strengthening defensive cybersecurity.

1.3. Purpose & Contributions

Statement of Purpose Section 1.1 explains why fully-automatic garbage collection is not yet a completely solved problem. The idea at the core of this document is that human guidance can (sometimes) improve the performance characteristics of automatic memory management. Subheaps represent one simple point in the design space to explore this idea. This dissertation investigates how, and to what degree, subheaps can be implemented and deployed to achieve their intended benefits.

To fulfill its purpose, this dissertation makes the following contributions:

- A design for a subheap API, with corresponding implementation, to allow the programmer to influence *where* and *when* the collector expends effort.

- Discussion of how subheaps can be applied in practice, especially in how subheaps allow adaptations for applications such as software caches.
- Discussion of the challenges involved in static removal of subheap write barriers, along with evaluation of a simple scheme for automatic barrier elimination.
- Preliminary evaluation of subheaps, illustrating both potential benefits as well as costs and drawbacks to the use of subheaps.
- An open source implementation of subheaps made publicly available[2] for other researchers to build upon.
- Thorough coverage of how subheaps relate to the literature on garbage collection and region-based memory management.

Make note of what is not being claimed in this dissertation. In particular, subheaps are not being presented as a universal improvement upon existing collectors, nor even as a desirable mechanism to include (in their current minimalist form) in future languages. Rather, subheaps show promise and work well for small programs, but results on larger programs indicate that more research is likely to be needed before (some future variant of) subheaps will be ready for general programmer consumption. Section 7 covers these views in more detail.

1.4. Roadmap

The remainder of this chapter introduces basic terminology for garbage collection. Section 2 lays out the core ideas behind subheaps: the concrete API between the programmer and the collector, the underlying principles and design constraints behind the API, prototype uniprocessor implementations, and a variety of extensions to

[2]Source and documentation for subheaps at `https://eschew.org/projects/subheaps/`

the core. Section 3 discusses the practical use of subheaps: why, where, and how programmers can use the subheap API to reduce collection costs. Section 4 evaluates subheaps. Section 5 reflects on some of the challenges facing subheaps, and speculates on future work to address those challenges. Section 6 lays out related work, and Section 7 concludes.

1.5. Garbage Collection, Briefly

Subheaps occupy a niche in the design space for garbage collection. To understand the design space for subheaps, it's useful to have a baseline understanding of concepts and terminology from the GC literature. What follows is a terse overview; the Garbage Collection Handbook [JHM11] gives a fuller understanding of the history and practice of garbage collection. Readers with a background in GC may safely skip ahead.

Garbage collection comprises *reference counting* and *tracing*-based techniques, which are dual to each other [BCR04]. Each alternative aims to reclaim allocated memory from the *heap* of allocated objects, starting from the program's *roots*: registers, stacks, and globals. Reference counting tracks how many copies of a given pointer exist in the heap; when the heap contains zero copies of a pointer, the associated memory is *dead* and can be *freed* for reuse. Tracing computes the transitive closure of references from the roots; any allocated data not thusly accessible can be reclaimed. Reachability is the standard approximation to the undecidable property of *live*ness, and the terms are often treated synonymously.

The three "elemental" tracing algorithms are mark-sweep, compaction, and semispace collection. Mark-sweep allocates objects from one or more *free lists* of available memory, and returns dead objects to the free list(s) in a "sweep" of the heap. Unlike compacting or semispace collectors, mark-sweep does not copy objects as it runs.

Copying enables a fast allocation scheme called *bump allocation* in which objects are allocated from a large chunk of address space by incrementing (or decrementing) a pointer. These building blocks may be combined to form hybrid collector designs.

The most common hybrid is *generational* collection, in which the heap is split into two or more *spaces* (disjoint sets of objects) termed generations. The simplest generational design has two spaces. Newly-allocated objects are placed in the youngest generation, also called the nursery. Objects which remain alive at the next collection graduate into the oldest generation, called the mature space. By maintaining a *remembered set* of generation-crossing references to be used as an additional source of roots, the young generation can be collected without inspecting the whole heap. Remembered sets are kept up-to-date by *write barriers*: small pieces of code emitted by the compiler that run every time a value in the heap is modified. Write barriers effectively allow the *mutator* (the *client* program relying on the GC) to communicate relevant information to the collector. Generational collection is one example of *partitioned* or *space-incremental* collector design.

1.5.1. Garbage Collection Tradeoffs

One major tradeoff in the design of garbage collectors is time versus space: in general, the more space is available, the less time must be spent reclaiming memory. Smaller heaps must be collected frequently. Conversely, with a sufficiently large heap, garbage collection isn't needed and thus takes zero time [SJBL10].

Another major tradeoff is latency (pause time) versus throughput (overall time taken). Large pauses are undesirable for interactive applications such as GUIs or servers. But techniques that improve latency, such as *incremental* collection, often degrade overall performance. A key design choice is *amortization*: doing work in larger batches

improves throughput and degrades latency.

The latency-vs-throughput tradeoff is also reflected in treatment of *fragmentation*. Fragmentation occurs when free space is divided into many pieces, each too small to be individually useful. This reduces the effective size of the heap and makes CPU caches less effective. Copying collectors eliminate fragmentation. For a *popular* object, one with many references to it throughout the heap, updating all references to the newly copied object can degrade latency. The effect on throughput of combating latency is a mixed bag; copying is not free but its cost may be offset by increased locality for the mutator.

Related to the question of batch sizing is the tension between local operation and cycle-completeness. To reclaim a given allocation requires a summary of the rest of the heap. Examples of such summaries include reference counts and remembered sets. Summaries allow processing smaller portions of the heap at a time, which improves latency and may even boost throughput. But when a reference cycle crosses a summarized boundary, no purely-local operation can identify the cycle. Global views—encompassing both sides of the boundary—are needed to handle *cyclic garbage*. Cyclic structures, such as doubly-linked lists and trees with back-pointers, are common enough to warrant consideration. As a result, practical garbage collectors either use tracing or augment reference counting with dedicated cycle collection routines.

Finally, in block-structured or space-partitioned collectors, there is a tradeoff in the choice of block size. Small fixed-size blocks reduce the wasteful impact of unallocated space (also known as *internal fragmentation*) but produce many inter-block references that may need to be tracked. Larger blocks reduce such overheads. Subheaps aim to resolve the tradeoff by supporting a small minimum block size and letting subheaps grow dynamically rather than imposing a fixed maximum size. This relies crucially

on programmer input to choose advantageous heap partionings.

1.5.2. Concurrent Collection versus Work Reduction

Collectors can take advantage of surplus hardware resources via *concurrent* and/or *parallel* techniques. Such approaches can improve both throughput and latency, but they do not reduce overall work done. Instead, they increase total work due to synchronization overhead.

Work reduction can matter for energy constrained environments, such as mobile devices. It is also vitally important for getting robust control over collection costs in large heaps. A standard machine-independent measurement of work is the *mark/-cons ratio*, defined as (allocations or bytes of) data marked or copied, divided by data allocated.[3] The mark/cons ratio approximates the average per-allocation work done by the collector. Eliminated work will be reflected in a lowered mark/cons ratio.

[3]The term *cons* to represent allocated data refers to the Lisp function for allocating pairs.

CHAPTER 2 : Subheaps

The core goal of subheaps is to provide a flexible mechanism to give users control over the throughput and latency costs of tracing GC by focusing collection effort on high-yield parts of the heap. The remainder of this chapter discusses the core principles behind subheaps, illustrates the constraints implied by those principles, introduces a design for subheaps satisfying the constraints, and details the elements needed for an efficient implementation of the subheap design.

2.1. Subheap Principles

The fundamental task of a garbage collector is to automatically and safely reclaim memory for reuse. Although their goal is to find dead space, GCs must waste effort identifying live data. Tracing GCs expend this effort at collection time, whereas reference counting operations are performed by the mutator. This wasted effort is the primary cost of garbage collection.

Subheaps furnish two mechanisms: they enable programmer-controlled subdivisions of the heap, and allow programmers to collect particular subdivisions. These mechanisms correspond to the "where" and "when" of collection. They in turn allow programmers to influence the performance characteristics of tracing in principled ways. Collection effort can be redirected "spatially" by biasing collection towards dead objects and away from live ones. Grouping objects with similar lifetimes can improve performance by reducing the collector's wasted effort. Collection effort can also be shifted "temporally" by explicitly triggering collection.

2.1.1. Risks

The mechanisms provided by subheaps are a double-edged sword. Unlike some schemes for memory management, it is not possible for subheaps to violate type- or memory safety. However, careless use of subheaps can cause application performance to degrade rather than improve. The reason is that explicitly triggered collections can waste effort (re-)examining live objects. With regular garbage collection cycles, which are usually driven by memory pressure, this effort is limited by the rate of program allocation and the amount of memory reclaimed per collection. Unlike regular garbage collection cycles, the potential overhead of subheaps is disconnected from such limitations.

2.1.2. Cost Model

The most costly part of garbage collection is tracing through live data, whereas the benefit of collection is in reclaiming dead space. By using the subheap mechanisms to exploit knowledge of the lifetimes of (some) objects within their applications, programmers can reduce the cost or increase the benefit of collection. In the limit, when every object under collection is dead, the primary cost of collection—that of tracing live objects—goes to zero. Even when no tracing occurs, collections have non-zero cost due to scanning stacks and remembered sets.

Most use cases for subheaps will improve performance by reducing the amount of tracing required, thereby shrinking the performance impact of garbage collection. Particular configurations of subheaps can, however, have other impacts on overall program runtime. Mutator performance can be degraded by the execution of write barriers and remembered set maintenance needed to support subheap collection. More subtly, use of subheaps can change program locality properties, which could either

improve or degrade performance. These second-order costs will be more apparent when an application of subheaps does not produce large savings in reduced tracing costs.

2.2. Design Constraints

The core principle behind subheaps—of reducing GC cost by eliminating tracing work—leads to several constraints for an implementation of subheaps. First, subheaps should optimize for collecting dead space; collecting dead objects should be nearly free. Second, subheaps must be able to efficiently partition the heap in arbitrary ways. Third, subheaps must support the collection of arbitrary sets of subheaps.

The desire to make collection cost shrink to zero for all-dead subheaps suggests that any operation which has cost proportional to the size of a subheap—rather than the size of the subheap's live data—is verboten. However, constant factors matter. Per-object operations, such as eager free-list sweeping, would be unacceptably slow, but manipulating chunks of space at coarse granularity can reduce the associated constant factor enough to become an insignificant cost. Thus, subheaps aim to exploit *amortization* for efficiency. However, quick reclamation of dead space is not the end of the story. While tracing live data is the largest cost of collection, it is not the only cost; finding roots in stacks and remembered sets has non-zero cost. And in many applications, the cost of allocation outweighs the cost of collection, so care must also be taken not to degrade allocation for the sake of collection.

Note a non-constraint for this incarnation of subheaps: because subheaps are meant to exploit human knowledge, we acccept the burden of having source code, rather than targeting bytecode or object code. Future research on automating use of subheaps might expand deployment of subheaps to bytecode-only codebases.

2.3. Subheap API

Source code is required because programmers must modify their programs to make use of the subheap API. Here is the low level API for using subheaps:

```
subheapCreate   :: { () => Subheap }
subheapActivate :: { Subheap => Subheap }
subheapCondemn  :: { Subheap => () }
subheapCollect  :: { () => () }
```

The function name is on the left, followed by a type signature. Function types are delimited by curly braces, with arguments and return types separated by arrows. The unit type, `()`, is a placeholder similar to `void` in the C family of languages.

As the program executes, new allocations go into some subheap, called the *active* subheap. The `subheapActivate` function marks the given subheap as being the new active subheap. It also returns the previously-active subheap, making it easier to manipulate subheaps in a cleanly nested way. We refer to the subheap returned by the program's first call to `subheapActivate` as the default subheap. In a program that ignores the subheap API, all allocations remain in the default subheap.

Condemning a subheap indicates that its contents should be prioritized at the next collection cycle, but does not initiate reclamation. There are several possible design choices for the precise semantics of condemnation. We focus on the simplest version, in which condemned status is associated with subheaps rather than their contents; thus, condemning does not have "snapshot" semantics. Multiple subheaps can be condemned before collecting, and condemned status does not persist between collections. Compared to an interface which collects an explicitly-provided collection of subheaps, the condemned set abstraction is simpler, more flexible, and more efficient.

We can define a fifth operation, `subheapReclaim`, by composing `Condemn` and `Collect`. Why not provide only the convenient wrapper? The issue is that it *conflates* distinct elements of the programmer's knowledge, because it ties together issues of *where* and *when* the GC should run. It is also unnecessarily limiting: by reclaiming only a single subheap at a time, we lose the ability to bypass stale remembered sets. Still, `subheapReclaim` is the more convenient interface to reclamation, and suffices for many common use cases.

The activation-based, dynamically-scoped execution model for the subheap API was chosen as a simple point in the design space to begin exploration. It is arguably the least essential (or least principled) element of the subheap design, and would be the easiest to experiment with in future iterations of research on subheaps.

2.4. Subheap Implementation

Our prototype subheap-augmented garbage collector is based on the Immix mark-region [BM08] heap design. Mark-region combines the speed benefits of contiguous allocation with the space efficiency of sweeping. Importantly, given that successful use of subheaps implies proportionally less tracing and more reclamation, mark-region also enables rapid reclamation of empty space. Another appealing property of Immix is that it has been extended with several variants of generational collection, based on copying, sticky mark bits, and reference counting [BM08, SBYM13]. Subheaps have been integrated atop two distinct generational baselines: StickyImmix and ImmixRC. Details of these extensions may be found in Section 2.5.

The primary implementation of subheaps has been written for an ML-like language called Foster. The Foster project is (was) an exploration of using language-based technology to reliably eliminate conventional overheads of type safety. Subheaps

were developed to address GC costs in this performance-sensitive context. The Foster compiler targets LLVM and implements both traditional compiler optimizations such as inlining as well as GC-specific optimizations for stack slot management and write barrier elimination. Both the Foster compiler and its subheap-enabled runtime are freely available for anyone to study, use, or modify, at `https://eschew.org/projects/subheaps/`.

The remainder of this section details the design and implementation of Foster's subheap-augmented collector.

Overview The heap is structured into 32 KB `frame`s, which are in turn composed of 256-byte lines. Each line is associated with a mark byte, a used byte, and a stamp. The mark byte is standard; the used byte is needed when combining subheaps with generational collection, and the stamp helps manage remembered sets. Fixed-size heap metadata is segregated in demand-paged virtual memory, allowing constant-time lookup of metadata for specific portions of memory [SMB04, Kus15]. Line-based metadata is used to calculate frame residency statistics.

Remembered Sets To enable independent collection, each subheap maintains a remembered set of incoming pointers. The representation of remembered sets is mostly orthogonal to the design of subheaps. The primary restriction is that for emulated reference counting (see Section 3.5) to work, remembered sets must record the slots containing pointers, rather than pointer targets. Foster uses a standard sequential-store-buffer design. Coarser granularities could also be viable; the literature describes several potentially relevant optimizations [SMB04, Sjö14, Ada07]. Regardless, the asymptotic overhead of remembered sets (sometimes referred to as *remsets*) ultimately depends on the heap division chosen by the programmer.

One unique wrinkle arising from collecting arbitrary sets of subheaps is the need for the runtime to purge stale remembered set entries. Otherwise, reuse of either the source or target of a remembered pointer can result in the runtime interpreting arbitrary bit patterns as a pointer. This issue can be finessed when the runtime controls, or at least knows about, the future order of collections [Ste99]. But with subheaps, programmer input controls collection order. Thus the runtime must identify when lines holding remembered pointers have been reused. This is accomplished with a per-line timestamp, recorded in remembered sets and incremented upon reclamation (or reallocation). A 32-bit stamp per 256-byte line imposes space overhead of 1.5%. Line stamps would not be needed for subheaps atop conservative Immix [SBM14].

Allocation Allocation for subheaps mirrors that of Immix. Allocation requests are routed through a pointer representing the current subheap. Subheaps grab free line spans from a cache, falling back to linear inspection of used line marks, and record the set of lines they own. Activation of a new subheap overwrites this pointer, and "steals" the unused trailing lines from the previous subheap, to minimize wasted space. By managing space around groups of lines rather than whole frames, this scheme enables fine-grained behavior while still preserving efficient amortization for coarse-grained subheaps.

The main function of reconstructing spans from line metadata, versus keeping an explicitly-represented pool of available spans, is to combat fragmentation from line spans being reclaimed piecemeal. The downside is that purely linear sweeps do not preserve locality in the presence of high-frequency subheap churn. A cache of available lines permits flexibly combining locality preservation with fragmentation avoidance.

As with the baseline Immix collector, object allocations larger than 8KB are diverted

to `malloc`, limiting size-based fragmentation in the block-structured heap to 25%. Each subheap keeps a list of the large objects it owns, and scans the list at each collection to find unmarked entries. Due to the minimum size requirement, separate treatment of large objects does not alter the overall analysis of performance considerations for the block-structured heap. Arraylet techniques [BCR03b, SBF+10, PZM+10] could be employed by a language runtime or standard library to ensure the data for large arrays is managed within the block-structured heap.

To support collection of arbitrary partitionings, the subheap implementation must be able to efficiently find the subheap associated with a given object. This operation is used "internally" to identify subheap-crossing pointers, both in the write barrier and during collections. Following Yak [NFX+16], we provide fast mapping from objects to subheaps by embedding a subheap identifier in object headers. In Foster, this 32-bit identifier conveniently fits into the unused half of an eight byte object header.[1]

Per-Frame Metadata Two additional words are associated with each 32 KB of virtual address space. One word stores a few bytes of statistics: per-frame counts of available lines and holes, used to prioritize opportunistic evacuation [BM08], plus a count of how many lines in that frame belong to the default subheap, used to permit compaction. Frames containing large allocations are associated with a dynamically allocated four-element extension, since there can be at most four large objects per 32 KB region of memory.

Condemnation The `subheapCondemn` function records the given subheap in the *condemned set*. It also sets a flag in the given subheap to speed determination of condemned status for individual objects at collection time. Condemnation is constant

[1] The other half stores object type information; the header was eight bytes rather than four due to Foster keeping 16-byte object alignment.

time; it does not inspect or modify any remembered set entries. Condemnation is idempotent until the next collection of a given subheap.

Reclamation The first step of a collection cycle is to establish the condemned set, either from explicit user requests or as selected by the runtime. The next step is to gather locations of potential incoming pointers from the remembered set, ignoring entries between condemned objects. The overall set of incoming pointers from non-condemned objects, plus globals[2] and stacks, constitutes the root set. If the root set is empty, the entirety of the condemned set can be reclaimed immediately. Otherwise, tracing and marking from the roots proceeds as usual, with the proviso that the collector ignores non-condemned objects. At the start of a collection, the condemned set's marked line maps must be cleared (unless the current collection is operating as a generational nursery). After marking finishes, mark bytes for the condemned set are copied to the line-used map. After each collection, the condemned set and associated flags must be reset.

Scanning of line maps is proportional to the size of the condemned set rather than the size of the live data, but the associated constant is quite small. Measuring the cost of scanning line maps for an unfragmented subheap of 1 GB revealed a mean cost of 325 μs, implying a reclamation rate of 3 GB/ms. This rate of reclamation applies to *almost entirely empty* subheaps. In subheaps with even a small percentage of live data, the cost of tracing dominates the cost of reclaiming space. Neither cost applies to subheaps with no incoming pointers, which can be reclaimed without scanning linemaps. Inspecting line maps is more than two orders of magnitude faster than the cost of allocation itself.

[2]Globals are traditionally a source of roots. In Foster, globals are statically allocated and immutable, and thus are never part of the root set.

When collection occurs without an explicitly-chosen condemned set, the runtime must choose which subheaps to collect. The simplest heuristic is to collect all subheaps; this guarantees that any available space will be made available for allocation, without any data being spuriously kept alive by remembered sets. Collecting a smaller subset of the heap—such as only the default subheap—is possible, and can improve performance by reducing duplicate tracing of data in long-lived subheaps. Doing so risks performing duplicate work if the chosen subset yields sufficiently little free space that fallback collection of the whole heap is eventually needed. Section 2.7.1 explores how the user can influence the set of implicitly-chosen subheaps in order to improve performance and avoid wasted work.

Used Bits Non-subheap collectors can maintain a simple invariant: after collection, used lines are marked, and unmarked lines are not used. This allows used status to be be derived from mark bits rather than being explicitly represented. With subheaps, an uncondemned line can be used but not marked. This dissertation's design, which combines subheaps with sticky mark bit generational collection, requires separate metadata for line mark versus used status. Used bits are set during allocation and cleared post-collection. Mark are set during collection, and only reset before mature space collections. The relevant state machine is illustrated in Figure 1. By keeping the invariant that cached spans are marked used, the allocation cache reduces the overhead from (re)setting used bits.

Subheap Representation and Lifecycle While the semantics for subheap objects (that is, language values of type `Subheap`) given so far could be adequately represented by an opaque integer value, the prototype instead uses heap-allocated handles to allow the runtime to detect unmarked subheap values. Rather than provide a means via the API for explicit destruction of subheaps, the runtime automatically

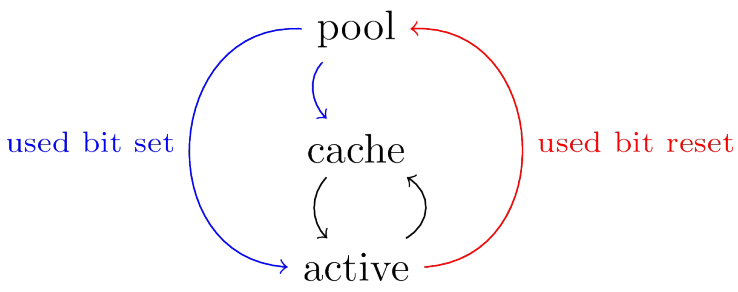


Figure 1: Per-line state machine for the "used" bit. Note that these are logical states; the pool is represented implicitly, whereas the line cache is represented explicitly.

destroys subheaps which are both unmarked and empty. Subheaps which are empty but not unmarked might be activated in the future, whereas non-empty unmarked subheaps cannot be destroyed until the objects allocated within them die (or are evacuated).

The backing object for a subheap occupies 360 bytes in the current prototype. This space cost breaks down into: 24 bytes each for bump allocators devoted to small and medium-sized objects; 24 bytes for tracking large array allocations; 32 bytes for tracking allocated spans; 120 bytes for the subheap remembered set (with timestamps), 104 bytes for a generational remembered set, two bytes to hold "condemned" and "short-lived" flags, and 30 bytes of padding for alignment. While engineering effort could reclaim some of this space, there will always be some unavoidable bookkeeping overhead associated with each subheap. This overhead imposes severe diminishing returns on the benefit of having subheaps share space at granularities finer than a single line.

2.4.1. Subtle Elements of Subheap Collection

A common optimization in traditional collectors is to lazily perform GC cleanup actions, such as mark bit sense flipping. This avoids the cost of resetting all marked

objects by redefining object mark status from an absolute marked/unmarked status bit to be defined relative to a global "sense" bit. Doing so permits "unmarking" the entire heap in constant time, thereby reducing the collector's workload. Unfortunately, flipping the sense of the mark bit is only possible for full-heap collections;[3] flipping the meaning of the mark bit for an individual subheap is not coherent.

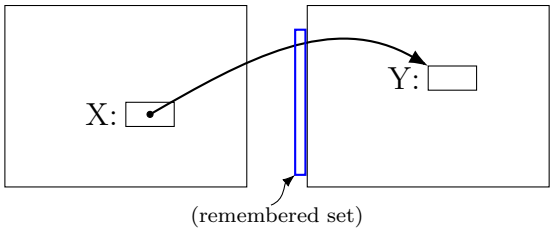
To minimize the impact of this foregone optimization, the design for subheaps relies instead on locality via segregated mark bits for both lines and objects. Long-deployed hardware tricks, especially prefetching, provide attractive constant factors for linear walks through memory. Besides having little impact on GC speed, segregated mark bits also impose low space overhead. With a minimum object size of 16 bytes, one bit per (potential) object requires only 0.78% space overhead. Using a full byte per object, which avoids bit-manipulation overhead in a serial collector and ensures atomicity without the cost of compare-and-swap atomics for concurrent marking, imposes an additional space overhead of 5.47%.

Another subtlety with subheaps has to do with remembered sets. In particular, whenever two or more subheaps are collected, their remembered sets must be trimmed of any stale entries from condemned subheaps. A stale entry is a pointer either from or to an object left unmarked after collection. This scenario most commonly occurs for whole-heap collections.

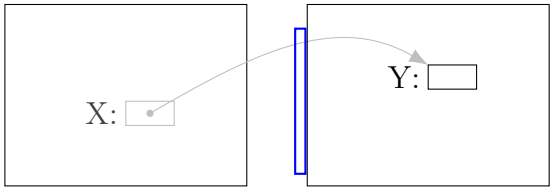
If stale entries are not trimmed from remembered sets, unsoundness can arise from the following sequence of events:

1. A reference to object Y is stored in slot X, so Y's space records slot X in its remembered set.

[3] In a generational collector, nursery evacuation provides the opportunity to clear mark bits.

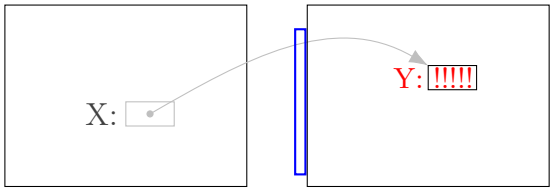


2. The object owning slot X dies.

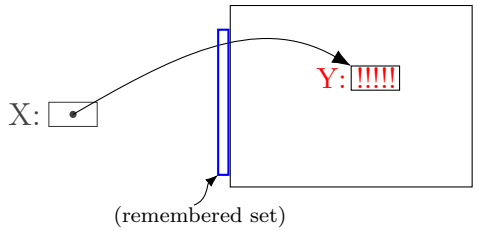


3. A GC occurs which includes the subheaps for X and Y. Because both sides are condemned, Y's remset entry for X is ignored. Y will be left unmarked, assuming X was the last object referring to Y, because X is dead.

4. Because Y is unmarked, subsequent allocation in Y's subheap puts an arbitrary bit pattern in X's referent (particularly the object header).

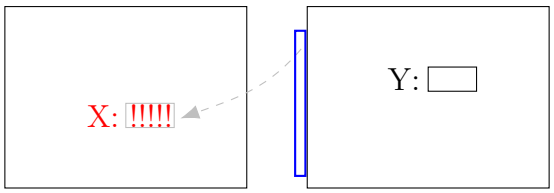


5. The next collection of Y's subheap consults the remembered set and finds an entry for slot X. Assuming that X's memory has not been reused yet, it still contains a valid-looking bit pattern for the ghost of Y. Attempts to trace through this ghostly pointer are erroneous; a GC invariant (the integrity of header words) has been violated.



Eagerly zeroing reclaimed memory would prevent the problem, but also drastically increase the (latency) cost of reclaiming large subheaps with little live data. It is cheaper to instead lazily detect and remove stale remembered set entries.

A similar situation can arise without simultaneous collection of multiple subheaps. Suppose after step 2 above, only X's subheap is collected. Having been reclaimed, the source slot X can be overwritten with an arbitrary bit pattern, of arbitrary type, by future allocations.



Yet the subheap containing Y can still find slot X through its remembered set. Thus, when Y's subheap is collected, it must treat the bit pattern it reads with caution. Two possibilities are to either parse the heap to verify that the slot is typed to hold a pointer, or fall back to conservative [SBM14] treatment of the potential root.

We instead have the collector maintain metadata to help identify stale remset entries. Lines are timestamped to indicate when they were last initialized; remset entries copy the stamp. When a remset entry's stamp disagrees with the line's stamp, the entry is stale. By making stamps large enough to avoid wraparound, this mechanism suffices to identify stale entries. Otherwise, cautious treatment would still be needed to deal with spurious agreement due to wraparound. A stamp size of four bytes per line

amounts to space overhead of roughly 1.5%.

These situations do not arise for generational designs because generational collectors maintain stricter invariants: they disallow collection of the mature space independently from the nursery, and they either recompute or clear the nursery's remembered set on full-heap collections.

2.4.2. Subheap Write Barriers

Like generational collectors, subheaps require a (compiler-inserted) write barrier to notify the collector of changes made by the mutator. Specifically, the subheap write barrier is responsible for identifying all subheap-crossing pointers at program run time. One source of such pointers, shared with generational collection, is explicit mutation by the mutator. But the subheap write barrier faces an additional source of potential subheap-crossing pointers: initializing writes of heap object slots.

Consider the venerable `Cons(x,y)`, which writes the pointers `x` and `y` into a freshly allocated object. A generational write barrier is unconditionally redundant: the cons cell is by definition the youngest object in the heap, and the generational barrier can only trigger when storing young pointers in older objects. In contrast, the subheap barrier is only redundant if the values being written currently reside in the subheap containing the cons cell (which is to say, the active subheap). This precondition will be true for some call sites and not others. So to fully eliminate such subheap barriers, the compiler must generate multiple variants of some functions. Doing so carries costs, both statically in compilation time, and dynamically at runtime due to instruction cache pressure.

Conversely, there are situations in which it is easier to eliminate subheap barriers than generational barriers. One example is array initialization—consisting of an allo-

cation for the array slots, followed by sequence of allocations and writes for the array elements. The stores to array slots can often be done barrier-free when evacuation has been ruled out. Subheaps merely have to prove that no subheap activations occur in between the array allocation and the stores into the array. A compiler trying to eliminate a generational write barrier must prove that none of the allocations preceding the barrier could have triggered a nursery collection, because generational collectors usually evacuate the nursery.

A common trick in systems programming is to give separate consideration to the "fast" and "slow" paths for a potentially-expensive operation. The fast path for the subheap write barrier simply establishes that the requested write does not create a subheap crossing pointer. The slow path updates the target subheap's remembered set. Static elimination of write barriers can reduce the number of "fast path" barriers executed, but cannot alter the number of "slow path" barriers, which is a function of the mutator's use of the subheap API. Programs which make sparing use of the subheap API will have the majority of their barrier costs spent in the fast path. Programs which make very fine-grained use of subheaps will often see the bulk of their barrier costs absorbed by the write barrier's slow path.

Barrier Implementation The write barrier for subheaps is listed in Figure 2. The fast path for the write barrier uses the `heap_for` helper function to compute the subheaps associated with the source and target objects. This helper simply reads and decodes the subheap from each object's header. For writes between objects within the same subheap, the subheap write barrier defers to whatever barrier the baseline collector needs (if any). With the two calls to the helper function inlined, the subheap barrier code compiles to an 8-instruction fast path on x86-64, presented as Figure 3.

```
void write_barrier(void* val, void* obj, void** slot) {
  *slot = val;

  if (!val) return;

  immix_space* hv = heap_for(val);
  immix_space* ho = heap_for(obj);
  if (hv != ho) { hv->remember_subheap(slot); }
  else { baseline_write_barrier_if_any(val, obj, slot); }
}

void immix_space::remember_subheap(void** slot) {
  // This is the slow path.
  incoming_ptr_set.insert(slot);
}
```

Figure 2: Subheap write barrier (C++)

```
        movq    %rdi, %rax
        movq    %rsi, (%rax)      # do the write itself
        movq    %rsi, %rcx
        shrq    $32, %rcx
        je      .after_wb         # no barrier for null pointers
        movl    -4(%rsi), %edx
        cmpl    -4(%rax), %edx
        je      .after_wb         # also skip intra-subheap writes
        xorl    %ecx, %ecx
        testl   %edx, %edx        # are we writing to
        sete    %cl               #   the default subheap?
        movq    %rsi, %rdi
        movq    %rax, %rsi
        movq    %rax, %rdx
        callq   subheap_write_barrier_slowpath
.after_wb:
```

Figure 3: Subheap write barrier (x86-64 asm), sans baseline write barrier

2.4.3. Subheap Barrier Optimization

The core function of the subheap write barrier is to preserve the invariant that each subheap can quickly identify incoming pointers from the rest of the heap. Given a write of object pointer P into slot S, a barrier is necessary if P and S might be located in different subheaps.

Thus subheap barrier optimization can be cast in terms of alias analysis (aliasing of objects' subheaps rather than the objects themselves). Alias analysis has been thoroughly explored by the static analysis research community.

The Foster compiler implements a coarser analysis focused on the current subheap; this obviates the need to symbolically represent subheaps. The core of barrier optimization relies on an interprocedural forwards dataflow analysis. Freshly-allocated objects are (by definition) located in the current subheap; the analysis maintains a set of objects in the current subheap (Ψ). Values returned from function calls are added to Ψ when the function only returns values in the current subheap. Stack slots are inserted to and removed from Ψ to match the values written. Calls which might activate a new subheap nullify the set of objects in the current subheap. A write of P into S needs a barrier unless P and S both appear in Ψ.

There are several sources of imprecision in the above algorithm, but more sophisticated analyses that would ameliorate some of the algorithm's shortcomings are well known. Most obvious is the restriction to the current subheap. Slightly more problematic are various forms of heap dependence, including closure environments and mutable heap cells. For closure environments, the challenge is to verify that no subheap activations occur between the creation and invocation of the closure. Mutability, of course, introduces the challenge of aliasing.

The return value of functions can be a source of overapproximation. Consider the example snippet of Figure 4. The function to compute a prefix of a byte sequence is defined by cases. One of those cases (in which the supposed prefix happens to be longer than the original sequence) can simply return the input unchanged; all other cases allocate a fresh object to represent the newly computed sequence. Because taking "too many" bytes is rare, most programs will only exercise the fresh-allocation code paths, but since the input could reside in any subheap, the overall function result is not guaranteed to be located in the current subheap.

Finally, the biggest conundrum for static optimization of subheap barriers is likely the handling of function arguments. Functions are naturally polymorphic over what subheap contains each of their arguments. Such polymorphism inhibits barrier elimination. In the general case, code duplication is needed to eliminate this implicit polymorphism. Unrestricted specialization risks exponential blowup in code size, even when confined to the special case of current-subheap-or-not. Ultimately, a function can have very many call sites with unique signatures for its function arguments' subheaps. Maximally specializing every call site invites large increases in code size, with negative implications both for compilation time and instruction-cache efficiency at run time. On the other hand, failing to specialize call sites introduces avoidable overhead in the form of barriers which will never trigger.

This tradeoff is more naturally handled by just-in-time (JIT) compiler infrastructures, which can specialize hot functions on demand. Ahead-of-time compilers could incorporate user-provided feedback, but relatively few programs are performance-critical enough to warrant the hassle of feedback-directed optimization.

```
bytesTake :: { Bytes => Int64 => Bytes };
bytesTake = { ba => len =>
  case ba
    of _ if len >=SInt64 bytesLength ba
                              -> ba
    of _ if len ==Int64 0     -> BytesEmpty
    ...
  end
};
```

Figure 4: bytesTake may return its input or allocate

Optimization Dilemma For (whole) programs which do not make any use of subheap operations, barriers can be safely disabled *en masse*. This observation eliminates superfluous barrier costs for programs which do not use subheaps. There is, however, a consequential wrinkle: introduction of subheap operations off the mutator's critical path can result in added overhead on the critical path, due to barriers inserted by conservative static analysis. This is an unfortunate dilemma between gratuitously slowing down programs for the sake of consistency, or introducing a small but silent potential performance regression from the introduction of subheaps.

2.5. Generational Variants

Generational collection is a widely adopted design point across serial, concurrent, and parallel collectors. The primary motivation for subheaps is to provide control for tackling program patterns which generational collectors struggle with. However, the benefits of subheaps should not come at the cost of sacrificing generational collection.

Generations and subheaps are clearly related: both partition the heap to boost collection efficiency, and both rely on infrastructure such as remembered sets. Yet there are important differences between generations and subheaps. First, nurseries are usually evacuated at each collection. Evacuation both enables direct bump allocation and permits bulk clearing of the remembered set. Copying objects between subheaps com-

plicates static reasoning about which subheap an object lives in, thereby undermining barrier optimization for compilers and static reasoning about collection performance for humans. Movement of objects between frames within the same subheap is less problematic; it merely imposes overhead to update the target remembered sets of any subheap-crossing pointers in moved objects.

Second, generational collectors impose a strict collection order: every major collection must be preceded by a minor collection. This ordering allows for unidirectional rather than bidirectional remembered sets. In contrast, subheaps use bidirectional remembered sets and allow collection of arbitrary subheaps in arbitrary orders. (Section 2.7.1 describes a scheme which eases this invariant, enforcing a partial collection order to reduce the cost of remembered set maintenance.)

Luckily, evacuation is merely conventional, not a strict requirement for generational behavior. Existing research on the Immix heap organization [BM08, SBYM13] shows that an Immix variant based on the sticky mark bit design of Demers et al [DWH$^+$90] performs "very competitively" with a more traditional evacuating nursery. Shahriyar et al [SBYM13, Sha15] also describe a scheme for deferred reference counting in an Immix setting, with performance characteristics strongly reminiscent of a sticky mark bit design. We describe how both designs interoperate with subheaps.

The idea with sticky mark bits is to implement "nursery" collections by simply not resetting mark bits between collections. Since tracing ignores marked objects, this effectively restricts collection to examining only newly-allocated objects. The only remaining wrinkle is that a second remembered set must be kept; the barrier must catch writes of new (unmarked) objects to old (marked) objects. An additional optimization when mark state is kept "out of line" is to reflect mark state in object headers, reducing the working set needed during mutator operation. One bit from the header

designates "old" objects, with the invariant that all old objects are marked.

The conditions for the subheap barrier and the sticky mark barrier complement each other. The subheap barrier concerns itself with subheap-crossing pointers, and the sticky mark barrier need then only catch (a subset of) non-subheap-crossing pointer writes.

Shahriyar's reference-counting design for Immix, called RCImmix, also uses header bits to record object age and coalescing status. An old bit allows RCImmix to avoid tracking mutations to freshly-allocated objects (along with registers and the stack). For mutations to old objects, RCImmix uses the other bit to *coalesce* updates. Rather than explicitly tracking every overwritten pointer, as a naive reference counting design does, RCImmix captures snapshots of mutated old objects and sets the logged bit to avoid repeated captures. Doing so avoids redundant work arising from repeatedly mutated objects. At collection time, RCImmix: (i) applies increments to the root set; (ii) processes enqueued increment and decrement operations; and (iii) enqueues decrements for the root set. Note that step (ii) includes an increment for snapshotted objects, along with decrements for the captured fields.

In RCImmix, increment operations are recursively applied to new objects. Thus, new objects end up being traced, much like in the nursery of a generational collector. Decrements in RCImmix are recursively propagated when an object's reference count reaches zero. Rather than Immix's traditional on-the-side boolean mark byte for lines, RCImmix uses the mark byte to store a count of live objects within the line, updated by increments and decrements as new objects are found and old objects die. To deal with cycles, ImmixRC uses backup tracing, which re-computes reference counts and permits Immix-style opportunistic compaction.

ImmixRC's barrier has the same basic structure as with Sticky Immix; the most obvious difference is the object snapshot taken in the slow path.

Given that ImmixRC is based on reference counting, do we actually need to augment it with subheaps to perform well on workloads suited for reference counting, such as long-lived mutable caches? Indeed we do. The key is that due to deferral, we cannot immediately reclaim dead cache entries. Without subheaps, we are likely to trigger collection from heap exhaustion in the middle of allocating a new cache entry's object graph. At that point, we will apply the deferred reference counting operations, which will effectively trace through all newly allocated data since the last collection, as well as trace through whatever data has died since the last collection. Compared to a non-RC Immix collector, with or without sticky mark bits, ImmixRC does produce consistent GC overhead independent of heap size. This is beneficial in tight heaps but a net loss in more generously sized heaps. However, subheaps provide the possibility of eliminating GC costs entirely, which ImmixRC cannot. For an example of this phenomenon, see the evaluation of software caches in Section 4.4.

One ill effect that can be amplified by the combination of generational collection and subheaps is floating garbage and its repercussions. Floating garbage refers to dead objects that remain uncollected after a partial collection. Generational collection reduces total collector work in part by deferring the collection of old objects, thus giving them more time to die. However, when old dead objects contain subheap-crossing pointers, their delayed reclamation can retain garbage in other subheaps, raising costs in both space and time. Given that explicit subheap collections reset mark bits, programmers do have *a* tool to reduce or avoid floating garbage, but it is, admittedly, a blunt one.

2.6. Further Considerations

The discussion so far has laid the groundwork for an understanding of the pros and cons of subheaps in a somewhat abstract, simplified context. This section explores two concerns of relevance for real-world implementations of subheaps: how to deal with garbage cycles that span subheap boundaries, and how the subheap API should interact with non-linear control flow.

2.6.1. Subheap-Cyclic Garbage

A key point in space-incremental collectors, which focus on collecting one region at a time, is how to collect garbage cycles that cross regions. Some collectors (such as MC [SM03], MC^2 [SMB04], CBGC [Hir04], and most generational collectors) enforce a linear or partial order on region collections. This guarantees that cycles will be isolated within one full collection cycle. Garbage can persist between full collection cycles; such persistent garbage is referred to as *float*. The Train algorithm enforces restrictions on where objects can be evacuated, guaranteeing that garbage cycles will eventually be isolated to a single train, possibly after many collection cycles. In Klock's regional collector, a concurrent process marks logical snapshots of the heap and removes globally unreachable objects from remembered sets.

In other space-incremental designs, the choice of region granularity is determined by the algorithm or runtime. With subheaps, control over subheap granularity ultimately falls to the programmer. This means that subheaps can be coarsened (by the programmer) to avoid the problem entirely. Large remembered sets and cyclic garbage in remembered sets are, in some sense, symptoms of a mismatch between the structure of the application's heap and a pre-chosen, application-ignorant heap structure. Subheaps provide a means by which the runtime's heap partitioning can

better reflect the application's needs.

Second, subheaps give the programmer control over the precise subset of the heap collected at a given time. If the programmer knows that cyclic garbage has been created that spans multiple subheaps, the programmer can condemn multiple specific subheaps to eventually be collected simultaneously. In the (likely?) case that the overall subheap data doesn't have the same lifetime as the cyclic portion, the cyclic portion cannot be quickly and efficiently reclaimed by scanning multiple subheaps simultaneously.

Maintenance of a points-into set allows cycles to be identified without a whole-subheap scan. Differentiating garbage cycles versus non-garbage cycles requires a full scan.

This issue—of handling cycles between regions—is one of the key fundamentally hard problems in garbage collection. Arguably, the solution most in line with the spirit of subheaps is precisely to let humans shoulder some of this burden, when it makes sense to do so. Under the view of subheaps as a mechanism driven by human intuition, appeal to deliberate choice of subheap granularity might be an acceptable (if not ideal) design tradeoff. However, if one looks to a more automated future, in which subheaps are more of a reification of an automated analysis, then either the subheap runtime or the analysis must be prepared to prevent or handle the threat of subheap-cyclic garbage.

2.6.2. Control Flow Interactions

In a language with only simple control flow constructs (such as loops and function calls) the subheap API can be used to construct higher level abstractions with strong guarantees. For example, here is Foster code implementing an operation called `inTempSubheap`:

```
inTempSubheap = { thunk =>
  newSubheap  = subheapCreate !;
  prevSubheap = subheapActivate newSubheap;
  result      = thunk !;
  subheapActivate prevSubheap;
  subheapReclaim newSubheap;
  result
};
```

This code captures the allocations performed by the given function into a fresh subheap, which it collects after the function returns. Cleaning up after a function call in this manner can be very efficient when few objects survive (like regions or stack allocation), while still safely allowing for survivors (unlike regions or stack allocation). More importantly, this code provides a strong invariant: the temporary subheap is completely encapsulated, so the caller of `inTempSubheap` will never see the active subheap change.

However, many languages provide more advanced forms of control flow—such as exceptions, async/await, coroutines, effect handlers, and continuations—which complicate the quest to provide strong abstractions atop the subheap API. For example, if `thunk` throws an exception in `inTempSubheap`, the restoration of the previous subheap may be skipped. This is an instance of a well-known tension: state (such as the active subheap) is harder to reason about in the presence of complex control flow.

Most languages provide features, such as `finally` blocks in Java or `unwind-protect` in Common Lisp, which can be used to restore robust state management. For some control primitives, such as continuations and asymmetric coroutines, enforcement of invariants must be done individually. For others, such as exceptions and effect

handlers, the language can automatically enforce a scoped discipline on subheap activation, thereby lowering the burden on the user of the subheap API.

2.6.3. Whither Withered Pointers

Many languages offer one or more forms of *weak reference*. The basic idea of a weak reference is to refer to an object without preventing it from being collected. Weak references[4] can give programs improved control over size-adaptive caches, canonicalization tables, and finalization [JHM11]. How do subheaps interact with weak references?

Semantics The intent of the condemned set abstraction is to provide programmers with a clear idea of what subset of the heap will be inspected during a collection. Some forms of weak reference carry "full heap" semantics that are in tension with the local nature of subheaps. In particular, Java specifies that when a weak reference is cleared, all other equal-strength references referring to it must also be atomically cleared [UJR14]. The purpose of this restriction is to ensure that the mutator cannot observe inconsistencies in reference state.

Clearing of Soft references in Java is at the behest of the runtime, meaning it would be legal for an implementation of subheaps to only clear Soft references during full-heap collections. However, Java also provides Weak references with similar atomicity requirements and for which clearing is non-optional. Thus, to be in full compliance with Java's semantics for Weak references, an implementation of subheaps would need to track reference strength in remembered sets, as well as keep a Reverse Reference Table as used by reference-counting collectors [JHM11].

[4] As a general term, encompassing Java's Weak, Soft, and Phantom references, plus constructs such as ephemerons [Hay97] in other languages.

Performance On the one hand, reference processing requires additional processing passes, which can inflate the cost of collecting the condemned set. On the other hand, we would expect most explicit collections to contain mostly or entirely dead objects; in such situations, the cost of reference processing would be minimal.

2.6.4. An Accounting of Costs

Subheaps come with both direct and indirect costs. These costs are mostly mentioned elsewhere in this dissertation, but are presented here in a unified list for clarity. This following considerations are sorted roughly in ascending order of importance:

- The tracing loop must do a little extra work to check objects against the condemned set. This is a very small cost in practice; note that on modern processors, extra instructions do not always result in more cycles.

- Allocating objects becomes (very slightly) slower due to the need to add subheap identifiers to object headers.

- Subheaps *de facto* require that the underlying compiler be safe for space complexity [ADM98, SA00] so that dead bindings at the source level—especially in loops—are translated to dead roots at runtime. The analysis required for optimal placement of root management requires both forwards and backwards dataflow information. This makes the problem more technically interesting but also less amenable to off-the-shelf dataflow analysis infrastructure. Static analysis and function cloning for subheap barrier removal adds additional compiler complexity.

- Remembered set processing must account for stale entries, necessitating a choice between conservatism or line stamps (see Section 2.4).

- The subheap write barrier is more costly than a generational barrier.
- Static elimination of write barriers *de facto* forbids intra-subheap evacuation.
- Careless use of subheaps can result in arbitrarily high performance overheads due to repeatedly performing unexpected tracing work. (Evacuation could reduce this overhead in some situations).
- The combination of per-object and per-line segregated metadata imposes roughly 8.6% space overhead.
- Remembered set maintenance costs (which depend entirely on the user-chosen subheap configuration) can outweigh the collection-time savings of subheaps.

2.7. Refinements of the Subheap API

The basic subheap API given in Section 2.3 suffices for most uses of subheaps. It can, however, be extended to reduce reliance on ad-hoc heuristics or reify existing functionality in order to improve performance. Such seemingly simple supplements surface surprising subtleties.[5]

2.7.1. Subheaps for Temporary Data

Maintenance of remembered sets provides independent collection of subheaps. This can be broken down into dual benefits. First, data outside of a subheap can be ignored when collecting the subheap. This improves efficiency when data within the subheap is dead and data outside of the subheap is live. Conversely, data within a subheap can be ignored when collecting the remainder of the heap. This permits avoiding wasted effort in repeated collection of (subheap-internal) data known to be live.

[5] sadly sometimes sans solutions

Although their phrasing above is intentionally similar, the applications of these principles in practice sees larger differences. The first benefit effectively reduces the overhead of reclaiming known-dead space; instead of needing to trace the entire heap to establish deadness, collection can instead examine the (much smaller) set of remembered pointers. The second benefit can be obtained by corralling long-lived data in a separate subheap, avoiding repeated tracing when the main body of the heap is collected.

The simple presentation of subheaps given so far conflates these two benefits. However, in some situations, only one of the two benefits is needed for a given subheap. Consider a subheap created to hold short-lived temporary data. Recording incoming pointers is needed to allow efficient reclamation of the data within the subheap once it dies en masse. However, we may have no real need to collect other subheaps independently; if that is the case, then any outgoing pointers recorded from the temporary subheap are superfluous.

Relaxing remembered set maintenance can enhance performance. Given two subheaps, A and B, we need not remember any incoming pointers into A from subheap B as long as subheap B is always collected whenever subheap A is. Note the asymmetry: B can still remember pointers from A, and thereby retain support for collecting B independently of A. Exploiting this asymmetry can improve performance (by reducing remembered set maintenance burdens) in situations such as the temporary subheap example. In practice, the most important relationship is between the default subheap and each individual non-default subheap. Extending the precedence relation to arbitrary pairs of subheaps is possible [Hir04] but we leave such exploration for subheaps to future work.

To capture this opportunity, we can introduce a new type of subheap, made available

via an addition to the subheap API, to hold short-lived data. Short-lived subheaps do not contribute to the default subheap's remembered set, and in turn must be collected whenever the default subheap is. Concretely, the subheap write barrier (see Section 2.4.2) gains one small expansion in the fast path: writes into short-lived subheaps of values from the default subheap may skip remembered set maintenance.

The addition of short-lived subheaps paves the way for the runtime to make a more principled selection of "default" condemned set: the active subheap, plus any short-lived subheaps.[6] The remaining subheaps have been implicitly designated by the programmer as holding long-lived data, and avoiding their repeated collection can reduce total collector work.

2.7.2. `subheapOf` *and Subheap Equality*

When tracing, the subheap associated with each newly-encountered object is looked up, in order to ascertain the object's condemned status. Adding this operation to the core API involves no further implementation burden, and can avoid the need to explicitly pass around `Subheap` objects in some cases.

The details of how to best encapsulate the object-to-subheap mapping—and how to handle edge cases—are not entirely obvious. The root of the issue is a mismatch of abstraction layers: programming languages deal in values rather than heap objects. Untyped languages provide values such as `nil` or `undefined`, which do not have any clearly associated subheap. In typed languages (such as Java) which distinguish between primitive types like `int` and pointer-represented types like `Object`, the `subheapOf` operation can be given a type that restricts it to values represented with pointers. But this still leaves the question of how to handle `null` pointers (or, in

[6] Programs which do not create any short-lived subheaps can use the full-heap heuristic.

functional languages, values created from nullary constructors). Returning a default value such as the empty subheap makes for a more forgiving interface. Enforcing partiality (such as by throwing an exception or returning a `Maybe` value) is more "honest" but makes the operation harder to use robustly.

From the perspective of the GC implementor, either of these choices is reasonable, and are equally easy to accomodate. The issue is of more concern for compiler writers and others who care about language semantics and metatheory. In particular, is it legal for program transformations (which might change the representation of values) to change which subheap is returned by any given call to `subheapOf`? Consider the `Maybe` type constructor, values of which are either `None` or `Some` value. Although a naive language implementation would heap allocate values of type `Maybe`, it has long been recognized that using a null pointer to encode `None` allows the compiler to elide allocations. However, the `subheapOf` operation provides a means to (potentially) detect this optimization. A very similar issue arises for the question of giving subheaps observable identities.

The subheap API is defined in terms of values of type `Subheap`. One question implicitly raised by the subheap API is whether it is legal to perform equality comparisons (such as with the `==` operator) between subheaps. Forbidding such equality checks places draconion restrictions on the use of subheaps as first-class values. For example, it would not be possible to construct a `Set` of subheaps.

The issue of implementation details (such as object representation) leaking to the language semantics is amplified by subheap equality, which provides a direct means to observe differences between given `Subheap` values. The combination of `subheapOf` and subheap equality also unlocks new opportunities for mischief. For example, in many languages in the ML family, function values cannot be hashed or com-

pared for equality. So you cannot write `f == g`, but with subheaps you could write `subheapOf(f) == subheapOf(g)`, thereby distinguishing previously indistinguishable values.

2.7.3. A Failed Experiment

An earlier iteration of the subheap prototype avoided storing subheap identifiers in object headers and instead associated it with per-`frame` metadata. In this design, each frame held objects from only one subheap.

To avoid imposing unavoidable internal fragmentation for small subheaps, this design was in turn extended with a scheme for a new kind of subheap to allow intermixing subheap ownership at line granularity. With a line size of 256 bytes, reserving five lines (1280 bytes) per frame allowed recording a subheap pointer for each line (plus some extra metadata), for an additional space cost of 3.9%.

The most obvious tradeoff for the increased space efficiency was higher overhead for collection and allocation, both of which deal with the additional metadata for line-granularity subheaps. Another shortcoming was that this design for subheaps relied on the programmer to decide a priori whether a given subheap should optimize for fast allocation or low space overhead. Implicitly, this assumed that programmers can identify (at least in hindsight) which subheaps will have low expected space utilization.

A less obvious downside was the potential for induced fragmentation between, rather than within, subheaps. Because line-granularity subheaps need disjoint space from "full" subheaps, a program could encounter heap exhaustion even when there is ample free space available (for line-granularity allocations).

CHAPTER 3 : Using Subheaps

The previous chapter covered the big-picture principles behind subheaps, and also explored lower-level details of the subheap API design and implementation. These are *why* and *what*, respectively. This chapter addresses the use of subheaps: *where*, *when*, and *how*, including concerns such as modularity and debugging.

3.1. "Hello, World"

We begin by illustrating the line-by-line application of subheaps to a venerable garbage collection microbenchmark. The `binarytrees` program [Gou18], a descendent of Hans Boehm's GCBench [Boe], creates and discards a series of binary trees while also holding a reference to a long-lived tree. It is intended to coarsely approximate a generic generational-collector-friendly workload.

Figure 5 shows Foster code implementing the core of the binarytrees microbenchmark. There are four primary allocation sites in this code. Line 7 allocates a "stretch" tree whose purpose is to avoid any overhead from growing the heap during the remainder of the microbenchmark. Line 10 allocates a long-lived tree, which increases the tracing workload for non-generational collectors. Lines 15 and 16 allocate short-lived trees, which (once fully built) are immediately consumed by the `check` function.

Even with this simple computation, there are multiple ways to apply subheaps. The first question to answer is: how many subheaps should be created? The short-lived and long-lived trees should be kept separate in order to allow minimal-cost reclamation of the short-lived trees' memory. This can be done with two explicitly created subheaps, though only one is needed, as the long-lived tree can be kept in the default subheap. Because the data at hand is self-contained, the difference between these

```
type case Tree                                                                  1
  of $TNil                                                                      2
  of $Node Int32 Tree Tree;                                                     3
                                                                                4
benchmark = { n : Int32 =>                                                      5
    maxN = if n >=SInt32 6 then n else 6 end;                                   6
    stretchN = maxN +Int32 1;                                                   7
    c = check (make 0 stretchN);                                                8
    io "stretch tree" stretchN c;                                               9
    long = make 0 maxN;                                                         10
    minN = 4;                                                                   11
    REC depth = { mn => mx =>                                                   12
        REC sumT = { d => i => t =>                                             13
            if i ==Int32 0 then t else                                          14
                a = check (make i d);                                           15
                b = check (make (0 -Int32 i) d);                                16
                ans = a +Int32 b +Int32 t;                                      17
                sumT d (i -Int32 1) ans                                         18
            end                                                                 19
        };                                                                      20
                                                                                21
        if mn <=SInt32 mx then                                                  22
            n = bit ((mx -Int32 mn) +Int32 minN);                               23
            i = sumT mn n 0;                                                    24
            m = 2 *Int32 n;                                                     25
            iot m "\t trees" mn i;                                              26
            depth (mn +Int32 2) mx                                              27
        end                                                                     28
    };                                                                          29
    depth minN maxN;                                                            30
    io "long lived tree" maxN (check long);                                     31
};                                                                              32
                                                                                33
make = { i => d =>                                                              34
    if d ==Int32 0                                                              35
        then Node i TNil TNil                                                   36
        else i2 = i *Int32 2;                                                   37
             d2 = d -Int32 1;                                                   38
             Node i (make (i2 -Int32 1) d2) (make i2 d2)                        39
    end                                                                         40
};                                                                              41
                                                                                42
// check :: { Tree => Int32 }  computes a checksum of a Tree.                   43
// bit   :: { Int32 => Int32 } returns an int with the n'th bit set.           44
// io and iot are helpers to print tree depth and checksums.                    45
```

Figure 5: Foster code for binarytrees

two choices is purely stylistic. In a more realistic application, the opportunity for isolation in a separate subheap must be weighed against the potential consequences of creating subheap-crossing pointers.

The second and more consequential decision is: where should explicit collections take place? One choice would be to collect the short-lived trees once per iteration of the `sumT` loop. Colocating allocation and collection points makes it easy for a human reader to follow along. There are four potential points in the code where a collection could be added to implement this policy: after lines 13, 14, 16, or 18. Adding code after line 18 is a non-starter because it would mean that `sumT` no longer forms a tail-recursive loop. Adding a collection after line 14 would correspond to collecting before running each loop iteration. This can ensure that the target subheap is empty before allocating into it, but it leaves data from the last iteration of the loop un-collected. Collections on lines 13 or 16 will clean up after each iteration. Putting the collection after line 16 is arguably clearer.

Another choice would be to separately collect each subtree allocated in `sumT`. Doing so may lead to improved mutator cache behavior from keeping a smaller working set, but those benefits are unlikely to outweigh the doubling of collection costs relative to the once-per-loop approach.

Alternatively, multiple small trees may be allowed to accumulate between collections. This policy could be implemented by conditional collection inside the `sumT` loop, or by moving the collection point to occur in the (outer) `depth` loop rather than the (inner) `sumT` loop.

Collecting less frequently carries the opposite tradeoff: larger working sets but lower collection costs. Allowing subheaps to grow before collecting their contents leads to

fewer collections and thus lower total per-collection fixed overhead. In the extreme case, explicit collection is never triggered and the benefits of subheaps will not be realized. The primary risk of less-frequent collection is of inadvertently triggering an implicit collection and incurring the tracing costs which subheaps are meant to avoid.

This section's purpose is to highlight the existence of the choices faced by a programmer looking to apply subheaps to a concrete codebase. In practice, such choices will be resolved by a combination of intuition and experimental iteration. We defer examination of the quantitative consequences of these choices to Section 4.3.1.

3.2. Modes of Usage

How will programmers know where to modify their programs to use subheaps? The answer has to do with the *lifetimes* and *connectivity* of program objects.

Viewing the heap as a directed object graph, an efficient subheap configuration will find one or more *partitions* which both contain objects of similar lifetimes and form small cuts with the remainder of the graph. The partitionings need not form strictly minimal cuts in the graph theoretic sense, but too many subheap-crossing edges will result in high costs for remembered set maintenance. Note that by definition, any reachable partition will have one or more incoming references, but it need not have any outgoing references. Partitions with no outgoing references are prime candidates for subheap management.

Of course, this is a coarse mental model. The heap evolves over time, as object references are created and deleted. Even if a small cut exists at the time of collection, the cost of building the remembered set will have already been paid for by the mutator. Having objects of similar lifetimes within a subheap makes it easier to find collection points at which most or all objects within the subheap will be dead. Whenever

two connected objects have different lifetimes, a choice arises in whether to co-locate them within a subheap. Putting the objects in different subheaps incurs the cost of (dynamically) tracking the subheap-crossing references connecting the objects, a burden paid by the mutator. When co-located, however, the burden falls on the collector: prompt reclamation of the space for the shorter-lived object comes at the cost of tracing the longer-lived one. Alternatively, one can "spend" space to delay reclamation. This is a fundamental space-time tradeoff; subheaps provide tools for the programmer to make a choice as they see fit, but do not eliminate its existence.

The preceding paragraphs give a somewhat low-level view of the connection between programs and subheaps. Many programmers might find it easier to identify higher-level patterns of object usage within their programs. One common and broad category amenable to improvement with subheaps is *phased* behavior [WM89]. Phases are a common phenomenon in real-world programs, and the boundaries between phases are often where large volumes of objects from the previous phase die [XSaJ07]. Phases often, but not always, correspond to distinct program constructs like loops or function calls, which is why previous work has focused on such constructs [Har06, Cor06]. Examples of phases in real-world programs include server response loops [XSaJ08, XSaJJ07], compiler passes [BVEB07], and big data processing [NFX+16, BOF17].

The extension for temporary subheaps described in Section 2.7.1 allows additional knowledge of object lifetimes to be exploited. For example, a program with long-lived data can segregate that data in a (non-temporary) subheap, thus effectively instructing the GC to avoid tracing the long-lived data unless absolutely necessary.

Another stategy is to capture *domain-driven lifetimes* with subheaps. Examples include tabs in a web browser, entries in a cache, and memoized incremental subcomputations [HA08]. This category of allocation is particularly challenging for generational

collectors because object lifetimes are determined by external events and thus rarely satisfy the weak generational hypothesis. Section 3.5 explores this category, and its connection to Hayes' notion of key objects, in more detail.

3.3. Iterative Deployment & Debugging

When applying subheaps to a particular program, having a mental model is useful, but not always sufficient. The details of object lifetimes and connectivity can be surprising even when investigating a relatively small and familiar piece of code. Faced with a larger unfamiliar code base, reading the code itself is an inefficient way to ascertain the broad strokes of the program's heap usage. Instead, it is much easier to have the runtime convey information to help guide the deployment of subheaps.

Such feedback can come in several different forms to answer several corresponding questions. One simple question is: will a given activation of subheaps capture the majority of a program's allocations? Object profiling can help identify certain parts of a codebase as being more allocation-heavy, but higher order code can make it harder to leverage profiled information with subheaps. A simpler and more direct mechanism is to have the runtime report what proportion of allocations are captured by non-default subheaps. This allows the programmer to verify that they are indeed capturing as much data as they expect to.

Another key question is: will a given deployment of subheaps actually lead to an improvement in GC costs? Programmers would appreciate direct feedback on collection efficiency to confirm or refute their hypotheses about appropriate points to trigger subheap collection. Even though the end goal is to reduce wall-clock execution times, direct measurement of this metric has several downsides. Even for deterministic programs, wall-clock time results are often not deterministic. Hardware performance

counters can count elapsed cycles with low variance, but even these metrics are usually not completely deterministic. The cost of such variance is longer iteration loops due to the need to run programs multiple times to establish statistical significance. Machine-independent statistics, such as mark/cons ratios and percentage of allocations captured in (non-default) subheaps, can provide fully deterministic reflections of the impact a given subheap configuration has, and are thus invaluable in the process of iteratively applying subheaps.

Note that a mark/cons ratio alone does not account for every GC-related cost. In particular, remembered set maintenance and stack scanning costs are not reflected in the mark/cons ratio. The subheap runtime can easily capture and report additional (deterministic) statistics such as remembered set entries created and traced to help programmers understand the implications of their chosen subheap usage. Overall, both deterministic and non-deterministic performance metrics are worth collecting.

Finally, extensions of the subheap API can improve the programming cycle. For example, when a programmer initiates a collection, they often have an expectation of whether that collection will need to do any tracing work. When such assumptions are violated, the performance impact of subheaps can quickly shift from a boon to a bane. Even with a variety of robust metrics as described above, currently such expectations must be manually verified in testing. This is both labor-intensive and potentially fragile as codebases and allocation patterns evolve. A more robust solution would be to allow the API to capture (and check) such beliefs. A separate improvement would be to assign human-readable labels to particular subheaps; this would have obvious benefits for the clarity of debugging feedback.

3.4. Modularity

Subheaps occupy a middle ground between the fully automated nature of traditional garbage collection and the fully manual nature of `malloc()`/`free()`-style memory management. Although manual memory management provides a great deal of power and flexibility to the programmer, it also brings inconvenience and a tendency towards immodularity. This immodularity comes from the need for disparate pieces of code to agree on, and cooperatively implement, a division of responsibility for reclaiming dead objects. A natural question then is: do subheaps bring with them the familiar drawbacks of manually freeing objects?

Some of the most problematic patterns found with manual memory management are mitigated with subheaps. For example, the dynamically scoped nature of subheap activation, combined with the implicit presence of garbage collection, enables callers to capture and manage the memory allocated by callees. With manual memory management, deallocation for the callee's allocations must be baked into either the caller or callee. With subheaps, the choice of where to direct the callee's allocations, and thus implicitly the assignment of responsibility for collecting the chosen subheap, can be made on a per-call basis.

Another key property of subheaps is that they can be deployed on an as-needed basis. While subheaps carry a variety of costs (in runtime overhead, programmer effort, code modification/distortion, etc), these costs need only be paid when the benefits from a particular use of subheaps outweighs the drawbacks.

However, subheaps are admittedly not inconvenience-free. A module which allocates objects of disparate lifetimes may need to activate several different subheaps during its execution in order to facilitate efficient reclamation of each subheap without

interference from unrelated objects. This can certainly be inconvenient. In some situations, a callee must change its signature to be explicitly parameterized by the subheaps to activate; this blurs the line between inconvenience and immodularity.

Particular uses of subheaps may also be forced to change data representations to "thread through" subheap handles to the appropriate place in their code. One concrete example of this phenomenon is in a cache, which maps keys to values. A natural data structure to represent this mapping is a hash table in which each bucket holds either a reference to a value or a null pointer. Each bucket's value should be held in a different subheap to allow independent collection. Without subheaps, pseudocode (in Foster syntax) to update a an entry might look like this:

```
updateCacheEntry = { table => key => valueGenerator =>
    entry = HashTable.lookupEntry table key;
    value = valueGenerator ();
    HashTable.setEntry table key value;
};
```

In the subheap-enabled version of this code, we must put each generated value into the subheap for the associated bucket:

```
updateCacheEntry = { table => key => valueGenerator =>
    entry = HashTable.lookupEntry table key;
    associatedSubheap = subheapOf entry;                //
    oldSubheap = Subheap.activate associatedSubheap;    //
    value = valueGenerator ();
    HashTable.setEntry table key value;                 //
    _ = Subheap.activate oldSubheap;
};
```

Of course, segregating the cache entries into separate subheaps is a means to an end, namely, the ability to efficiently collect the old cache entry when it dies. This requires explicitly clearing the cache entry reference so that the old value is recognized as being dead. Collection must be done before the new value is generated, otherwise the new value will be traced and collection will be inefficient:

```
updateCacheEntry = { table => key => valueGenerator =>
    entry = HashTable.lookupEntry table key;
    associatedSubheap = subheapOf entry;
    oldSubheap = Subheap.activate associatedSubheap;
    HashTable.setEntry table key NullEntry;            //
    _ = Subheap.collect associatedSubheap;             //
    value = valueGenerator ();
    HashTable.setEntry table key value;
    _ = Subheap.activate oldSubheap;
};
```

However, this code is still not correct. The problem was mentioned in 2.7.2: null entries are represented with a non-allocated value in most mature language implementations. Thus the call to `subheapOf` will work for established keys but fail when first installing a key's value. ("Fixing" this by heap-allocating null pointer values would presumably carry catastrophic performance overhead.) Instead, the cache implementation must be augmented to record each entry's subheap in a separate table:

```
updateCacheEntry = { table => subheaps =>
                                key => valueGenerator =>
    entry = HashTable.lookupEntry table key;
    associated = HashTable.lookupEntry subheaps key;  //
    oldSubheap = Subheap.activate associated;
```

```
        HashTable.setEntry table key NullEntry;
        _ = Subheap.collect associated;
        value = valueGenerator ();
        HashTable.setEntry table key value;
        _ = Subheap.activate oldSubheap;
    };
```

Alternatively, a single (immutable) table could map keys to pairs of subheap handles and mutable references to a cache entry. In either case, the need to route allocations to a particular subheap necessitated changes in the data representation.

3.4.1. Immodular Urges and Barrier Optimization

Some code patterns—see Figure 4 in Section 2.4.3—inherently lead to imprecision in the static analyses that drive barrier optimization. These sources of imprecision could be avoided by restructuring code to not violate the invariants that the static analysis tracks. While often possible in theory, such alterations would have significant costs in human effort and code clarity. As with most questions of compiler-driven optimization, there is some tension between performance and modularity. This tension derives from the need to optimize the subheap write barrier.

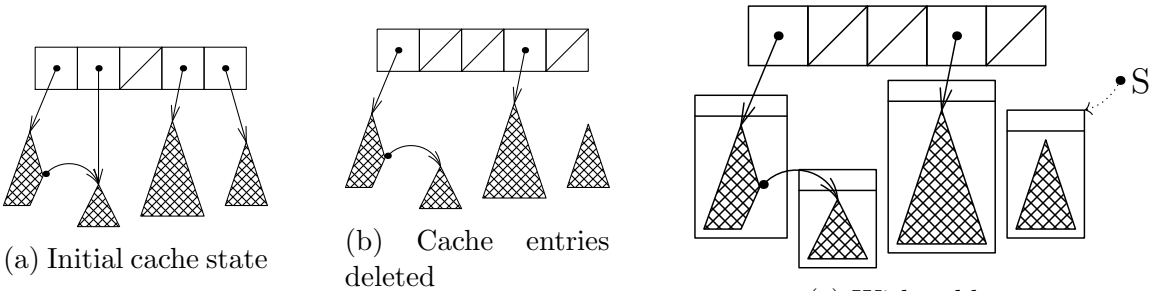
In contrast, there is little risk of programmers wanting to rearrange code for the sake of optimizing generational write barriers. The difference can be traced to three root causes. First, generational barriers are less pervasive, since they are not needed for initializing writes. Second, generational barriers are more difficult to remove, since doing so requires reasoning about when and where minor collections can be guaranteed to not occur. Finally, generational write barriers are significantly cheaper to execute than subheap write barriers; thus the reward for removing them is much smaller. As

a consequence of these factors, most work on optimizing generational write barriers has focused on making barriers cheaper to execute, rather than static optimizations to remove barriers.

3.5. "Reference Counting" with Key Objects

Prior work on memory management has focused on software caches due to their wide deployment in performance-sensitive environments, such as web servers and the cloud [TAV14, NGB16, PVV+17] . Caches are a prime example of how lack of control over memory management performance drives continued use of unsafe programming languages: Redis [SN18] and Memcached [Mem18] are both written in C.

Figure 6 illustrates an idealized version of such a software cache. Our cache initially contains four entries. The first entry references the second entry, but otherwise the entries are isolated. In Figure 6b, two entries are removed from the cache. The storage for one of the entries is unreachable by the program (dead), and the programmer knows it, but a simple tracing garbage collector cannot prove this fact without traversing the entirety of the heap—including the other (live) cache entries. When the cache occupies a large fraction of the heap, this spells disaster, because the small amount of free space will cause collection to be frequently re-triggered. As cache en-



(a) Initial cache state

(b) Cache entries deleted

(c) With subheaps

Figure 6: Simplified cache heap structure

tries are allocated and removed, the program will spend most of its time re-tracing the unchanged cache entries, over and over. The work is redundant, but the programmer cannot convey that knowledge to the garbage collector.

This example is a simplification of the memory management involved for programs like `redis` [SN18] and Memcached [Mem18]. Terei, Aiken and Vitek [TAV14] identify Memcached as representing a common, important class of real-world programs that pose significant difficulties for tracing collectors. Memcached maintains a set of key-value cache entries, with a client interface for additions, deletions, and key lookups. Such programs feature three elements that are difficult to simultaneously reconcile: large heaps, stringent latency demands, and client-driven object lifetimes that do not obey the generational hypothesis. Memcached is well suited to reference counting (or, of course, manual memory management). With reference counting, full-heap scans are not necessary, and each cache entry can be immediately reclaimed as soon as it becomes dead.

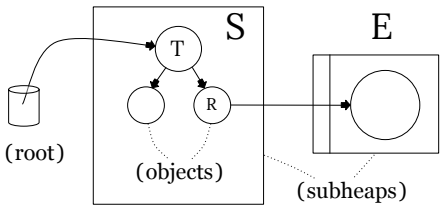
As in Figure 6c, when each cache entry is allocated in a separate subheap, all references pointing into the subheap will be tracked by the subheap's remembered set. Second, before a deletion overwrites or nulls out an entry's reference, we would first look up the entry's associated subheap. After the entry's reference has been annulled, the cache's deletion routine triggers a subheap reclamation. The subheap reclamation will inspect the remembered set. With a remembered set that retains slots (rather than pointer values), the first step of collection is to check that the slot contents still point into the subheap being collected. Since the slot contents were just annulled, the remembered set will be empty. Given that there are no references to the cache entry on the stack, the net effect is that the cache entry's subheap can be reclaimed without needing to trace the entirety of the heap.

This example is a special case of the idea of key object opportunism [Hay93]. Hayes theorized that the lifetimes of groups of objects could be inferred from the lifetime of a single key object within the group, rather than being tied to program phase behavior. To find key objects, he considered a variety of heuristics—such as random selection, stack reachability, and programmer hints—but his primary heuristic was to focus on those objects appearing in remembered sets.

If collection of a subheap is triggered once all subheap-external copies of the key object references have been deleted, then the subheap's contents can be reclaimed in bulk, and we'll have behavior somewhat akin to reference counting. Rather than counting copies of a pointer made as it is copied around the heap, as traditional reference counting does, we "count" incoming pointers at subheap-collection time, via the subheap's remembered set. This scheme resembles Bobrow's technique for managing cyclic structures [Bob80].

Immutability The cache example makes inherent use of mutability to establish the non-liveness of the cache entry in the remembered set. But not all languages support explicit mutability. Can this trick with subheaps work in pure languages? Let us re-enact the scenario with an immutable data structure.

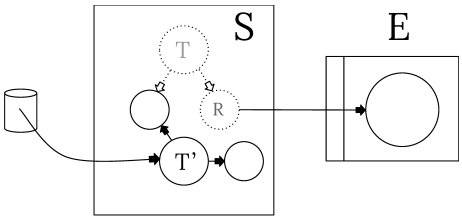
In our setup on the right, we have a binding for node T, which in turn has a right child R pointing to the cache entry in a separate subheap.



Now, consider what happens when we "remove" the cache entry from an immutable spine. Instead of overwriting any pointers, we allocate a new node T' which does not point to R.

Calling `subheapReclaim E` will be stymied by the remembered set entry for R, whose

death remains undiscovered. Perhaps surprisingly, calling `subheapReclaim S` first may not help! First, even though the R node is dead, the line it is on may still be used, preventing the remembered set entry from being trimmed. Second, reclaiming space does not alter its contents; to do so would shatter our performance goals. We cannot overwrite dead space without sacrificing the notion that the cost of collection can be independent of the amount of space reclaimed. Thus, even though R is known to be dead, its ghost can remain until overwritten by subsequent allocations in S.



If we trigger collection after condemning both S and E, the remembered set entry will be ignored, and both R and the cache entry will be eligible for reclamation. The downside is that if we try to reclaim multiple cache entries back-to-back, we will have to trace through the mostly-live contents of S multiple times.

A better solution is to condemn without immediately and explicitly triggering a collection. Having done so, there are two routes to initiate collections. The first option is to proactively trigger collections at a later point in time, either on the basis of elapsed work or elapsed time. The second option is to passively wait for a "regular" collection (of the condemned set) to be triggered due to heap exhaustion. Collecting on an as-needed basis minimizes duplicated collector work from re-scanning live condemned data, whereas proactive collections can schedule GC effort [DEE+16] to minimize interference with latency-sensitive periods of application work. This demonstrates the benefits from fine-grained control over *where* and *when* the GC expends effort.

Compare and Contrast A few differences from traditional reference counting are worth pointing out. First, the work to inspect remembered sets is less incremental

than manipulation of reference counts. Also, our "count" ignores intra-subheap pointers. One negative consequence is that this scheme, unlike precise reference counting, cannot be used as dynamic evidence of unique ownership. But we retain the on-demand reclamation of reference counting, and gain the ability to efficiently reclaim subheap-internal cycles—a perennial issue for traditional reference counting.

Bacon, Cheng, and Rajan [BCR04] explored the duality between tracing and reference counting. They observed that generational collectors, due to their remembered sets, are a hybrid of reference counting and tracing. With subheaps, that hybridization is put in the programmer's hands, to potentially significant benefit.

CHAPTER 4 : Evaluation

Subheaps provide a mechanism for leveraging explicit programmer cooperation to guide garbage collection effort. The goal of this cooperation is to improve the performance of tracing garbage collection on otherwise-problematic programs. This suggests several facets of evaluation for subheaps:

- **Performance** can be measured in several different ways, primarily throughput and latency. Many factors outside our scope—such as processor cache configuration and compiler optimizations—influence end-to-end program performance. Space usage must also be accounted for due to pervasive space-time tradeoffs. The core premise of subheaps is that careful usage of subheaps can improve performance (in some programs) by reducing GC work.

- **Usability** aids wide adoption; a system that can only be effectively used by experts will only achieve a small fraction of its potential influence. Usability has many incarnations: How often does effective usage of subheaps break modularity? Do subheaps interfere with ongoing maintenance of a codebase? How much background knowledge of GC is required to use subheaps? Does the difficulty of applying subheaps scale linearly, or perhaps sub-linearly or super-linearly, with the complexity of the underlying application? Questions of usability are of great relevance to the success or failure of subheaps "in practice," but are mostly outside the scope of this dissertation.

- **Applicability**: what programs have sufficiently high overheads to make usage of subheaps worthwhile? Domain-specific GC techniques are less widely useful than domain-agnostic approaches. Likewise, subbheaps will find wider applica-

bility if their usage can be encapsulated in libraries or other sub-components of whole applications.

These categories are not sharply defined. For example, applicability is predicated on (lack of) performance, which in practice depends on the "baseline" collector chosen. Other concerns cut across multiple aspects. For example, scaling up application complexity, as reflected in the choice between microbenchmarks and macrobenchmarks, has elements of both usability and applicability concerns.

Because subheaps are not a fully automated mechanism, there is **no singular characterization** of their performance. The same program can be modified to use subheaps in many different ways, with varying degrees of payoff *vs.* programmer burden. To shed light on what subheaps can achieve in practice will ultimately require a human subject research protocol. The evaluation in this section illustrates the potential performance gains from careful use of subheaps. Careless use of subheaps, on the other hand, can impose nearly arbitrary performance degradation; it is entirely the programmer's responsibility to avoid unprofitable modes of use.

This evaluation focuses on microbenchmarks and small applications, for two reasons. First, the development of subheaps for Foster impedes the direct reuse of existing benchmark suites. Second, the manual effort needed to use subheaps suggests that subheaps will be most successful for non-standard applications—such as big data analytics and caches operating in tight heaps. Traditional benchmark suites do not encounter the requisite crushing GC overheads. An evaluation of subheaps in a more traditional setting would of course help shed more light on the mechanism's strengths and weaknesses, but even this dissertation's more limited experiments reveals interesting phenomena.

4.1. Experimental Platform

Except where otherwise noted, experiments were run on a Core i5 6600K CPU running at 3.5 GHz[1] with 32 GB of DDR4-2400 RAM. Heap sizes for Java programs were specified with both `-Xmx` and `-Xms`. To keep cross-language benchmarks as fair as possible, pointer compression was disabled. Software versions: Ubuntu 16.10 with kernel 4.8; Go 1.10; OpenJDK 10 (with ZGC; HEAD commit 55e292c8ab9b); Foster and C++ code compiled with LLVM 6.0 and `-O2 -march=native` flags. In all cases, we measure wall clock elapsed time using the `perf` utility.

4.2. Conway's Game of Life

Gualandi and Ierusalimschy [GI18] identified a benchmark, based on Conway's Game of Life, as being GC-heavy. Its simplicity makes for a good introductory setting to analyze and dissect the application of subheaps in full detail. The main body of the benchmark driver is presented in Figure 7.

The benchmark allocates two scratchpads and initializes one, then loops, printing the current state of the board and updating the next based on the rules of the game. Building up a printable string is done via concatenation (rather than mutation), and thus requires substantial allocations to hold intermediate strings. Once the final string has been printed, the previously allocated strings immediately become garbage.

The size of the game board is configurable, and determines the rate of garbage production. However, somewhat counter-intuitively, increasing the size of the game board decreases the proportion of time spent collecting in minimally-sized heaps. This is because the amount of live data processed at each collection does not change, so the

[1] SpeedStep disabled in BIOS, and Turbo Boost disabled via the Linux kernel.

```
main = {
  luasteps = 2000;
  steps = luasteps -Int32 1;

  n = 40;
  m = 80;
  curr_cells = newcanvas n m;
  next_cells = newcanvas n m;

  glider = prim mach-array-literal
            (prim mach-array-literal 0 0 1)
            (prim mach-array-literal 1 0 1)
            (prim mach-array-literal 0 1 1);

  enumRange32 1 9 { i =>
    enumRange32 1 17 { j =>
      i0 = (5 *Int32 i) +Int32 1 +Int32 (j *Int32 j);
      j0 = (5 *Int32 j) +Int32 1;
      lifespawn n m curr_cells glider i0 j0;
    };
  };

  lifedraw n m curr_cells;

  REC loop = { gen => curr => next =>
    if gen <=SInt32 steps
      then
        lifedraw n m curr;
        print_text_bare "\n";
        lifestep n m curr next;
        loop (gen +Int32 1) next curr;
      else
        ()
    end
  };

  loop 0 curr_cells next_cells;
  lifedraw n m curr_cells;
};
```

Figure 7: Conway's Game of Life microbenchmark in Foster

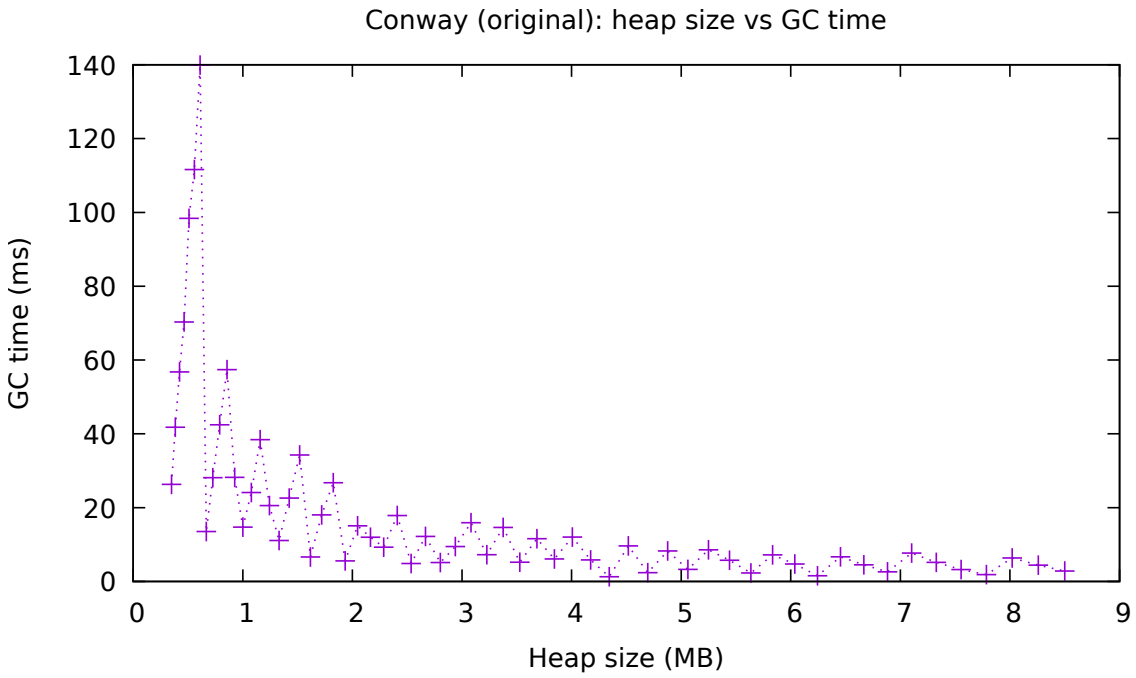
ratio of GC work to mutator work decreases as mutator work increases. Later graphs will explore this aspect of the microbenchmark's behavior.

For the configuration listed in Figure 7, with Foster's baseline Immix collector, the program triggers one collection per game step and spends 25 ms (5.1% of total execution time) in garbage collection.

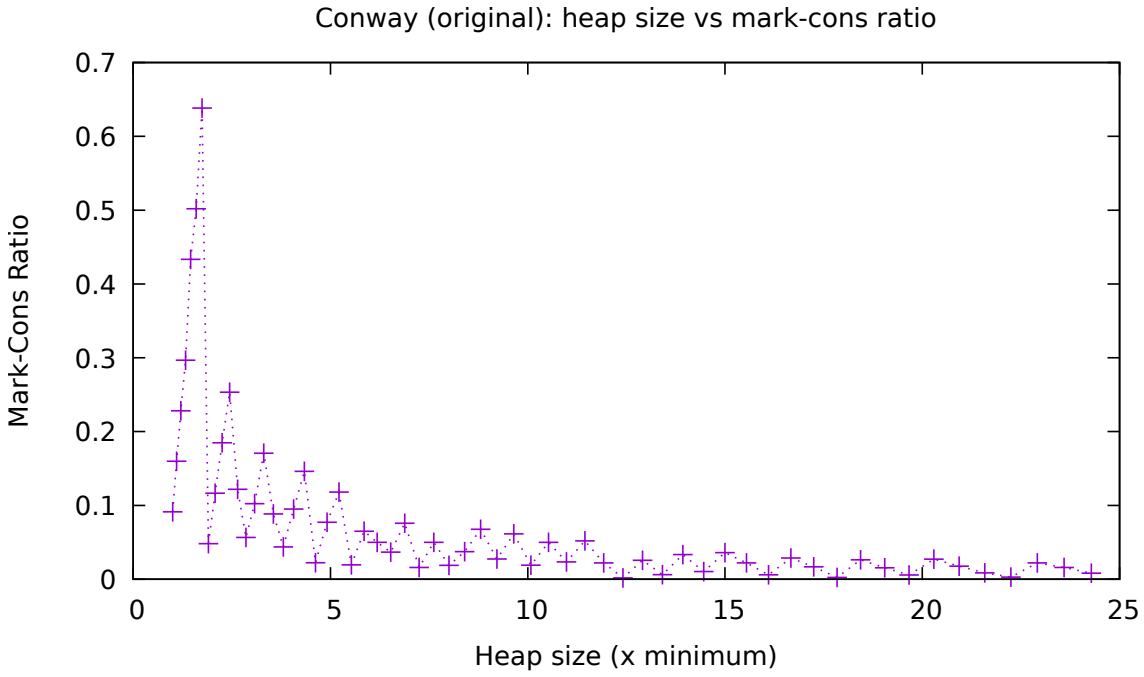
Figure 8 shows how collector workload varies with changes in heap size. The result is a noisy curve; Figure 9 decomposes those results into the combination of the number of GCs triggered (a smoothly decreasing curve) and the per-cycle observed live size (sampled from a periodic distribution). As Figures 8a and 9b show, small increases in heap size can make collection take more time rather than less. This is due to the way heap sizes interact with the periodic nature of allocation in this simple microbenchmark, causing collection to happen (on average) when more data is live.

Figure 11 lists the driver of the Conway benchmark after being modified to use subheaps. Figure 10 shows that using subheaps to consistently trigger collections of completely dead data, once per loop iteration, eliminates all tracing work regardless of heap size. Because no marking work is ever done, the mark/cons ratio is zero.

Doing no tracing work does not mean that GC is entirely free. For example, stacks must still be (repeatedly) scanned, and line marks inspected. However, both latency and GC work costs on this benchmark are reduced by an order of magnitude with the use of subheaps. Mean GC pause times decrease from 25.6 microseconds to 2.1 microseconds, and max pause times decrease from 296 microseconds to 17 microseconds. The net cost of memory reclamation is 5.4 ms, including the overhead of recording timing statistics.

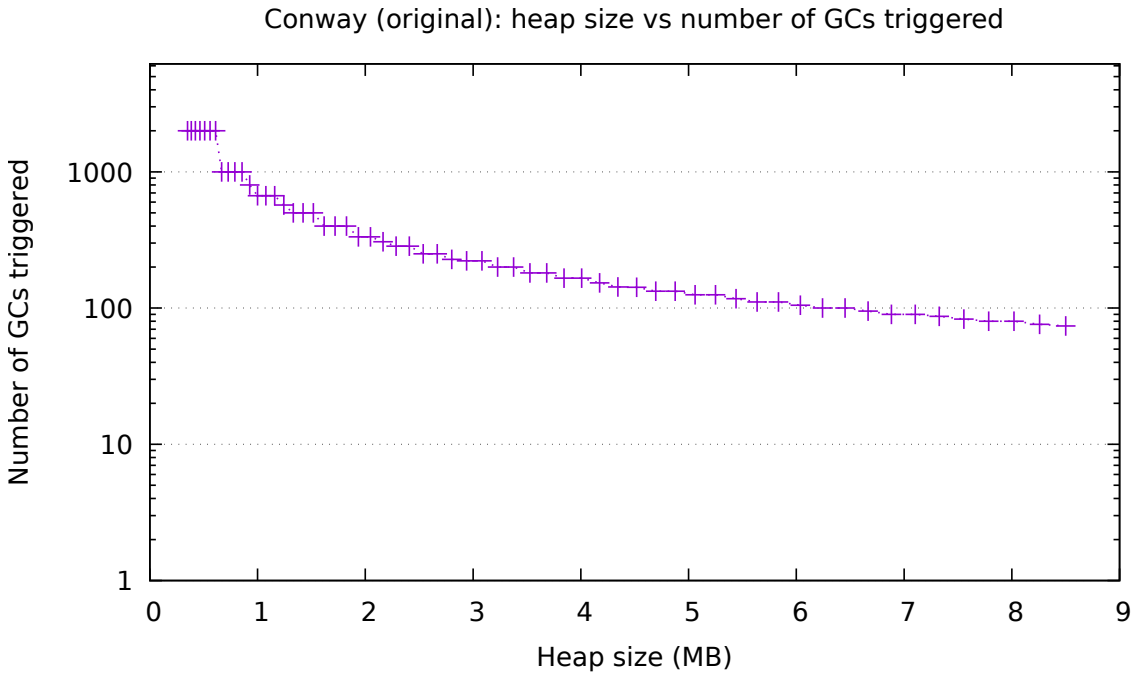


(a) Observed GC time as heap size varies: a spiky curve.

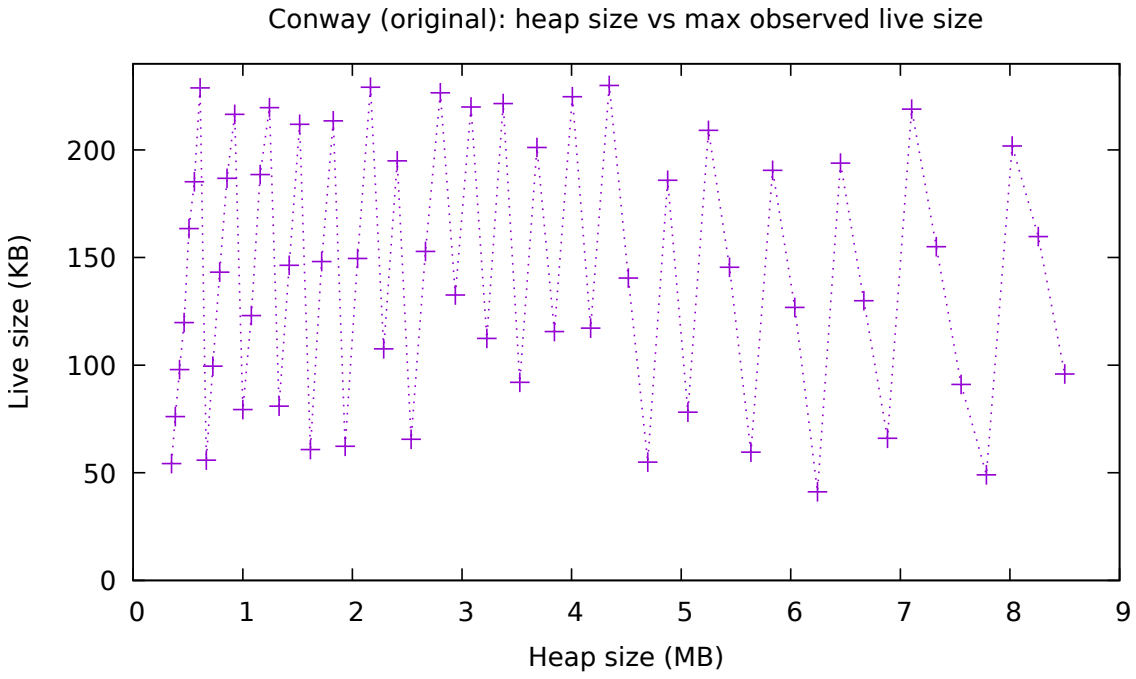


(b) Mark/Cons Ratio (Obj/Obj) almost perfectly mirrors observed GC time.

Figure 8: Conway (Foster) baseline results show close correspondence of (machine-dependent) wall-clock time and (machine-independent) mark/cons ratio.



(a) Number of GCs monotonically decreases (on log scale) as heap size increases. Note that the number of GCs is insensitive to changes in the size of very tight heaps.



(b) Max observed live size varies unpredictably with changes to heap size. Note that max observed live size is quite sensitive to changes in heap size. For a non-copying Immix collector, max and average live sizes coincide.

Figure 9: Conway (Foster) baseline results illuminating why Figure 8 is a spiky curve.

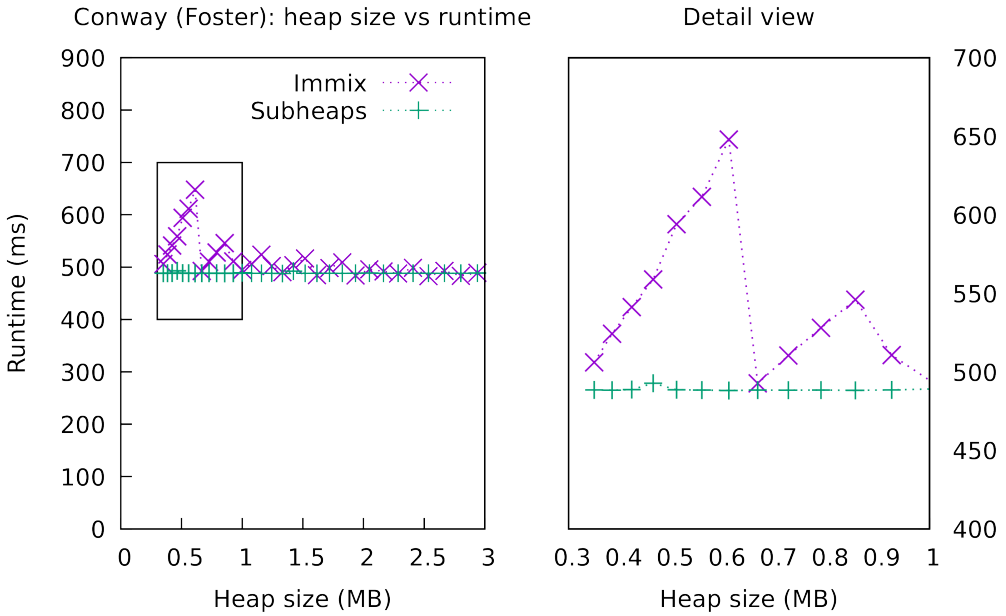


Figure 10: Subheaps eliminate tracing work from the Conway benchmark and produce a heap-size-insensitive performance profile.

More on Barriers The measurements listed were taken with the Foster compiler's subheap barrier optimizations enabled. Without static optimization of GC write barriers, the benchmark would see an increase of roughly one fifth in the number of barriers dynamically executed—from 26 million to 32 million. Given the small size of the benchmark, we can fully characterize the barriers remaining after optimization. There are twelve barrier sites in the generated executable:

- Six sites arise from text concatenation when building the output string to display, in the implementation of `lifedraw`. These barriers are not eliminated because the output string is built up in a mutable style, with a temporary string stored in a ref cell. The barrier analysis does not track mutable state and thus conservatively assumes that the barrier might be needed.[2]

[2]Mutable state is not the only challenge to removing these barriers. The optimizer would also have to switch from an in-current-subheap analysis to a subheap-aliasing analysis, because the long-lived scratchpads are kept outside the current subheap, which is used to collect temporary allocations.

```
main = {                                                                    1
  luasteps = 2000;                                                          2
  steps = luasteps -Int32 1;                                                3
                                                                            4
  n = 40;                                                                   5
  m = 80;                                                                   6
  curr_cells = newcanvas n m;                                               7
  next_cells = newcanvas n m;                                               8
                                                                            9
  glider = prim mach-array-literal                                          10
            (prim mach-array-literal 0 0 1)                                 11
            (prim mach-array-literal 1 0 1)                                 12
            (prim mach-array-literal 0 1 1);                                13
                                                                            14
  enumRange32 1 9 { i =>                                                    15
    enumRange32 1 17 { j =>                                                 16
      i0 = (5 *Int32 i) +Int32 1 +Int32 (j *Int32 j);                       17
      j0 = (5 *Int32 j) +Int32 1;                                           18
      lifespawn n m curr_cells glider i0 j0;                                19
    };                                                                      20
  };                                                                        21
                                                                            22
  sh = foster_subheap_create !;        // New subheap for                  23
  old = foster_subheap_activate sh;    // short-lived data.                24
  lifedraw n m curr_cells;                                                  25
  foster_subheap_collect sh;           // Clean up after lifedraw.         26
                                                                            27
  REC loop = { gen => curr => next =>                                       28
    if gen <=SInt32 steps                                                   29
      then                                                                  30
        lifedraw n m curr;                                                  31
        print_text_bare "\n";                                               32
        lifestep n m curr next;                                             33
        foster_subheap_collect sh;     // Clean up for next loop.          34
        loop (gen +Int32 1) next curr;                                      35
      else                                                                  36
        ()                                                                  37
    end                                                                     38
  };                                                                        39
                                                                            40
  loop 0 curr_cells next_cells;                                             41
  lifedraw n m curr_cells;                                                  42
};                                                                          43
```

Figure 11: Conway's Game of Life microbenchmark in Foster, using subheaps. Added lines are commented.

- One site traces back to a special case for concatenation of small strings, which copies data to increase locality and decrease pointer chasing. In this case, the issue is not mutable state but rather overapproximation due to a helper function. The helper function for array concatenation special-cases copying of zero-length arrays, and conditionally returns either a fresh array (which is guaranteed to be in the current subheap) or one of the input arrays (which is not). Thus the return value of the helper is not guaranteed to be allocated in the current subheap, and a barrier must be emitted in case the input array was not allocated in the current subheap.

- Two sites correspond to one-time initialization of a nested array (`glider`).

- Two sites arise from initialization of a closure's environment; the analysis driving barrier optimization is unable to prove that two objects closed over in the function's environment will be located in same subheap as the freshly-allocated environment record (namely, the current subheap).

- One site arises from initialization of a nested array in a loop (in the body of `newcanvas`).

It would be possible to extend the barrier optimization analysis to track properties, in limited cases, of objects stored in mutable state, albeit at significant cost in complexity. Code that tried to avoid the shortcomings of the barrier analysis in the face of mutable state might find itself stymied by another difficulty: tracking properties of higher-order functions. Faced with a loop combinator, a first-order analysis will note that it calls an unknown function—the loop body—and conservatively assume that any usage of the combinator could invalidate the set of known-current values by activating a new subheap. Thus, a robust analysis would have to handle both

mutable state and higher order functions. Finally, the overapproximation due to use of a function helper is even more difficult to resolve with enhanced static analysis.

Generational Collection To determine the difference in efficacy between Foster's baseline Immix collector and a generational collector, I implemented a simple copying-nursery generational collector for Foster (in addition to the sticky mark bits implementation used elsewhere).

Surprisingly, for the original version of the Conway microbenchmark, generational collection performs worse overall than the non-generational Immix baseline. The primary reason is that almost no objects survive multiple collections, as illustrated in Figure 9b, meaning the baseline Immix collector does not repeatedly trace many long-lived objects. Second, the cost of evacuating arrays is higher than simply marking them. Figure 12 examines the data in more detail, showing how performance varies with heap size for the generational collector plus three other Immix variants. This graph conveys several interesting pieces of information.

At the bottom of the graph is a line, labeled A, showing the ideal performance without any GC-related overhead. Next, B shows the performance of subheaps. In both cases, garbage collection is arranged to occur when no objects are live, and thus both lines are flat: heap size does not affect their performance. The delta between the two lines reflects the higher mutator costs from subheap allocation, primarily from the extra instructions from expanding objects headers from 4 bytes to 8 bytes, and from the requirement to set "used" line marks at allocation time.

The behavior of a copying generational nursery is exhibited with data points labeled C and D. The lower line, C, isolates the performance impact of resizing the generational nursery itself. It shows only the cost of copying data out of the nursery,

not including the necessary full-heap collections. The nursery can be smaller than the benchmark's live size, which is why this is the only line that starts at a heap ratio of zero rather than one. The upper line, D, shows the combined cost of nursery and full-heap collections when the nursery is fixed at one fourth of the total heap. The generational collector has a larger minimum heap size due to the requirement that the mature space keep a (conservatively-sized) free reserve for the nursery.

Finally, the behavior of the Immix collector, labeled E, shows the unusual saw-tooth behavior seen in Figure 8a. Note that the width of each tooth is equal to the minimum heap size, with the lowest (lowest overhead) points being at (or just beyond) integer multiples of the minimum heap size.

This phenomenon occurs because the microbenchmark does its work—including its allocation—in a loop. The difference between the smooth curve of the generational nursery and the spiky curve of the non-copying collector reflects how each loop iteration influences the next. With a copying nursery, any data live at collection time is evacuated, so the amount of available space left in the nursery is fixed. Mismatches in the size of the nursery and the per-loop allocation load give rise to periodic behavior in the triggering of nursery collections and, crucially, the amount of data copied out in each nursery collection. The net effect is that the many "modular remainders" induced by the delta between nursery size and loop workload produce a per-collection load average that is mostly insensitive to the size of the nursery. Thus the total cost of nursery evacuation, which is the average cost per event multiplied by the number of events, smoothly rises as the nursery shrinks because the number of (nursery) collections needed grows.

In contrast, the sawtooth pattern for the Immix collector reflects a different phenomenon. With the Immix collector, live data at collection time remains in place,

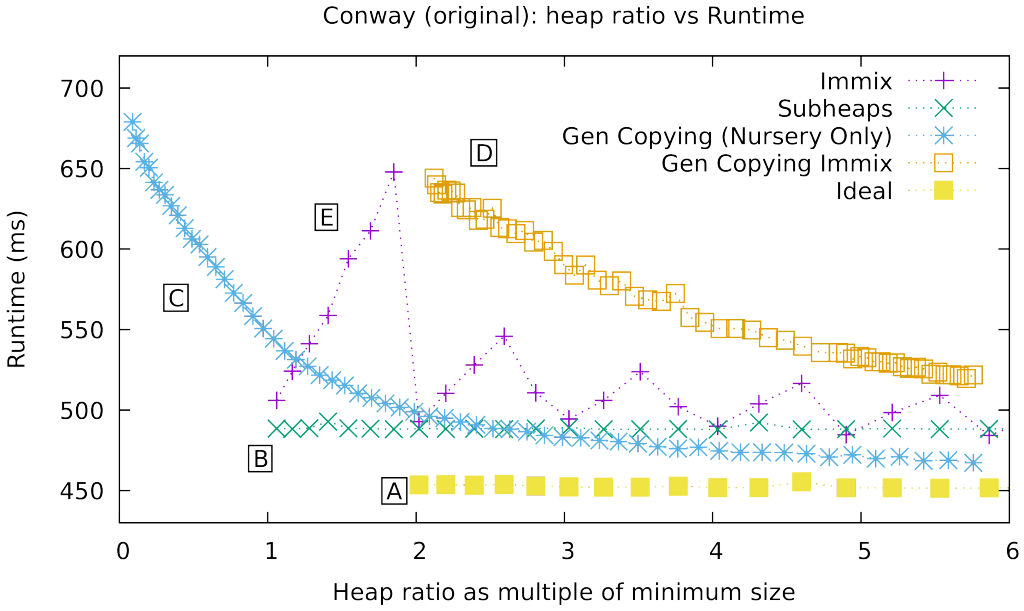


Figure 12: Detailed impact of generational collection on Conway.
Generational Immix provides no improvements over the non-generational baseline (but see Figure 13 for a more complete story). Note the restricted range of the y-axis.

thereby *reducing the amount of remaining space* to be used for the next loop's allocations. The pattern then repeats, until the amount of remaining space is a multiple of the loop's allocation load. At that point, a "stable attractor" has been reached, and the synchronization produces a consistent per-collection tracing load with little variation. This explains why each sawtooth's width is equal to the minimum heap size: because that is also the amount of data allocated in each loop iteration. The "excess" space beyond each multiple of the loop workload produces a consistent per-collection tracing burden. It also explains why the most efficient sizes are multiples of the minimum heap size. Regardless of the heap size chosen, the per-collection tracing load stabilizes quickly, usually within six collection cycles.

Ambient Heap Size In a variant of this benchmark, we keep a forest of objects live for the course of the program's execution. Doing so increases the benchmark's

realism, since most programs deal with a mix of short- and long-lived data. It also accentuates the difference in how subheaps enable a qualitative change in behavior in tight heaps.

There are now two input variables of interest: program heap size and ambient live data size. The larger the ambient data size, the greater the cost of each collection. The greater the proportion of heap size devoted to ambient data, the more frequently collection will be triggered. Figure 13 illustrates these effects. It shows that generational GC (whether in-place with sticky mark bits, or evacuating with a copying nursery) outperforms plain Immix with moderate amounts of live data, but does not avoid the trend of exponential increase in total GC cost. The fluctuation in the Sticky Immix results are not random variation or measurement error; they capture the same sensitivity to heap sizing as in Figure 12. These fluctuations are obscured for the non-generational Immix results by the log scale of the Y-axis and the higher costs incurred by repeated tracing of the ballast. As before, subheaps are unaffected by the presence of long-lived data.

Adding 5 MB of long-lived data significantly impacts both latency and throughput: mean per-GC latency for Immix rises to 3.02 milliseconds (from 12.1 microseconds before). Figure 15 shows bounded mutator utilization [SMB04] curves for three collector configurations on the Conway benchmark. These plots show the worst-case GC efficiency at varying time scales. The x-intercept, where mutator utilization first creeps above zero, indicates the largest observed pause. The y-intercept at the right of the graph shows the proportion of time spent on GC over the course of the whole program run. The shape of the curve in between these two points reflects the distribution of pauses incurred by the GC. Figure 15 illustrates how this benchmark sees a much larger impact on latency from subheaps than from generational collection.

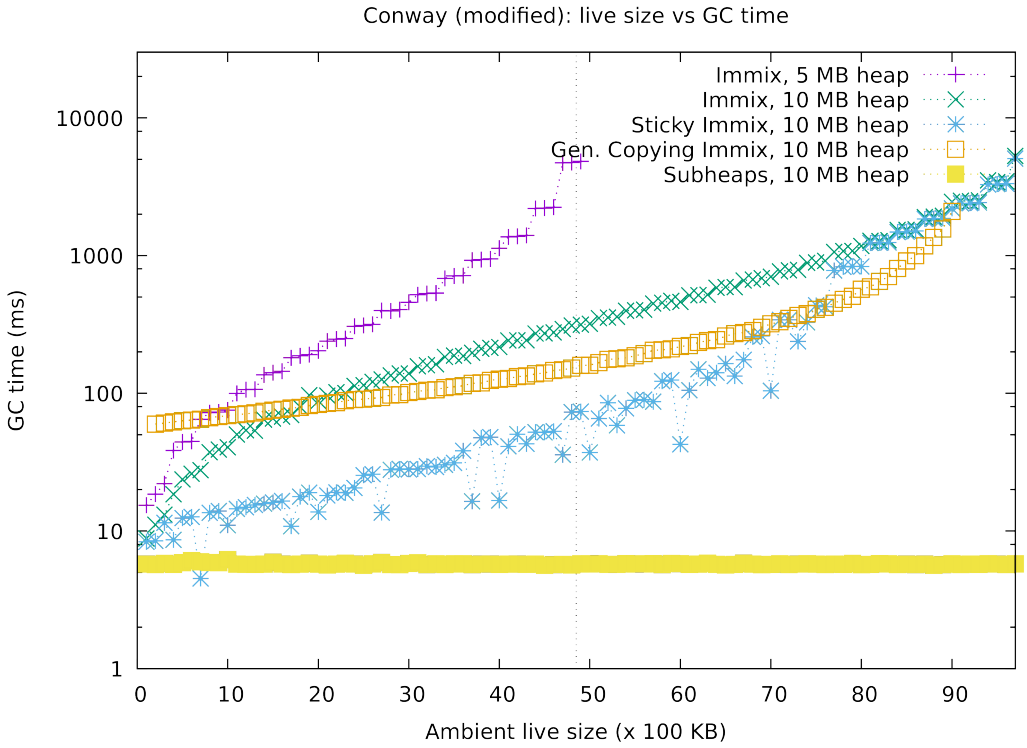
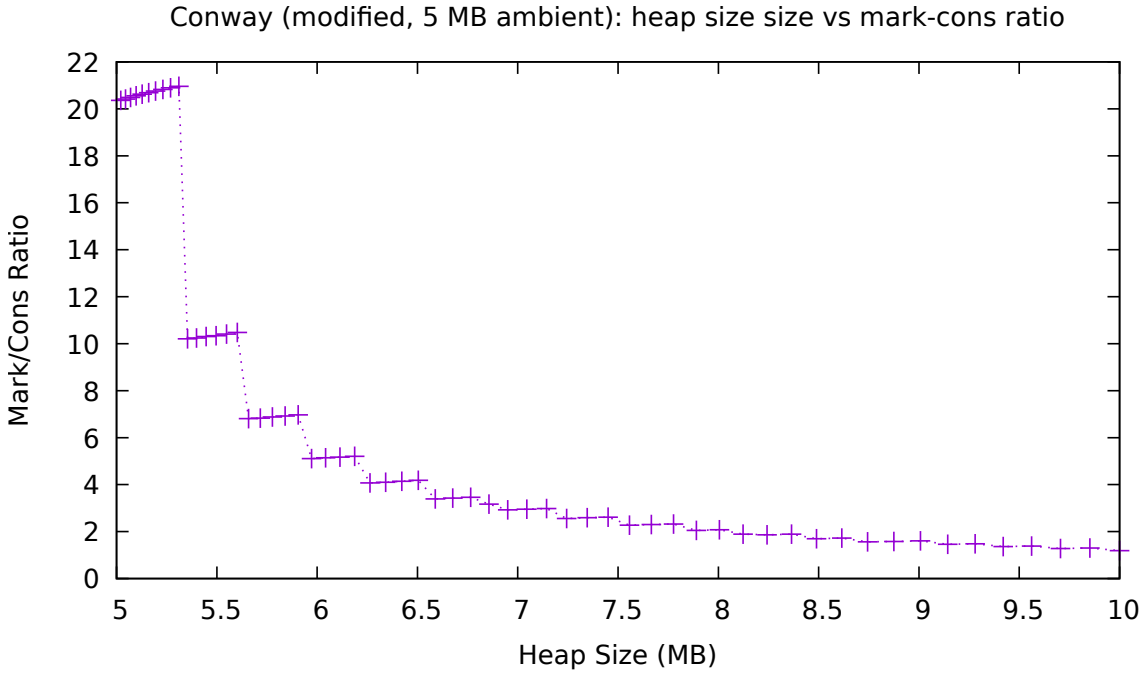
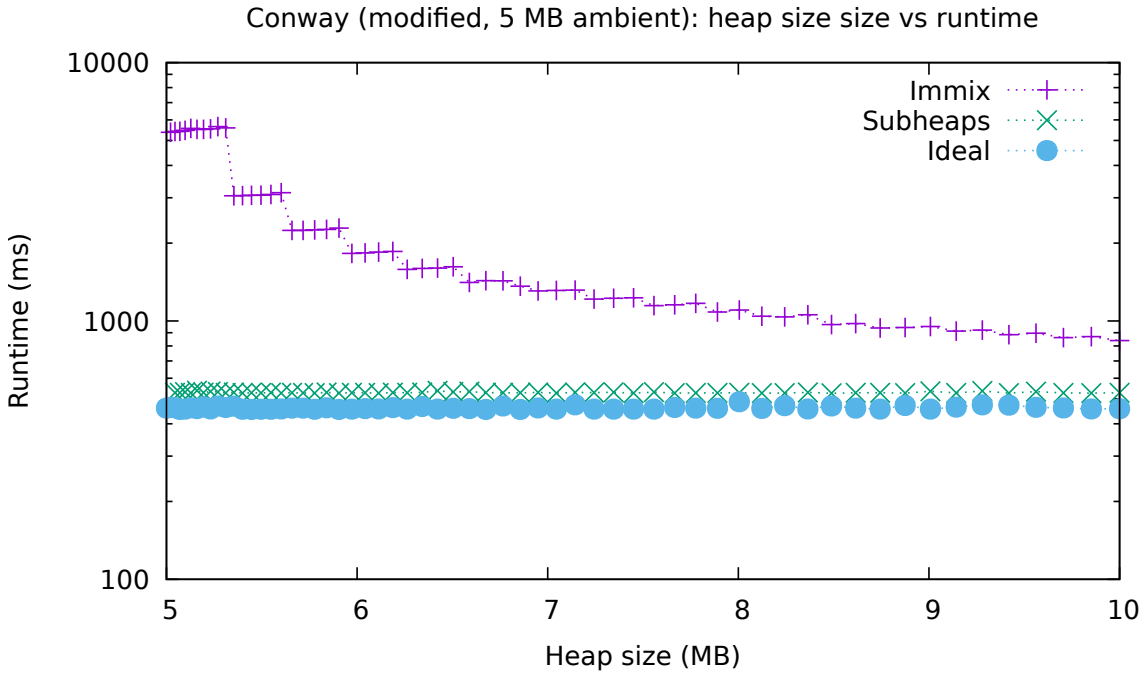


Figure 13: Impact of ballast on Conway.

Increasing ambient live data quickly produces exponentially increasing GC time burdens for both the plain and generational Immix collectors. Subheaps maintain a flat profile. The leftmost data points, with no ambient live data, correspond to the "original" configuration.

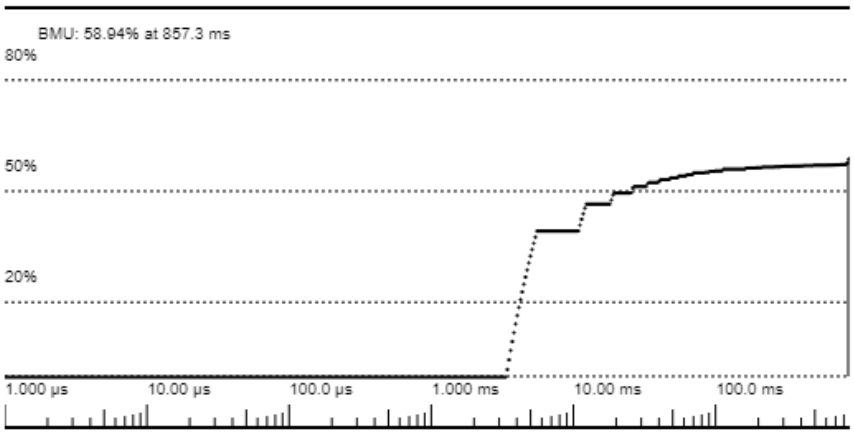


(a) Mark/Cons ratio drastically rises due to long-lived data (compare to Figure 8b).

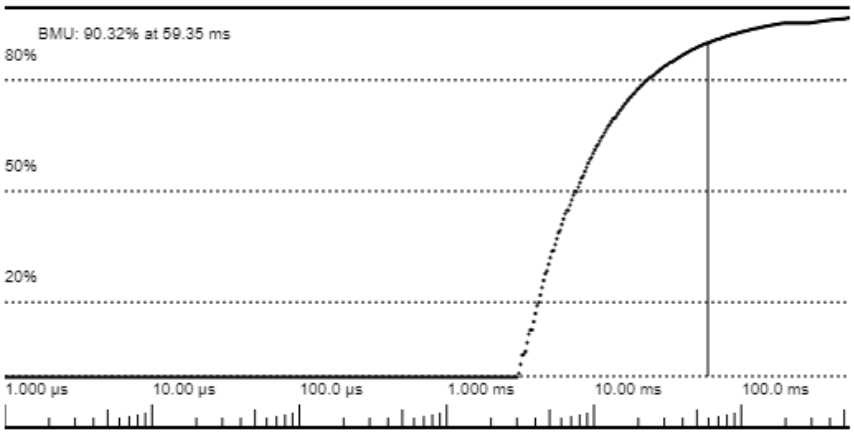


(b) Subheaps eliminate repeated tracing, producing large speed (and latency) gains. Importantly, the runtime with subheaps (and generational collection) is not merely *faster*, it is also *consistent*—the benchmark's runtime is no longer sensitive to choice of heap size.

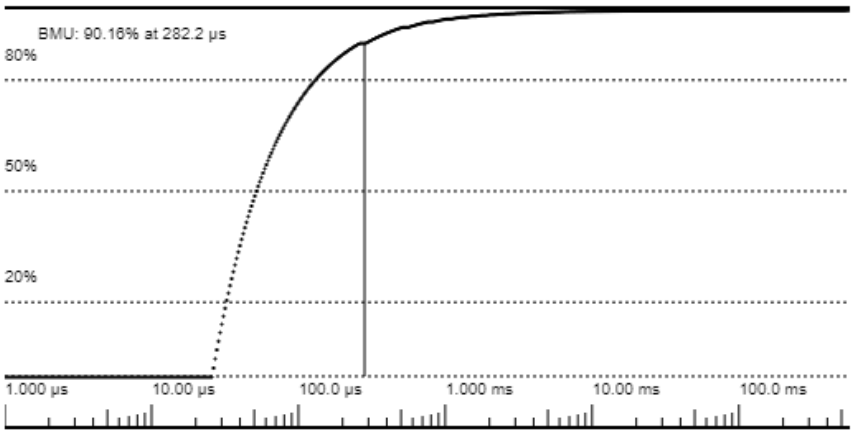
Figure 14: Adding 5 MB of long-lived "background" data significantly changes the benchmark's results in comparison to Figure 8.



(a) Bounded Mutator Utilization curve: non-generational Immix.



(b) Bounded Mutator Utilization curve: Sticky Immix.



(c) Bounded Mutator Utilization curve: subheaps.

Figure 15: Bounded Mutator Utilization curves for the Conway benchmark.

As Figure 14b shows, due to the increased cost of each GC, total runtime depends even more strongly on selection of heap size. Generational collection improves throughput with moderate amounts of ballast, but does not improve worst-case latency, since full-heap collections are still orders of magnitude more expensive than before. With subheaps, the long-lived data can be fully segregated and is never traced by the collector. As a result, the max observed GC pause is 11 microseconds.[3] Unlike with either the baseline or generational Immix implementations, the performance of subheaps does not vary with the amount of long-lived data.

Conclusion This benchmark reflects the potential for subheaps to corral a set of allocations with lifetimes scoped to a loop body. Its simplicity enables exploration of the impact of subheaps and subheap barrier optimizations in detail. Subheaps deterministically reduce the amount of marking required, producing large improvements to GC latency, and moderate throughput improvements compared to a generational collector. In the presence of long-lived allocations, subheaps enable both faster and heap-size-independent runtime. Subheaps eliminate more work than generational collection.

4.3. Tree Microbenchmarks

To explore the behavior of collection in a simplified setting, garbage collection researchers have long used microbenchmarks based on tree structures. This section examines how subheaps can be used to improve the performance of two such venerable microbenchmarks: `binarytrees` and `reynolds2`.

[3] Figure 15c shows a worst-case pause of 26 microseconds due to the overhead of printing statistics for computing mutator utilization.

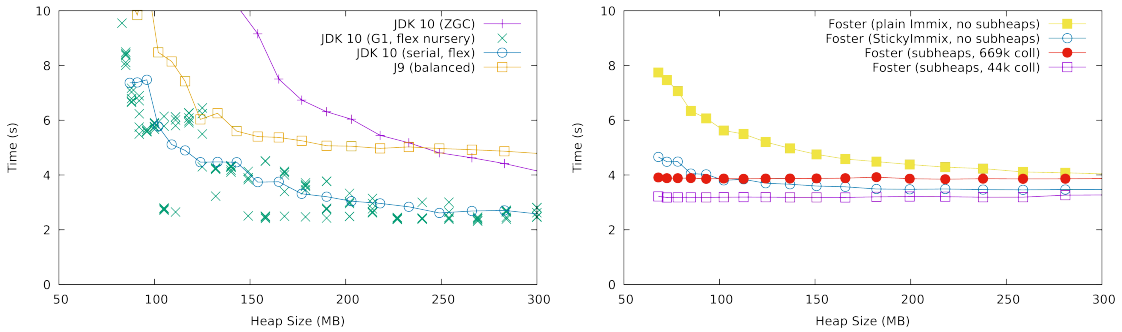


Figure 16: Binary-trees results for Java and Foster

4.3.1. `Binarytrees`

To explore the costs and benefits of aggressive reclamation, we ported the `binarytrees` microbenchmark [Gou18] discussed in Section 3.1 and modified it to make use of subheaps. The basic structure of the program is: a single large binary tree is allocated, which remains live for the duration of the program. In turn, increasing numbers of smaller trees are generated and traversed, after which they become garbage. Non-generational tracing collectors spend significant effort tracing the long-lived tree in the course of reclaiming the garbage generated by the smaller trees. But generational collectors still face a subtle problem: they are likely to evacuate the nursery in the middle of allocating a tree, thus prematurely promoting the already-allocated portion. On a longer-running program, such as a web server, such leakage would eventually trigger full-heap collection if not dealt with by concurrent collection of the mature space. Intermittent full-heap collections may have little effect on throughput, but they wreak havoc with tail latencies in distributed systems [MAHK16].

A key benefit of `binarytrees` is that it has been implemented by many people in a wide variety of languages. This makes it feasible to compare Foster's experimental collector against production-quality collectors on an identical workload. For example, Ferreiro et al [FCJH16] carefully investigate the impact of varying the size of a

generational nursery on collector performance for binarytrees in the context of the GHC implementation of Haskell.

Figure 16 plots the impact of varying heap size for several collector implementations in Java and Foster. Generational collection with StickyImmix closely matches the performance of Java's (generational) serial collector. The non-subheap collectors exhibit space-time tradeoff curves, and use of subheaps produces a flat performance profile, removing the program's sensitivity to heap size.

Two configurations for subheaps were measured, varying in how frequently short-lived trees were collected. Reclaiming every tree individually triggers 669,041 collections. Most of the collections happen with a subheap containing only a few frames, resulting in an average cost per collection of less than one μs. While the per-collection cost is low, the sheer number of collections results in lower performance than generational collection in generously-sized heaps. Fortunately, the programmatic nature of the subheap API—in contrast to schemes linked to inflexible entities like program scope—allows the programmer to trigger subheap collections on *some* rather than *all* loop iterations, thereby reducing GC costs below the cost of generational collection at all heap sizes.

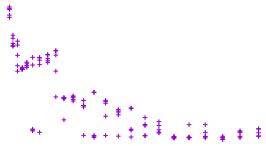


Figure 17: binarytrees on G1.
x-axis: heap size
y-axis: time

This benchmark exhibits a second interesting phenomenon: unpredictable collector costs with Java's G1 concurrent collector. Most collectors exhibited low variance and we display their results as an average of five runs, connected with lines. The throughput results from Java's G1 collector, displayed as raw data points in Figures 16 and 17, show high variance. At several heap sizes, the G1 collector is on the brink of being unable to satisfy mutator demand. In some executions, the program runs to completion with only young pauses. In other executions,

the collector falls behind and starts triggering full-heap collections. This demonstrates the potential for concurrent collectors to hit "performance cliff" behavior.

4.3.2. Reynolds2

Figure 18 lists the source code for the Reynolds2 microbenchmark, ported from the code in Tofte & Talpin's paper on region-based memory management [TT97]. As with binarytrees, the code constructs a complete binary tree, then walks over it. The twist here is that the tree walk itself allocates closures for the predicate provided to the `search` function, on lines 17 and 19.[4] In effect, as the tree is explored, a list of values to search for is accumulated in the form of a chain of predicate functions.

The code as written is compatible with region-based memory management, meaning that region inference does not result in exponentially-growing regions. (Tofte & Talpin note that a seemingly innocuous change—representing the list of values to search for as an explicit list, instead of via functions—destroys this behavior). Of course, space efficiency is not the same as time efficiency. Region-based memory management does not amortize the costs of reclamation for this workload. With MLKit 4.3.18, regions are 4.6x slower than non-generational GC. Subheaps offer more flexibility than region-based memory management in dealing with such granularity issues. The remainder of this subsection investigates how that flexibility affects performance.

For the size-24 input in a 1 MB heap, the baseline code allocates 67.1 M closure objects (32 bytes each; 2.1 GB total) and triggers 2335 collections, which takes 157 ms out of 1.9 s total runtime. The mark/cons ratio with plain Immix collection[5] is 2.417e-3, and in total 273k potential root values are examined.

[4]The functions passed to `oror` are reliably eliminated by inlining.

[5] Use of sticky mark bits degrades performance versus plain Immix on this benchmark, because every value that survives a nursery collection immediately becomes floating garbage, thus increasing the number of collections occurring.

```
1  type case Tree
2    of $Lf
3    of $Br Int32 Tree Tree;
4
5  mktree = { n =>
6    if n ==Int32 0 then Lf else
7      t = mktree (n -Int32 1);
8      Br n t t
9    end
10 };
11
12 search = { p => t =>
13   case t
14     of $Lf -> False
15     of $Br x t1 t2 ->
16       { p x } `oror` {
17         { search { y => { y ==Int32 x } `oror` { p y } } t1 }
18         `oror`
19         { search { y => { y ==Int32 x } `oror` { p y } } t2 }
20       }
21   end
22 };
23
24 reynolds2 = { search { x => False } (mktree 24) };
```

Figure 18: Reynolds2 source code in Foster.
Note that, like Haskell, Foster supports the use of regular identifiers as infix binary operators using the `` `ident` `` lexical syntax. The `oror` function is a functional implementation of the lazily evaluted `||` operator in C, or Standard ML's `orelse` keyword. An expression wrapped in curly brackets, with no arrow parameters, denotes a zero-argument function (a thunk).

One approach to using subheaps would be to segregate the tree being traversed from the closures allocated while traversing it. The intuition here is that the benchmark allocates one (short-lived) closure per (long-lived) tree node traversed, so perhaps the program's allocations can be split in order to (implicitly) focus collection effort only on closures and not on nodes. However, the original code takes a clever shortcut: on line 8, the left and right subtrees, which represent identical values, are in fact pointers to the same value. Thus what is conceptually a complete binary tree is implemented with a redundantly linked list, so the tree occupies less than one kilobyte out of the multiple gigabytes allocated by the benchmark. As a result, segregating the list-represented tree produces no savings at all.

We can, however, explicitly focus collection effort on the closures. Each closure's lifetime is scoped precisely to the activation of `search` that uses it. In the extreme, each closure could be allocated in a fresh subheap and collected precisely when it dies. As with regions, this scheme would suffer high overheads from rapid creation and disposal of subheaps. The minimum space expenditure for a subheap (one Immix line for the subheap contents, plus the backing subheap object) is an order of magnitude larger than a single closure; thus, storing each closure in a separate subheap would increase GC pressure by a corresponding amount.

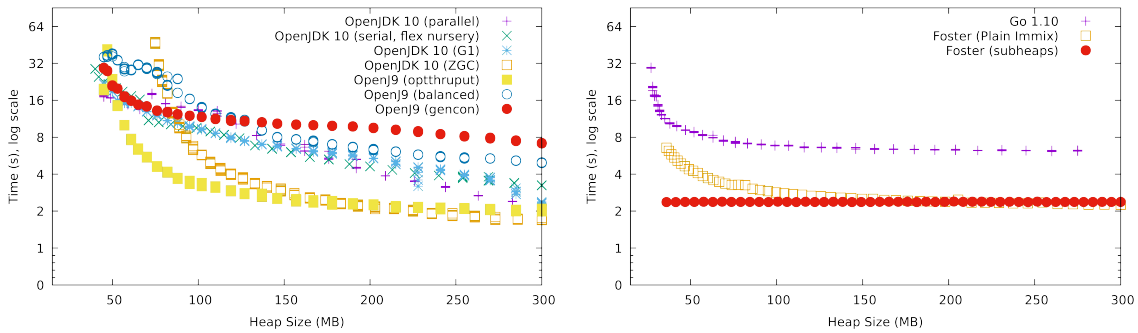
A better idea is to capture groups of closures. Since complete trees have most of their nodes near the leaves, a simple approach is to capture subtrees near the leaves (or more precisely, the closures allocated while searching through said subtrees) and ignore the intermediate nodes nearer the root of the tree. We can modify the recursive calls to `search` to occur in fresh subheaps only at a particular level using a helper akin to `inTempSubheap` from Section 2.6.2. Doing so produces interesting results with a Goldilocks-style phenomenon, captured in Table 1.

Table 1: Effect of varying granularity of subheap collection on Reynolds2 (input 24).

Subheap @ Depth	GC time (ms)	Mark/Cons Ratio	Explicit Collections	Implicit Collections
None	157	2.42e-3	0	2335
16	165	2.13e-3	0.5 k	2048
15	166	2.15e-3	1 k	2048
14	168	2.17e-3	2 k	2048
13	29	0	4 k	0
12	46	0	8.1 k	0
11	115	1.06e-6	16.3 k	1
10	197	2.13e-6	32.7 k	2
9	279	4.35e-6	65 k	4
8	420	9.40e-6	131 k	9

With the "right" choice of granularity for subheaps, all marking work is eliminated and no involuntary collections occur. When capturing large groups, the heap itself is too small to accommodate the data being routed into the active subheap, resulting in a net increase in collections (and GC time). When capturing smaller groups, the savings from reduced tracing (as reflected by a drop in mark/cons ratio) are outweighed by the cost of stack scanning, leading to a net increase in GC time. For example, compare the base configuration to subheaps at depth 8: the mark/cons ratio falls by two orders of magnitude, yet the GC time jumps almost threefold. The culprit is the cost of stack scanning, the per-collection cost of which did not change, even as the number of collections grew exponentially. In addition, as less total data is contained within subheaps, enough garbage accumulates to trigger (infrequent) collections, leading to non-zero mark/cons ratios.

After accounting for stack scanning, recording of runtime statistics, and the like, each non-marking collection takes roughly 5.5 μs. This benchmark clearly shows that even such miniscule costs add up when executed frequently enough.



(a) Java collectors

(b) Non-Java collectors

Figure 19: Minicache (B=1000, N=700, K=300) results

4.4. Software Caches

As discussed in Section 3.5, programs with large data caches pose a challenge for tracing garbage collectors. I wrote a small cross-language microbenchmark, called minicache, to quantify the cost of tracing GC for such caches. The structure of the minicache heap is illustrated in Figure 6 (minus the pointer between entries). There is an array containing B binary trees comprised of N nodes each. The minicache workload is to make K passes over the array, allocating and inserting one replacement subtree at a time. The results of running minicache ($B = 1000, N = 700, K = 300$) are presented in Figure 19. The left plot shows the behavior of a variety of collectors (parallel, serial, and concurrent) for the benchmark running on several widely used Java virtual machines. The right plot shows, with the same axes, the behavior for the same benchmark written in Go (using a concurrent collector) and Foster (with and without subheap augmentation). Each program was run five times per heap size. The plots show raw datapoints, not averages. Run-to-run timing variance was low.

Figure 20 explores different Immix variants on the Minicache workload. Generational collection with sticky mark bits performs worse than plain Immix—unsurprising, because the lifetime of cached data does not follow the generational hypothesis. Mean-

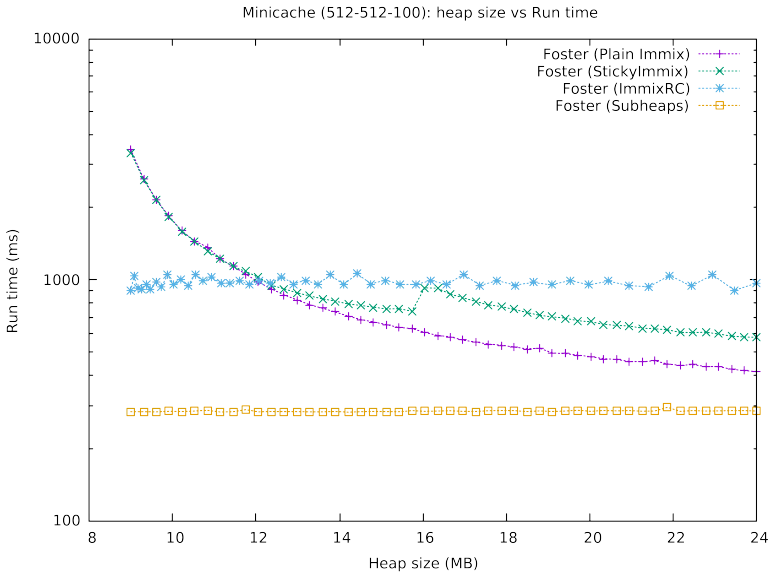


Figure 20: Comparison of Foster's Immix variants on minicache.

while, ImmixRC's behavior is independent of heap size. It is faster than non-RC Immix in small heaps and slower in larger heaps. While both ImmixRC and subheaps have flat profiles, ImmixRC is slower by a large constant factor. The difference is that subheaps reduce the benchmark's workload by making sure objects need not be traced, whereas ImmixRC merely enforces a heap-size independent workload: every allocation is effectively traced twice with little amortization (once each for recursive marking and unmarking).

What makes these graphs interesting is the *shape* of the results: the tracing collectors exhibit classic space-time tradeoff curves, which reference counting—even when "emulated" with subheaps—avoids. By putting each subtree in a separate subheap we combine the low cost of region-based reclamation with a reliably flat performance profile.

The authors of M^3 [TAV14] were partially motivated by the poor performance of a Memcached-like system written in Go, which at the time offered only a serial collector.

Recent versions of Go offer a state of the art concurrent collector. Would it have have avoided their woes? The results in Figure 19b suggest not. Especially in tight heaps, concurrent collection cannot overcome the sheer amount of work generated by the minicache workload.

4.4.1. memcached & ghost thereof

Minicache is designed to throw the memory management issues of a cache into stark relief. These effects are muted in a real cache for several reasons. First, minicache simulates a cache's workload with no superfluous influences: the workload involves no hashing, nondeterminism, or I/O. A real cache must do this extra work, which obscures the costs of GC. Second, minicache stores large, pointer-dense object graphs, which amplify GC work. Many caches store data like strings or binary blobs which do not need tracing. Thus, while caches are not GC-friendly, most caches will not observe the severe (exponential decay) throughput impacts illustrated.

However, throughput is not the only relevant performance metric for a cache. Latency is often a more critical concern for network-enabled cache servers. The minicache benchmark cannot realistically measure end-to-end latency. To demonstrate the effect of subheaps on a more realistic server, I implemented `mcd`: a minimal network-enabled clone of Memcached in Foster. Using a lexer compiled from C into Foster, plus bindings to the POSIX sockets API, `mcd` parses and implements the `GET` and `SET` commands in the memcached wire protocol.

The `mutilate` program [LK14, Lev14] was used to generate memcached wire traffic with a mix of 90% reads and 10% updates. Three configurations for `mcd` were tested. Since our workload induces 166 MB of allocation, a 170 MB heap is large enough to avoid GC entirely. Reducing the heap size to 130 MB results in one garbage collection

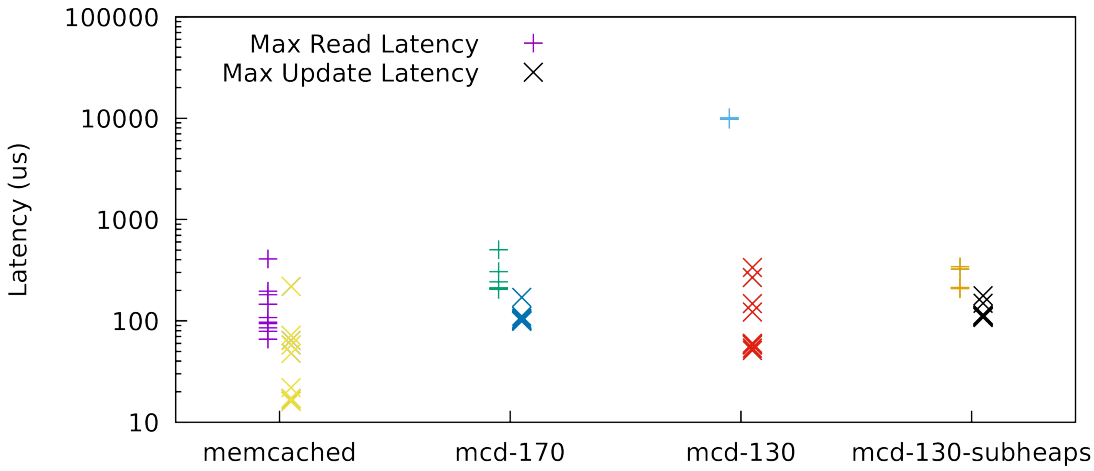


Figure 21: End-to-end memcached workload latency

cycle, which subheaps avoid. The results of testing these variants of `mcd`, along with `memcached` itself, are shown in Figure 21. Reading left to right: When the heap is large enough to avoid garbage collection, `mcd` shows max latencies comparable to `memcached`.[6] In a smaller heap, the cost of GC is reflected in severe degradation of max read request latency. The application of subheaps, in the smaller heap, successfully replaces one costly GC with almost thirty thousand cheap GCs, each of which costs barely more than a microsecond. This effectively eliminates the latency impact of garbage collection for the `mcd` server. The GC-induced throughput degradation for `mcd-130` is 1.8%, increased to 4.2% with subheaps. Most of the lost throughput for subheaps is due to repeated stack scans.

Experience Although the `mcd` server loop is relatively simple, it still highlights four interesting phenomena surfaced by applying subheaps in practice. Some of these findings have also been explored in Section 3.4.

First, when the goal is to not merely reduce but entirely eliminate full GCs, we

[6] `memcached`'s throughput is roughly four times that of `mcd`; like most functional languages, idiomatic Foster makes heavier use of allocation, indirection, and bounds checking than does C.

must capture all allocated data within the server loop, not merely the subset of data allocated within each cache bucket. Otherwise, the un-captured data will accumulate and eventually trigger a collection. When data of varying lifetimes is interleaved, proper separation can increase the subheap annotation burden.

Second: circumstances sometimes force allocation to occur before the "proper" destination subheap is known. The Memcached protocol has clients send servers lines with a command name, followed by a key, followed by command-specific fields. There is a bit of a catch-22 with the key's memory management: it must be allocated in a bucket's subheap to detect hash collisions, but the choice of what bucket—and therefore what subheap—to use can only be made after it has been extracted from the network, and thus allocated in some other subheap. To resolve the mismatch in object lifetimes, the programmer must store a fresh *copy* of the key in the chosen bucket's subheap. Failure to do so creates a long-lived subheap-crossing pointer, destroying the potential for subheaps to improve performance.

Third, there can be tension between separation of concerns in code versus data. Cache buckets can be empty, and each non-empty cache bucket needs an associated subheap. One scheme for this is to create subheaps on demand, as each bucket transitions from empty to non-empty. This allows the use of `subheapOf` (see Section 2.7.2) without needing any changes in data representation, but it mixes unrelated concerns in the server response loop. Alternatively, creating a subheap in advance for each bucket leads to better separation of concerns in code, but it no longer suffices to use a null pointer to represent an empty cache bucket. Some change in data representation is needed for the web server to map empty buckets to their respective subheaps.

Finally, care must be taken not to capture too much data in a subheap. Interleaved with the allocations that must go in each bucket, the server also generates some short-

lived garbage. Sticking this garbage in the long-lived cache entries inflates the amount of space needed to store cache entries in subheaps. Whether this is acceptable or not depends on the amount of provisioned heap space.

4.5. Self-Adjusting Computation

Self-adjusting computations (also known as incremental computations) are those which can efficiently re-compute results as inputs change. This genre of programs is known to be problematic for tracing GCs [HA08]. Self-adjusting computations violate many of the heuristics employed by garbage collectors, especially generational collectors. In particular, self-adjusting computations tend to have a large amount of long-lived data, with frequent mutatations to point to young data. Furthermore, object lifetimes do not satisfy the weak generational hypothesis because memoized computations are kept alive until made irrelevant by changed inputs. To avoid the inflated costs of garbage collection for self-adjusting computations, researchers have explored options ranging from custom compilers [LW10] to custom language extensions [HAC09] to rewriting libraries in different languages [HKHF14, HDH$^+$18].

On paper, subheaps appear to be a promising technique for managing the memory used by self-adjusting computations. First, the closures representing invocations of self-adjusting computations own the data allocated in their dynamic extent, minus the dynamic extent of their callees. This behavior—of capturing allocations within a dynamic extent—precisely matches the activation-based usage model for subheaps. Second, the deactivation of a closure is an explicit operation, suggesting a natural place to hook in a call to `subheapCollect`. Third, the fine granularity of self-adjusting closures—which often allocate only a few dozen bytes—suggests that subheaps can be applied with lower space overheads than systems with multi-kilobyte minimum

region sizes. The SAC library [Aca05, ABBT06] for self-adjusting computation in Standard ML provides a concrete example. The `qsort` test case allocates just shy of 50,000 modifiable references. Because a single subheap is lightweight—less than 600 bytes—the net cost with subheaps would be only 28 MB. If each modifiable reference were given a subheap with a full 32 KB frame, the net memory usage for the program would explode to 1.6 GB.

As with previously covered benchmarks, there are multiple means by which subheaps might be applied to SAC-using code. As it turns out, in the test suite and benchmarks exercising the SAC library, the incremental subsystem is often not the source of most allocations. For example, in the `qsort` benchmark, 78% of allocated bytes come from the verification step of the SAC test harness, which serializes incremental state and compares against a non-incremental implementation. Tackling these temporary allocations is an easy first step, but is a relatively trivial target for subheaps. Much more interesting is to manage the incremental computations themselves.

To apply subheaps to SAC requires some background on how the library works. The SAC library tracks a dependence graph of memoized computations and their input sources; when an input is updated, memoized computations are recursively re-run until reaching quiescence. Specialized memory management for SAC [HA08] hooks into this change-propagation process. In short, the library's correctness criterion is that incrementally computed results do not differ from their non-incremental equivalents. This consistency theorem implies that subcomputations—and their associated allocations—which have been invalidated are guaranteed to be garbage when change propagation completes. Even with a guiding principle for how to associate subheaps with domain elements, the details of how to multiplex the dozen-plus allocations arising from each call to `Modref.read` (see Figure 26) across subheaps remains somewhat

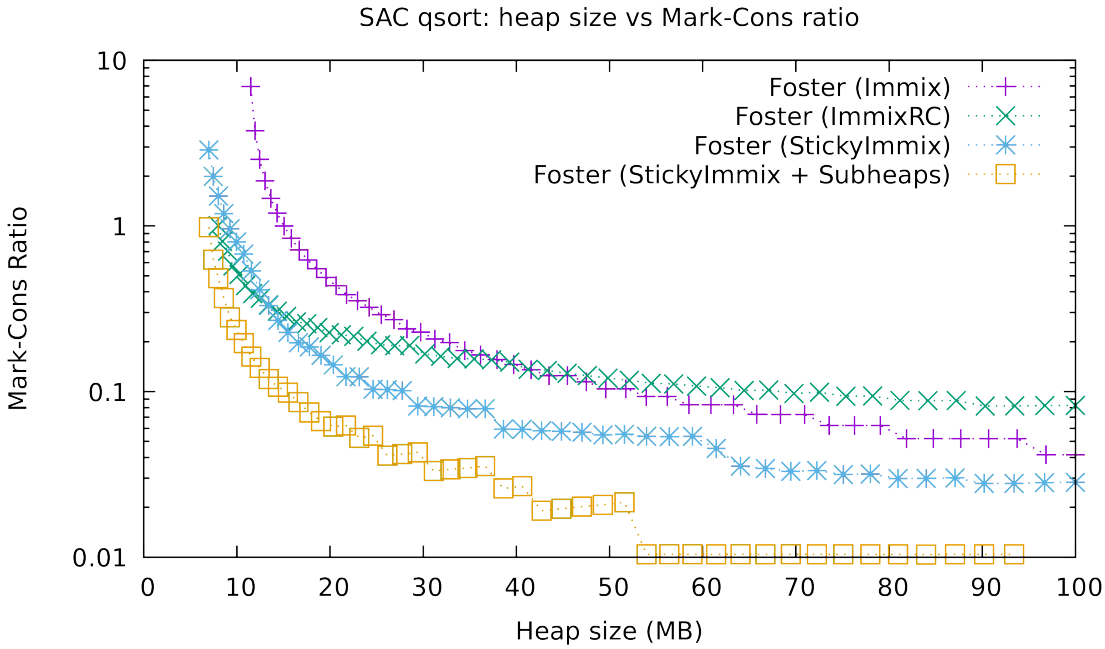


Figure 22: Mark/Cons Ratios for SAC qsort-500 example
(subheaps for temporary data only)

subtle.

To evaluate the impact of subheaps on self-adjusting programs written against the SAC library, I modified the MLton compiler [Wee06] to emit its SSA-based intermediate representation (augmented with new source-level subheap primitives) as Foster code. This enabled automated translation of whole SML programs, extended to use subheaps, into Foster.[7] Because the application of subheaps targets the SAC library rather than the programs written using the SAC library, we focus our examination on the `qsort` benchmark highlighted by Hammer et al [HA08].

Figure 22 shows how the amount of tracing work performed is affected by Foster's various Immix collectors. The baseline Immix collector is the slowest in moderate

[7] Applying subheaps to MLton itself is tempting, but MLton is a 32 MB binary, and its bootstrap produces more than 700 MB of Foster source—too large for Foster's prototype compiler to process in reasonable time.

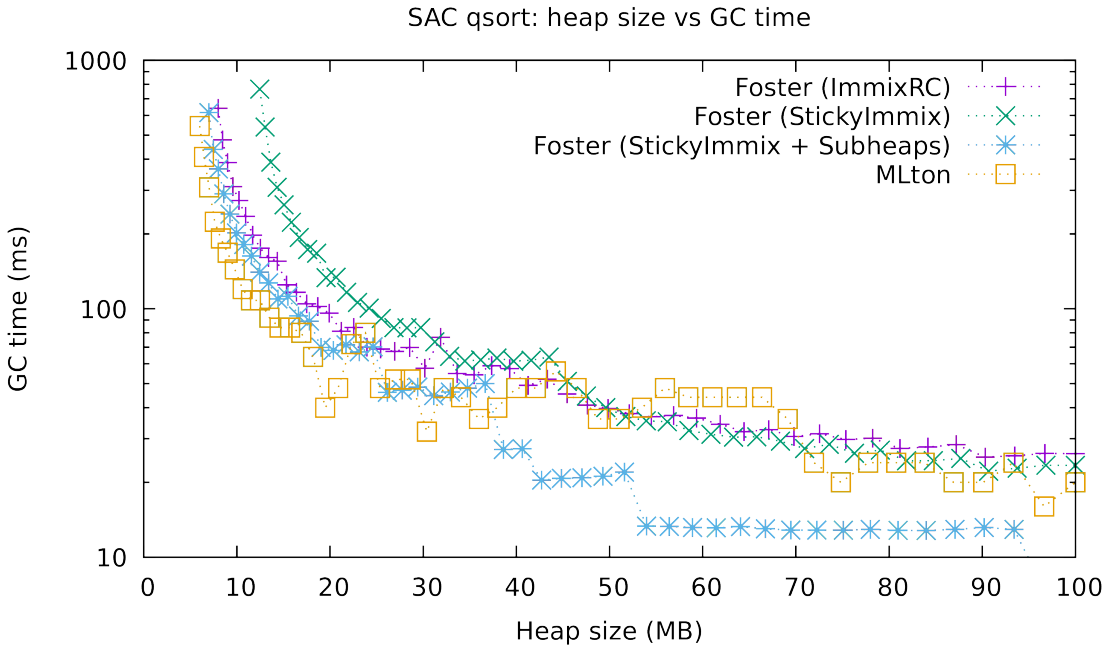


Figure 23: GC runtime for SAC qsort-500 example
(subheaps for temporary data only)

(< 5× minimal) sized heaps, but for oversized heaps it becomes slightly more efficient than ImmixRC, which spends effort incrementing and decrementing individual objects. Sticky Immix becomes more efficient than ImmixRC at a 1.7× heap. Using subheaps to corral temporary data produces uniform savings in tracing work. For example, with a 54 MB heap, Sticky Immix triggers 11 collections; using subheaps reduces that to a single GC.

However, tracing work alone does not tell the full story. Figure 23 shows that while GC time follows similar curves as does tracing work, it only varies by two orders of magnitude rather than four. The figure also shows that the GC time exhibited by Foster's GC backends is comparable to the MLton runtime's GC. Finally, subheaps also add costs to the mutator for executing write barrier checks and maintaining remembered sets (in general; this configuration does not add any cross-subheap re-

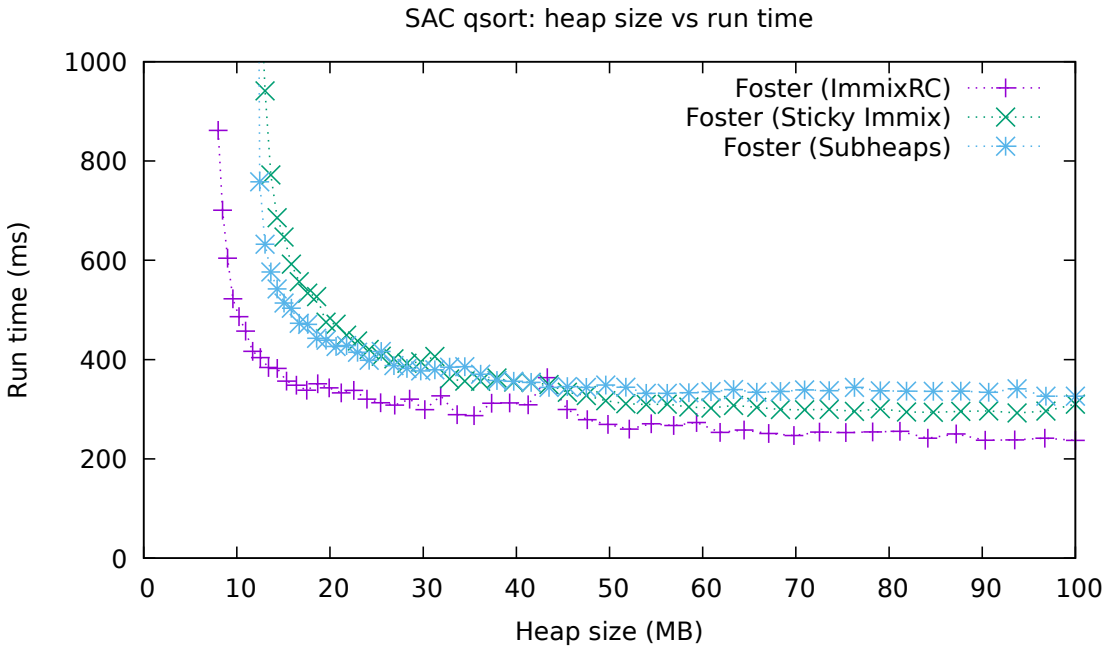


Figure 24: Program runtime for SAC qsort-500 example
(subheaps for temporary data only)

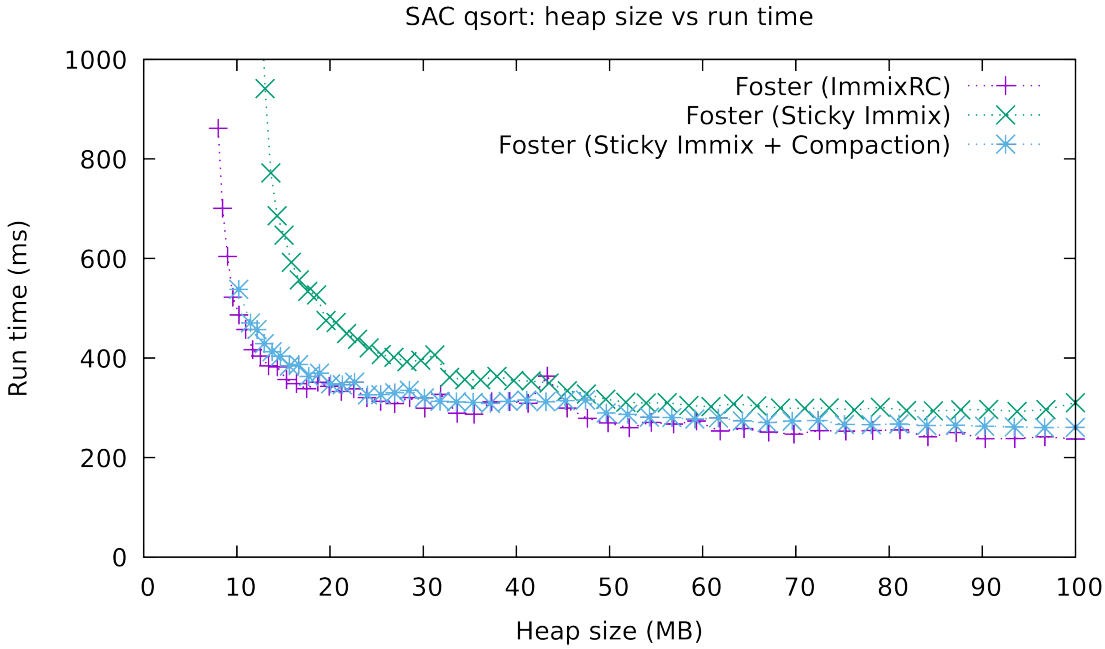


Figure 25: Impact of augmenting the Sticky Immix collector with compaction.
Compaction provides greater benefit than subheaps, particularly in tight heaps.

membered set entries). Figure 24 shows that in larger heaps, the mutator costs of subheaps outweigh the reduced tracing load. Reductions in program runtime from using subheaps for temporary data appear only in small heap sizes. Meanwhile, Figure 25 shows that the difference in Figure 24 between Sticky Immix and ImmixRC is almost entirely due to the latter's use of compaction, which permits execution in smaller heaps and leads to savings in mutator time. With compaction enabled, Sticky Immix produces very similar results as ImmixRC. Overall, using subheaps to manage the temporary data generated by the testing harness successfully reduces GC work but shows only modest improvements to program runtime.

What about using subheaps to manage the allocations of individual memoized computations? A primary challenge is that, as mentioned previously, self-adjusting computation involves intricate management of higher-order stateful functions. Even when looking at a single page of code, it is difficult to determine the optimal placement for subheap primitives. Consider the definition of `Modref.read` in Figure 26. We can walk through a few factors influencing how this code might take advantage of subheaps:

- On line 34, the definition of `run` silently allocates a closure that captures two variables. Likewise, calls to `delete` on lines 45 and 11 allocate a closure that will only be invoked at a statically-unknown later time.

- The `new` value captures the `run` closure and may capture previous definitions; the `delete` closure gets stored within the `TimeStamps` module as time stamp invalidators.

- A subheap activated after line 1 would not necessarily be activated when line 3 eventually executes. Long-lived closures use the subheap active at their invoca-

```
1  fun read modr f = let
2    fun delete node = fn () =>
3       case node of
4          ONE _ =>
5            let val WRITE (v, rs) = !modr in
6              case rs of
7                ONE _ => modr := WRITE(v,ZERO)
8              | FUN (p, reader as (_,t,_), n as ref next) =>
9                let
10                 val new = ONE reader
11                 val _ = TimeStamps.setInv (t, delete new)
12               in
13                 case next of
14                   ZERO => modr := WRITE (v, new)
15                 | FUN(pofn,_,_) => (pofn := new;
16                                     modr := WRITE (v, next))
17                 | _ => raise InternalError
18               end
19           end
20       | FUN (p as ref prev, _, n as ref next ) =>
21         case (prev,next) of
22           (ONE _, ZERO) =>
23             let val WRITE (v,_) = !modr
24                  in modr := WRITE (v,prev) end
25         | (FUN (_,_, nofp), ZERO) => nofp := ZERO
26         | (ONE _, FUN (pofn, _,_)) =>
27           let val WRITE(v,_) = !modr val _ = pofn := prev in
28               modr := WRITE (v,next) end
29         | (FUN(_, _, nofp), FUN(pofn,_,_)) =>
30               (nofp := next; pofn := prev)
31         | _ => raise InternalError
32
33   val WRITE (v,rs) = !modr
34   fun run () = let val WRITE(v,_) = !modr in f v end
35   val t1 = insertTime ()
36   val _ = f v
37   val t2 = insertTime ()
38   val WRITE(v,rs) = !modr
39   val new = case rs of
40         ZERO => ONE (run, t1, t2)
41       | ONE _ => FUN (ref rs, (run, t1, t2), ref ZERO)
42       | FUN (prev,_,next) =>
43         let val new = FUN (ref (!prev), (run, t1, t2), ref rs) in
44             prev := new; new end
45   val _ = TimeStamps.setInv (t1, delete new)
46 in
47   modr := WRITE (v,new)
48 end
```

Figure 26: Definition of Modref.read

tion sites, not their definition site. This is a downside of using dynamic scope instead of lexical scope.

- The calls to `insertTime` on lines 35 and 37 allocate timestamp nodes with lifetimes not clearly bound to other allocated data in the `read` function.

- Time stamp invalidation is stateful; the SAC library happens to call a timestamp node's invalidator *before* removing the reference to the node via mutation. This implies that timestamp nodes will be considered live for garbage collection purposes when invalidators run. The choice of how to order removal and invalidation cannot be distinguished by a non-subheap-aware client, but a client using subheaps can detect the difference in tracing performance.

- The various dereferences of `modr` can see values allocated in different calls to `read`; this complicates both reasoning about where it might be profitable to trigger subheap collections, and where it is sound to eliminate subheap barriers.

- Allocations occur on lines 7, 10, 11, 14, 16, 24, 28, 33, 34, 36, 39, 40, 42, 44, 46, and possibly 35. Data may become unreachable on lines 7, 14, 15, 16, 24, 25, 27, 28, 30, 43, and 46. Frequently only a portion of a a data structure will be rendered unreachable by any given mutation. In contrast to the Conway benchmark, there are no clear and obvious points in the code where any given subheap can be most profitably collected.

Another distinguishing feature of the SAC library is that it makes intricate use of recursion. It turns out that, even with only a single user-created subheap S, there are three—not two!—choices for where to put allocations in the body of a recursive function. The three possible "destination" subheaps are S, the default subheap, or the active subheap. The same call site can see different bindings for the active

subheap. While these decisions must be made for both recursive and non-recursive code, experience with SAC shows that recursion can easily obscure the correct choice.

Selectively enabling subheaps for portions of `Modref.read` allows us to measure the amount of allocation performed by various portions of the implementation. The simplest approach is to wrap the whole function definition with a subheap activation; this captures 48.29 MB out of the 451.0 MB allocated on the qsort-500 benchmark. Due to the dynamic scoping of subheaps, this does not capture allocations made by `delete` on lines 3-29, nor in the execution of `f` within `run`. Fixing these omissions nearly doubles the amount of subheap-captured data, to 90.15 MB.

However, maximizing the number of subheap-allocated bytes is not our only concern. To minimize remembered set maintenance costs, we should also find subheap boundaries that produce few remembered pointers, as discussed in Section 3.2. Figure 27 illustrates why minimal cuts are difficult to find for the SAC library. The timestamp nodes `t1` and `t2` are elements of a complex linked data structure maintained by the `TimeStamps` module. The `t1` node is additionally circularly linked, as its invalidator—the thunk produced by `delete new`—indirectly refers back to `t1`. The `new` node is also an element of a mutable circularly linked list of readers. These allocations are repeated for every read of each modifiable reference in the program.

Capturing a partial subset of these nodes in subheaps would produce poor performance: the arrows illustrated in Figure 27 are all immutable references. This means that, as discussed in Section 3.5, minimal subheap collection will be unable to identify dead data, and we will be forced to wastefully inspect the sources of subheap-crossing pointers as well. But capturing the entire cycle in a subheap (for each modifiable read) also produces problematic subheap-crossing pointers, this time within the guts of the `TimeStamps` module. Meanwhile, shunting all this data to a single subheap (or

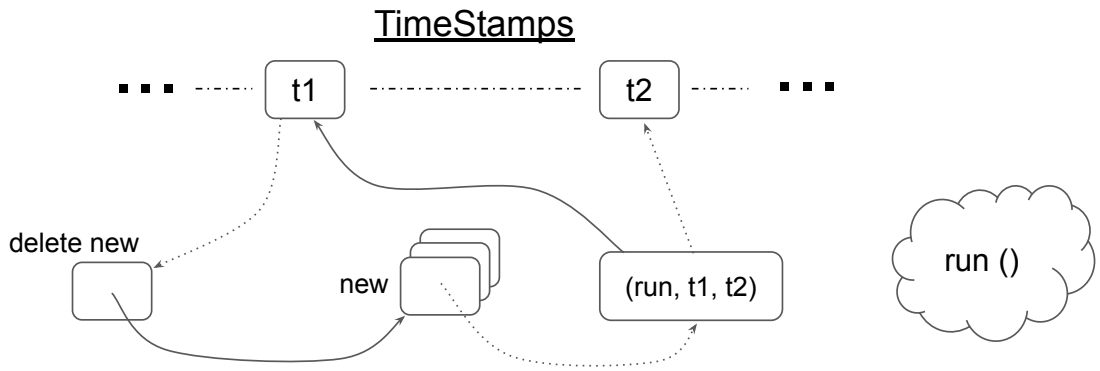


Figure 27: Partial heap structure allocated by Modref.read

even one subheap per modifiable reference) would avoid the burden of remembered pointers, but would also fail to effectively separate dead from live data.

There is one source of seemingly well-isolated data: the payloads generated by executing memoized closures. These payloads originate from lines 32 and 34 in Figure 26, and have no obvious ties to the surrounding code or data structures, which is why they are illustrated as a cloud around `run ()` in Figure 27. On qsort-500, these payloads account for almost exactly half of all data allocated by `Modref.read`. Unfortunately the payload data turns out not to be isolated. Capturing `run`'s data within per-read subheaps ends up doubling mutator time by causing 1.76 M subheap write barriers to trigger. The reason for this is made clearer by inspection of the definition of the incremental quicksort function in Figure 28: due to higher-order usage of the SAC library API, the memoized closures (on lines 9, 11, and 18) reference data from external subheaps and allocate new data that will be manipulated in other subheaps.

Figure 29 illustrates the disappointing results from trying to use subheaps in a fine-grained way to manage memory for the SAC qsort/500 benchmark. The leftmost bar illustrates a rough limit cost for the program without influence from garbage collection, obtained by measuring the program runtime in a sufficiently large heap

```
1  fun sort l =
2    let
3      val lift = C.mkLift2 (ML.eq, ML.eq)
4
5      fun qsortM (l,cr, prev,next) =
6        l ==> (fn c =>
7        case c of
8          ML.NIL => ML.write cr
9        | ML.CONS(h,t) => t ==>  (fn ct =>
10          lift ([Box.indexOf h, Box.indexOf prev,
11                                Box.indexOf next],ct,cr)
12                (fn (t,rest)  =>
13            let
14              val (_,hv) = h
15              fun fg (_,kv) = (Poly.evalOsgn(Key.compare (kv,hv)))
16              val (les,grt) = ML.split fg t
17
18              val bh = Box.fromOption (SOME h)
19              val mid = C.modref (rest ==> (fn cr =>
20                                      qsortM (grt,cr,bh, next)))
21            in
22              qsortM (les,ML.CONS(h,mid),prev, bh)
23            end)))
24   in
25     C.modref (qsortM (l, ML.NIL, Box.fromOption NONE,
26                                  Box.fromOption NONE))
27   end
```

Figure 28: Definition of SAC's qsort implementation

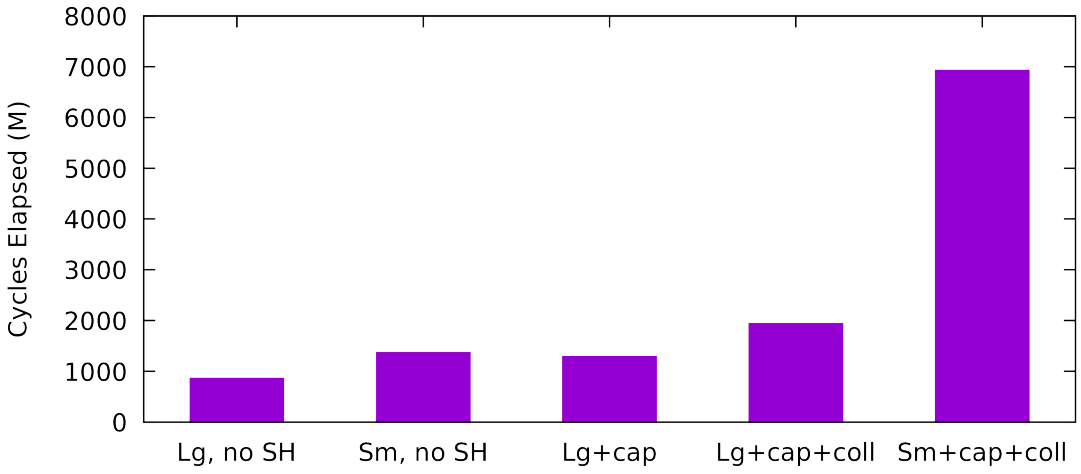


Figure 29: Cycle timings for fine-grained subheaps on SAC qsort/500

that GC does not occur.

The next bar shrinks the heap such that collection does occur; it suggests that GC accounts for more than a third of program runtime. Thus there is potential for subheaps to improve performance by reducing GC costs. The middle bar, labeled "Lg+cap", isolates the cost of mutator overhead from write barriers and remembered set maintenance when using fine-grained subheaps. In this configuration, the heap is again large enough that collections do not occur. Because of the intricate internal structure of the SAC data structures discussed previously, the mutator overhead from subheaps is nearly as expensive as the cost of collection without subheaps, suggesting that subheap collection must be very nearly free to produce a net win.

Unfortunately, the next bar shows that subheap collection is not free; in fact, it is very slightly more expensive than non-subheap collection. The final bar shows that, in a small heap, application of fine-grained subheaps produces a large degradation in runtime compared to not using subheaps at all.

These results show that, counter to my expectations when first considering the marriage of self-adjusting computation and subheaps, the potential for subheaps to manage memory in a fine-grained way is stymied by the intricate, higher-order nature of the SAC library's implementation.

CHAPTER 5 : Challenges & Future Work

This dissertation explores a bare-bones proof of concept incarnation of subheaps. There are several aspects of subheaps as they would likely appear in practice—ranging from curious to critical—that have been heretofore ignored for clarity of focus. This chapter explores a few of these "missing pieces."

5.1. Concurrency

Shared-memory concurrency impedes high-frequency reclamation of subheaps. The key issue is: to collect a given subheap, we must know whether or not there are pointers into the subheap from other threads' stack frames. When answering this question involves coordination between threads, the cost of such coordination limits the maximum frequency of collection.

A key premise of subheaps is that the cost of reclaiming dead space should be proportional to the amount of live space traversed. This premise is violated in a setting with concurrent threads that share memory, because the cost of scanning all stack frames is proportional to the number of outstanding threads.

The constant factors at hand also matter: in traditional concurrent collectors, stack scanning is often done in a brief cooperative stop-the-world pause lasting roughly one millisecond. In contrast, reclamation of a mostly empty subheap operates at megabytes per microsecond—and processing completely empty subheaps is even faster. If every reclamation of every subheap is forced to do a full scan of all stacks, the gains from moderately sized[1] subheaps would be mostly lost. Yak [NFX+16] forces

[1] That is, subheaps containing moderate amounts of allocated data. For an allocation rate of N MB/ms, reclamation of kN MB of allocated space need happen no more often than once per k milliseconds.

thread synchronization in this manner. The impact of this choice is acceptable in part because their domain (Big Data) implies large regions with relatively infrequent collections.

We can avoid stack scans if the runtime can statically or dynamically guarantee isolation between threads. Some collectors enforce such isolation dynamically by copying the transitive closure instead of recording a thread-crossing remembered set entry. However, such copying can be expensive in space, time, and latency, and often ends up being wasted work [Mol15]. When such copies are done silently by the runtime, it becomes more difficult for the programmer to reason about when and why certain collection points might be inefficient. As discussed in Section 2.4.3, it also curtails the effectiveness of static optimizations for write barriers.

Another possibility for combining subheaps with concurrency would be to revisit the basic semantics of the subheap API. The API in Section 2.3 is fundamentally synchronous in its treatment of collection requests. An alternative design could make condemnation record logical snapshots, which would be processed asynchronously. In this model, the collection API furnishes requests rather than commands. This change would complicate the implementation of subheaps, but would provide the runtime more flexibility in scheduling collections, potentially reducing wasted work.

One potential downside of subheaps in a concurrent setting—especially with a synchronous API—is of running into bottlenecks that do not affect single-threaded programs. The amortization of subheaps fundamentally depends on allowing garbage to accumulate, so as to reduce the average cost per reclaimed byte. Inoue et al [IKN09] illustrated this risk in the context of a web server. Their study revealed two interesting phenomeona. First, they showed that using region-based memory management reduced the cost of collection yet produced an overall slowdown due to degraded

mutator behavior. Second, they showed that the mutator regression was due to old dead objects being flushed from processor caches, instead of being reused as with traditional malloc/free systems. This produced contention on the memory subsystem that was not exhibited by the same system running in single-threaded mode.

The simplest route to achieving static isolation is to entirely forsake shared-memory multithreading. Such restrictions are often indepedently appealing for avoiding the conceptual complexity of concurrent mutation, weak memory models, etc. Subheaps may end up being most easily justified in the context of more restrictive programming models (such as message passing actors) that forgo shared-memory concurrency.

5.2. Untrusted Code

The bulk of our discussion around subheaps has assumed a non-adversarial runtime environment. Extending subheaps to work robustly in the presence of potentially-malicious code remains an unexplored problem. Two aspects deserve elaboration: structured use of subheaps and enforcing space limits.

First, it should not be possible for an untrusted callee to alter which subheap is active for its caller. In other words, whereas trusted code may benefit from free-form use of the subheap API, untrusted code must obey a stricter discipline of scoped subheap activation. Well-scoped primitives can be easily supplied, reducing the problem to constraining access to the full subheap API. Fortunately, restricting access to sensitive APIs has been a topic of much research from both the language and systems communities.

The second aspect is space limits. Examples in which untrusted code must be run in a partially isolated way include: auto-grading systems which must run code from students; web browsers running downloaded scripts; and cloud providers running

customer binaries. Similar issues (of excessive resource usage) also arise when code may allocate based on untrusted inputs, such as with decompressors and deserializers.

Systems software—both operating systems and virtual machines—provides virtual-memory-based mechanisms for limiting space usage. Unfortunately, such solutions also come with large overheads compared to language-based mechanisms. Building support for space limits atop subheap infrastructure may lead to lower runtime costs.

However, introducing space limits would also mandate a change in the trust model for subheaps. In particular, if subheap limits are used to limit untrusted code's resource usage, the API as described in this document is too permissive: it fails to restrict untrusted code from activating a non-limited subheap. There are also semantic changes implied by subheap limits (*e.g.* child subheaps must inherit their creator's limits). We would, in short, require a more capability-oriented API design for subheap management, with stronger notions of principals and trust. The connection is neither accidental nor subheap-specific: Yang and Mazières [YM14] observe that their design for a resource container API closely mirrors that of an information flow control API.

5.3. Stack Scanning Costs

The cost of scanning even a single stack can be a significant portion of GC effort. The garbage collection literature has explored how to minimize the cost of repeatedly scanning the program stack, using techniques such as stack markers [CHL98]. Such techniques implicitly assume a monolithic heap design, not an arbitrarily-partitioned one. Reducing the cost of stack scanning by extending stack markers to work with multiple subheaps would be a useful extension to the subheap implementation model.

5.4. Automation

The subheap design relies on the programmer to make good decisions about the creation, activation, and reclamation of subheaps. This raises two questions for future work to tackle: first, is reliance on the programmer a significant liability; second, can the computer help identify productive use of subheaps?

Reliance on non-experts On the one hand, the vast majority of programmers—those who currently ignore the GC—will be able to ignore subheaps. That is a key motivator for subheaps compared to alternatives such as memory management with substructural type systems. On the other hand, the programmers who do eventually use subheaps will almost certainly not be experts on garbage collection. It is unclear to what degree non-experts will be able to make effective use of subheaps.

Improper use of subheaps can degrade performance, for example if collection requests are triggered for large subheaps full of live data. It is easy to give the programmer concrete feedback about collection efficiency (data traced vs data reclaimed), which can reassure programmers that the collection points they've chosen are neither wasted work (due to low reclamation) nor inefficient (due to large amounts of tracing).

Inferring use of subheaps Computer assistance for using subheaps could take many forms. The ideal instantiation would be a fully automatic tool. Such a tool would take as input a codebase, perhaps augmented with a set of traces [HBM+06, Ric16], and identify where in the codebase to insert subheap API calls.

It is unclear whether such a tool is feasible. Analyzing object lifetimes [JR08, Xu13, BPSa+19] to give suggestions about where to collect or when to create new subheaps appears promising but challenging [VG17, JCMM16, BOF17]. One key technical

challenge is the cyclic dependence between collection efficiency and subheap decompositions: the value of a specific partitioning is determined by the (careful) choice of collection points it enables, and the usefulness of a given set of collection points is in turn dependent on the choice of subheap decomposition.

A particularly promising intermediate point for automation would be to focus on automation of subheap activation. This would have the human choose how to create and collect subheaps, while leaving the details of routing allocations into the right subheap to the runtime. Such automation would allow more fine-grained usage of subheaps without increasing programmer burden. The reason this division of labor seems promising is that connecting lifetime-related events—such as a browser tab closing, or a self-adjusting library's quiescence—with the idea of certain allocations becoming obsolete often relies on very high-level, often domain-specific, proof sketches. In contrast, disentangling an "interleaved" sequence of subheap-destined allocations seems more amenable to machine intervention.

Automation or assistance could take less extreme forms as well. Heap visualization [PG02] could help programmers decide which portions of the heap would be most amenable to subheap management. Subheaps work best when they reflect a human's understanding of heap structure; visualization could help programmers gain such understanding. Visualization could also help programmers evaluate the effectiveness of a particular configuration of subheaps.

CHAPTER 6 : Related Work

Chapter Contents

Subheaps sit at the intersection of many strands of the research literature. The insight that program allocation patterns can be exploited to improve GC efficiency is not new, but the design for subheaps synthesizes these insights in a novel way.

The literature has explored the benefits of program-tailored garbage collection [FT00, SK07, SBWC07, MZS09, CP15], selective choice of collection points [BVEDB05, XSaJ07, DEE+16, JCMM16], static partitioning [HDH03, GM04, XSaJJ07], dynamic partitioning [SHB+02, DFHP04, SMB04, DGK+02, UOO11, KC11, CM15], regions [Att94, Gay98, HMGJ04, CR04], and hybrids of tracing and regions [HET02, Cor06, Har06, SWB+15, XGD+15, RMAB16, NFX+16, VG17, BOF17, MHKS09] or reference counting [DB76, AP03, BM03, TAV14].

This chapter compares and contrasts subheaps with related research efforts. We begin by covering the most closely related work, highlighting how subheaps differ:

- Region-based memory management [TT97, HET02, GMJ+02, TBEH04] uses static analysis, usually in the form of a type system, to eliminate the need for garbage collection entirely. Subheaps rely on programmer guidance to avoid some pitfalls of RBMM, such as having tail calls and loop-carried dependencies without space leaks. Future work might combine the flexibility of subheaps with automation driven by static analysis.

- Generational garbage collection [LH83, Ung84, Moo84] heuristically focuses collection effort on the youngest data to gain efficiency by reducing tracing work. Older-first [SMM99] techniques use a different heuristic to efficiently handle some programs that are problematic for youngest-first collection. Subheaps give programmers non-heuristic control over heap partitioning, allowing them to focus GC effort on known-dead data or away from live data.

- M^3 [TAV14] and DSA [CP15] exhibit gains from widening the GC interface. Both rely on programmer annotations, attached to types, to improve GC efficiency. Subheaps give programmers more fine-grained control via a dynamically-executed API rather than static annotations on types.

- Yak [NFX+16] combines tracing collection and regions. Yak's regions correspond closely to subheaps. Yak's interface is more limited than subheaps, due to their differing aims: Yak aims to improve the efficiency of GC for big data applications, whereas subheaps seek to allow programmers to improve performance for a wider selection of problem domains. Both Yak and subheaps can in turn be seen as extensions of Leaky Regions [Har06].

	Subheaps	Static Regions	Generational GC
"Worst Case" Behavior	Extra copying & pressure on tenured space, and/or remembered set blowup	Space Leaks	Extra copying & pressure on tenured space, and/or remembered set blowup
When does reclamation happen?	Upon programmer request (deterministic)	At statically inferred (or checked) points, deterministically	Whenever the nursery fills up (nondeterministic)
Granularity of reclamation	Coarse	Fine	Coarse
Number of spaces	Arbitrary, determined by programmer	Arbitrary, inferred	Almost always statically fixed, usually two
Programmer "interface"	Modify source to use API	Rewrite problematic code	Command line flag tuning
Avoiding bad cases	Don't add use of subheaps (rely on GC instead)	Forced to rewrite code	Forced to rewrite code
Automatic?	No	Yes	Yes

Table 2: Comparison Matrix for Various Memory Management Approaches

- Hayes [Hay93] proposed the idea of key object opportunism, observing that clusters of objects often die all at once. Subheaps can emulate some of the beneficial properties of reference counting—in particular, immediate reclamation—by relying on programmers to identify key objects. The work of Hayes was mostly theoretical; subheaps provide an implementation and evaluation.

Table 2 provides an overview of how subheaps compare to regions and generational GC. The following sections examine in more detail the relationship between subheaps and specific variants of regions and GC.

6.1. Region-Based Memory Management

Tofte and Talpin provided the canonical instance of memory management based on regions [TT97]. Tofte's insight was that an integer-typed expression (in a pure language) might allocate arbitrary amounts of memory, but all of it could be deallocated once the result was computed.

In the Tofte/Talpin system, the heap was replaced by a stack of regions. Individual

results could be allocated into any in-scope region, and the choice of regions to use was made via type inference. When a region went out of scope, the type system guaranteed that all values allocated within the region were dead, and could be reclaimed in bulk. Thanks to this stack discipline, regions known to contain a statically bounded number of objects—so-called "finite" regions—were backed by memory allocated on the program's call stack.

Overall results, after several years of effort from Tofte and others [HET02], showed promise. In particular, the combination of region inference and GC reduced bootstrap compilation time for the ML Kit compiler from 2441 seconds to 1053 seconds, and regions often relieved the GC of 80-90% of its total workload.

Retrospectives from some of the flagship region-based projects showed mixed success; regions seemed to complement GC well, and there were occasional efficiency gains, but primarily regions were useful to reduce the load on the GC:

- The RBMM retrospective [TBEH04] says "it is not clear that infinite regions are such a good idea [...] and the experience with the garbage collector suggests that it is better to use garbage collection for objects that region inference puts into infinite regions, due to fragmentation problems."

- The authors of the Cyclone project observed [SHM+06] that while their mechanisms for safe memory management (building upon regions) usually reduced program working set sizes, there was only one instance in which overall program performance improved versus a conservative garbage collector.

Ad-hoc extensions to Tofte's region calculus, such as storage mode analysis [BTV96] and multiplicity inference analysis [Vej94], improved asymptotic performance and/or constant factors in certain circumstances. Even so, memory management based on

compiler-inferred lexically-scoped regions has well-known shortcomings, as discussed in Tofte et al's retrospective [TBEH04]:

- Data allocated within a loop and passed between iterations often must be kept live until the loop terminates. Similar problems occur for data allocated in compiler phases.
- Not all programs are amenable to region inference.
- Inference of lightly-used infinite regions can lead to (internal) fragmentation.
- Inferred `letregion` expressions can interfere with tail call optimization.
- Minor refactorings can result in large performance changes.

Much as with subheaps, Tofte and Talpin proposed region-based memory management as a compromise between fully-automatic garbage collection and fully-manual memory management. Many of region-based memory management's limitations derive from the decision to rely on automatic, compiler-driven region inference, and the decision to force region deallocation in stack order. Reliance on programmer input and arbitrary collection ordering allows subheaps to avoid some limitations of region-based memory management. In particular, subheaps can accomodate patterns of overlapping or non-lexically-scoped lifetimes, and any interference with tail calls is made explicit. However, even with programmer input, subheaps do carry limitations on the patterns of data that can be easily managed. For example, subheaps struggle to efficiently collect portions of circularly linked structures, for which the cost of remembered set maintenance can easily outweigh the reductions in tracing workload.

	Static/Dyn	Usage Model	Scoping Model	Features beyond TT
Tofte/Talpin [TT97]	Static	Type inference	Static, lexical	-
ML Kit [HET02]	Static, TT-based + GC backstop	Type inference	Static, lexical	Multiplicity Inference and Region Resetting
AFL [AFL95]	Static, TT-based	Type inference	Static, non-lex	Early dealloc, no MI
WCM [WCM00]	Static, not TT	Explicit CPS IR	Static, non-lex	all?
HMN [HMN01]	Static, not TT	Inferred IR (only first order source)	Hybrid, non-lex	Subsumes Kit & AFL
FMA [FMA06], Fluet [Flu07]	Static, not TT	Inference, explicit monadic src, sub-structural IR	Hybrid, non-lex	all?
Subheaps	Dynamic	Explicit source	Dyn, non-lex	n/a

Comparing RBMM Approaches

Gay & Aiken also propose explicit regions for use in C programs, via a language called C@ and a compiler called RC [GA98, GA01]. They use reference counting (of regions, rather than objects) to provide safety. In their original system, `deleteregion()` applied to a non-empty region is a no-op (which means it does not reclaim any space at that point; arguably, a space leak). RC changed the semantics to be a fatal error. The equivalent operation applied to a subheap will reclaim any available space, at line granularity, for reuse.

Their paper on RC [GA01] points out that some patterns exhibited by the real programs (such as the `lcc` compiler) in their benchmark suite cannot be represented accurately by the purely-static systems, which are forced to either leak space or force the programmer to rewrite their code. They also note that it is the programmer's responsibility to break region-crossing cycles before deleting regions. With the RC system, failure to break a region-crossing cycle is a fatal error. With subheaps, region-crossing cycles can be severed on an as-needed basis, and condemned sets can obviate the need to break cycles manually. The tradeoff for this flexibility is that subheaps make it easier for unexpected cyclic garbage to degrade the performance of

explicitly-invoked collections.

Gay & Aiken [GA98] studied the difficulty of programming with explicit regions, and found that the programs they studied "required only modest recoding to use regions, and the needed region organization was straightforward to derive." They observed that even when programs manipulated data in complex ways, it was not difficult to find simple and effective ways to organize that data in regions. Carrying over this line of reasoning to subheaps is straightforward; the primary extra cost incurred by subheaps is that of checking for, and recording, subheap-crossing pointers.

Cyclone [GMJ+02, HMGJ04, SHM+06] had a heap region, stack regions, and growable regions, plus unique pointers and reference-counted pointers. Growable regions did not need to have nested lifetimes. While Cyclone's machinery was powerful, it's not clear what the performance impact was. In his dissertation [Gro03], Grossman admitted that "simply using a garbage-collected heap is often as fast or faster than using growable regions."

Other Type Systems

It has long been known that substructural type systems (in particular, with linear of affine types) can reduce or eliminate the need for garbage collection. For example, Lafont [Laf88] described an abstract machine which could produce only acyclic heaps, and therefore needed no garbage collection.

Schemes to avoid garbage collection by static enforcement of rules tend to eliminate useful flexibility, such as circular data structures. Thus, general-purpose languages (such as Rust) that make use of substructural types usually retain the need for some form of automated garbage collection, either via tracing or reference counting.

Recent research has also explored the benefits of co-designing a concurrent language's type system and garbage collector, such as with the Orca project [CFD+17, FCD+18].

6.1.1. Unsafe Region Hybrids

Many programs written in unsafe languages like C can benefit from the bulk deallocation behavior of regions. In part because C does not provide memory safety or any portable way of mandating write barriers, such attempts often make it the user's responsibility to avoid use-after-free bugs.

Hanson proposed a C library for arena allocation and evaluated its benefits in the context of the `lcc` compiler [Han90]. In his scheme, as with subheaps, the programmer had to identify allocations of similar lifetimes to place within arenas. Barrett and Zorn [BZ93] extended Hanson's scheme, using profiling to automate the segregation of short-lived objects.

Berger, Zorn, & McKinley [BZM02] investigated folklore about the performance impact of custom memory allocation. They found that most custom allocators failed to outperform a well-tuned general purpose allocator. The exception was an allocator based on regions (arenas), which performed substantially better on certain applications, but suffers excess memory usage for common programming patterns such as dynamic arrays and producer-consumer designs. The same patterns cause difficulty for region-based memory management, for the same reason: a sequence of finite but overlapping lifetimes means that the region/arena is never completely empty. Subheaps avoid this pitfall, and as the experiments in Section 4 show, can still achieve equivalent performance to unsafe arena allocation for some programs.

Informed by their investigation of the efficacy of custom memory allocation schemes, Berger, Zorn, & McKinley proposed adding flexibility to arena allocation regions

with individual object deallocation, yielding **reaps**. Unlike reaps, subheaps seek to preserve memory safety and thus do not provide the flexibility of individual object deallocation.

Inoue et al [IKN09] also produced a hybrid of regions with manual deallocation. Their investigation of request-oriented region allocation in a multithreaded web server context revealed that both `malloc`/`free` and pure regions suffered from overheads. Deallocating individual objects raised the cost of allocation and reclamation, with a significant contribution being the cost of object coalescing and splitting, intented to avoid defragmentation. Meanwhile, use of traditional regions caused memory bandwidth contention as caches flushed dead objects back to RAM. Their proposed solution was to augment the `malloc`/`free` interface with a `freeAll` function to perform bulk reclamation, thereby producing a **defrag-dodging malloc** (DDmalloc). As with reaps, DDmalloc did not seek to preserve memory safety.

6.2. Garbage Collection

Generational garbage collectors have enjoyed huge success. They are effectively the baseline against which other memory management schemes are compared. However, it is well-known in the GC community that medium-lived data can be problematic for standard generational collectors: it raises copying costs when it leaves the nursery, then bloats the mature space and causes expensive full-heap collections. The literature on garbage collection has explored many ideas for how to build a better garbage collector, and many of these ideas are reflected in various facets of subheaps.

6.2.1. Phase-Aware GC

One category of related work is on garbage collectors that improve their efficiency by taking advantage of program phases. Hybrids of tracing GC and region-based memory management also fall into this category, because regions correspond to (statically identifiable) phases.

The work of Buytaert et al [BVEDB05, BVEB07] on **Garbage Collection Hints** uses profile-guided offline analysis to identify favorable collection points. Their experimental evidence shows that "garbage collection hints work well for long running applications that show some recurring phase behavior in the amount of live data. [...] Applications that do not exhibit a phased live/time function are not likely to benefit from GCH."

Subheaps aim to capture the same benefits, trading the engineering cost and complexity of profile-guided offline analysis with programmer burden. Both subheaps and GCH will trigger a subheap/nursery reclamation at the end of a phase or iteration, when live data is low. The most salient difference between the run-time behavior

of subheaps and GCH is that the latter can only trigger a collection of the nursery, which will include whatever unrelated data was present at the start of the phase (and its associated remembered set). In contrast, with careful activation of multiple subheaps, subheaps can focus effort solely on the data allocated within the dynamic extent of a phase itself, ignoring superfluous data in the rest of the heap.

Obviously, the analysis for GCH could be used to suggest potentially-beneficial points to insert subheap reclamation primitives. Determining how and where to place subheap creation and activation primitives in order to further improve the efficiency of reclamation—and avoid costly remembered set maintenance—would be an interesting avenue for future work.

In the **SEHMM** project, Stancu et al [SWB+15] explored a hybrid region/GC approach for Java. They only region-allocated objects that were provably dead at region exit. They found that roughly three quarters of memory could be region-managed. This means a smaller generational nursery can be used without sacrificing performance. With a large nursery, end-to-end speedups were small because baseline GC time was only 3.6% of total time. Subheaps identify dead data dynamically rather than statically; this brings flexibility but imposes program-dependent costs for remembered sets.

Leaky Regions proposed annotating method calls to indicate that their allocations would be garbage at method return [Har06]. Nested method calls lead to scoped region behavior. The evaluation showed that judicious annotation placement resulted in savings that outweighed the cost of the write barrier needed to prevent incorrect deallocations. Use of Leaky Regions produced significant heap size footprint reductions, and significant reductions in overhead for small nursery sizes. However, only in a few selected cases did the best annotations outperform a large nursery in total

throughput. Harris observes that the work of Buytaert is complementary to his: GCH is coarse grained, applying nursery collections to maximize reclamation yield; leaky regions are fine grained, minimizing heap footprints and maximizing memory reuse within a single collection cycle. Subheaps provide more general functionality, without the limitation to scoped regions. However, our experimental analysis showed similar tensions between improved collection efficiency and degraded mutator performance due to write barriers and locality effects.

Xian et al [XSaJJ07] observed that local and remotable objects in Java application servers have different lifetimes. These observations are mirrored by the study of Java object demographics conducted by Jones & Ryder [JR08]. The work on GC for application servers (**AS-GC**) investigated the possibility of using two nurseries in a generational collector, to avoid interference between the different sorts of allocations. Their heuristic was simple: any objects allocated during the execution of a remote method (that is, one that extends `java.rmi.Remote`) would be placed in the remote nursery. They found that AS-GC led to more frequent reclamations and higher efficiency as measured by nursery object survival rate, pause times, application-level workload sustained, and overall program throughput. A variant of subheaps with support for cross-subheap evacuation would be capable of encapsulating the domain-specific AS-GC heuristic within a library.

Xian et al [XSaJ08] also studied the correlation between server load and GC overhead. They found that increasing load on a Java application server caused the proportion of time spent on GC to spike to nearly 50%, even when running with a generational collector. The culprit was a combination of objects (such as for database connections) living longer under load, thus reducing the efficiency of minor collections, combined with paging triggered by a heap growing to accommodate the longer living

objects. Subheaps combat the former problem: they use programmer knowledge to perform "minor" collections with maximal efficiency. The paper also studied where collection could be performed most efficiently during server execution. They found that collecting when the heap was full worked well under light loads (when most objects had had sufficient time to die) but did not work well under heavy loads (due to object longevity). Under heavy loads, the most efficient time to collect was when the nursery was mostly—but not completely—full. Again, this perfectly matches the intuition for appropriate use of subheaps.

Phase-Adaptive Garbage Collection identifies program phases and preferentially invokes collection at phase boundaries [RKP09]. PAGC proposes to automatically partition the heap by dynamically monitoring application phases. They relocate objects by connectivity to form connected clusters (like the Train algorithm). They pay the cost of runtime analysis and relocation, in order to relieve the programmer of management concerns. Like GCH, they evaluate against SPECjvm98 and find that `javac` benefits the most. Across all benchmarks, their improvements to mark/cons ratios vary from -3.3% to 49.7% (average 19.9%), but some benchmarks suffer degraded performance from the extra copying done to dynamically aggregate objects by connectivity. Again, users of subheaps can face similar dilemmas. PAGC reduces time spent on GC by -4% to 41%, which translates to overall runtime improvement of -0.6% to 5.3%; GC was at most 12% of execution time to start. PAGC had relatively little effect on mutator utilization.

Preventative Memory Management [DZSO05] initiates collection at the start of program phases. In contrast to subheaps, it was based on statically modifying program binaries rather than source code. **MicroPhase** [XSaJ07], like GCH, reaps benefits from triggering deallocation at phase boundaries. They studied sim-

ilar benchmarks (SPECjvm98, SPECjbb2000, DaCapo) as many of the other listed works, and found closely matched results: some benchmarks are slightly degraded (their runtime monitor imposes 2% baseline overhead); others see moderate performance improvements; average speedup is 5% for SPECjbb2000.

Short-Term Memory for Self-Collecting Mutators [AHKS10, AHK+11] has programmers mark objects with logical expiration timestamps, allowing the runtime to collect expired objects en masse. As with subheaps, programmers must identify quiescent points in program execution and add code annotations to manage memory. Unfortunately, their approach sacrifices safety when applied to a GCed language. Their technical report specifies performance improvements in total execution time ranging from approximately +5% to -0.5%.

Yak [NFX+16] focuses on improving efficiency of GC for Big Data applications. It separates the heap into a Control Space, managed generationally, and a Data Space, managed with dynamically-sized regions. Arguably, Yak can be seen as an independently-developed multi-threaded version of Leaky Regions.

Yak bears quite some similarity to subheaps. The CS corresponds to the default subheap, and non-default subheaps are akin to DS regions. Like subheaps, Yak requires program modification to establish region boundaries and lifetimes. Yak's authors observe that (unlike traditional non-distributed applications) the programs they target effectively already identify phase structure and therefore region boundaries.

In contrast to subheap's general-purpose API, Yak adopts a simpler but more limited domain-specific interface. In particular, Yak's `epoch_begin()` combines subheap creation and activation, while `epoch_end()` combines collection, deactivation, and destruction. This slightly simpler interface limits Yak's expressiveness. First, because

there is no way to activate existing subheaps, it is impossible to separate out allocations that are known to be longer-lived than their brethren. With subheaps, such manual "pretenuring" has the potential to improve efficiency with reduced memory budgets. Instead, Yak relies on lattice-based evacuation to reduce wasteful object movement. Second, because the epoch-based API provides no way to trigger collection of an arbitrary region, Yak lacks the power of reference counting with key objects. Thus Yak cannot improve the efficiency of programs in which object lifetimes are well-known but neither generational nor nested. Important examples of such lifetimes include cache entries in memcached and tabs in a web browser.

Another difference from subheaps is that DS regions are collected only once; evacuation and region destruction go hand in hand. As a result of this restriction, Yak can use a (very slightly) simpler bump-pointer allocation scheme instead of the mark-region structure adopted for subheaps. However, the decision to collect each region exactly once also implies that Yak *must* evacuate. This in turn means that Yak, unlike subheaps, does not have a straightforward story for application in a conservative environment. It also poses difficulties for minimizing pauses in a concurrent implementation. The key is that marking can be done incrementally, unlike updating incoming references.

Broom also combines GC and RBMM, targeting distributed data processing systems based on message-passing actors [GGS+15]. Rather than using remembered sets, Broom proposes to use three types of regions (temporary, actor-scoped, and transferable) with restrictions on pointer destinations. Enforcement of the associated restrictions was left to future work. Like subheaps and Yak, Broom exposes an API that programs must be modified to use. Unlike Yak, Broom's API separates region (de)activation and destruction.

Hierarchical Memory Management for Parallel Programs applies to purely-functional fork-join parallel code [RMAB16]. The restriction to fork-join structured parallelism opens the opportunity to automatically use nested heap regions, without the need for programmer annotations. In an amusing coincidence, they refer to their heap structure as "superheaps". Memory is structured as a tree of heaps, mirroring the tree of parallel tasks. A parallel task join induces the unioning of a heap with its parent. To support bump-pointer allocation and efficient unioning, heaps are structured as a list of contiguous pages. Task stealing from a subtree is disabled while it is being GCed.

The restriction to purely functional programs brings other benefits. Unlike Yak, collections need not pause all thread stacks; only the subset of processors evaluating within a given task subtree must be scanned. Thanks to the lack of mutation implied by a (strict) purely functional language, and age-order collection between related heaps, there's no need for remembered sets or a write barrier. Also, parallel collection between and within subtrees is greatly simplified.

Subheaps support most of the operations provided by superheaps (and more). Hierarchical Memory Management (HMM) could mostly be implemented as a library atop subheaps. However, such an encoding would suffer from two sources of additional overhead. First, because subheaps lack the structured lifetimes and collection-order constraints of superheaps, they'd need the cost of write barriers and remembered sets. Under the reasonable assumption that subheaps would be joined more often than collected, work to track subheap-crossing pointers would be redundant. Second, subheaps would not enjoy the benefits of selective synchronization, which allows some threads to be entirely ignored during collection. This would hurt throughput, latency, and scalability.

So subheaps could extend HMM to impure programs, at some cost. This is moderately interesting; of more interest would be exploring ways to improve the work efficiency of GC in parallel code. HMM aims to allow parallel collection with minimal synchronization, but since it provides no interface to control where allocations end up nor when or where collections occur, HMM doesn't improve efficiency on a uniprocessor. The authors acknowledge that the questions of when and where to trigger GC are open issues. Heuristics targeting parallel code are not well explored, and heuristics developed in a serial setting may not be as effective. Letting programmers override heuristics with subheaps might then be even more compelling.

6.2.2. Generational Collectors

Rather than taking advantage of phases in particular (types of) programs, generational collection relies on an even more common property: most objects have short life spans. Generational collectors improve throughput by focusing their effort on the youngest objects. Generational garbage collection is usually co-credited to Ungar [Ung84], Moon [Moo84], and Lieberman & Hewitt [LH83]. Earlier partitioned schemes included work by Hanson, Ripley, and Griswold [Han77, RGH78].

In Lieberman & Hewitt's system, space was divided into multiple generations composed of multiple regions, each with a version number. Rather than the modern notion of a transparently-maintained remembered set, they forbade direct pointers from older to newer generations, insisting on indirection via an entry table. They suggested two possible treatments of the stack. If it is considered part of the oldest generation, it would require indirections for all its references. Alternatively, it can be included in the youngest generation, with the tradeoff that the stack must be scanned at every collection. Their paper draws the connection between entry tables and reference counts; both have trouble with cyclic garbage. However, because gener-

ations were not under programmer control, there was no possibility of emulating the benefits of reference counting. Lieberman & Hewitt raised the possibility of benefits from programmer influence over region sizing and object placement. They did not suggest programmer-triggered reclamation of individual regions/generations, nor did they explore the details of how programmers might influence generational collection or what magnitude of benefit might be obtained.

Moon's collector [Moo84] distinguished between static, ephemeral, and dynamic object lifetimes. Ephemeral objects correspond to a nursery with aging spaces (called levels), and dynamic objects to the old space. Objects were partially separated by assumed lifetime at creation, and information about an object's space and level was encoded in its address (similarly to our scheme for block metadata). However, objects were not strictly space-partitioned; objects of different lifetimes could be interleaved within a space.

Ungar [Ung84] divided the heap into a nursery, an aging semispace, and the old space. In Berkeley Smalltalk II, the old space was backed with offline storage. Ungar identified the *tenuring problem*, of data that dies after reaching the old space, as a potential impediment to the efficiency of a generational collector. One of the proposed solutions, left as future research, was "hints from the executing program."

Appel [App89] proposed a flexible nursery sizing policy, in which all space not used by the mature generation is devoted to a semispace nursery. The nature of a semispace nursery means that half of the remaining heap must be reserved as tospace. However, this is a very pessimistic assumption; the entire premise of generational GC is that most of the time, a significant fraction of the nursery will be dead! This, in turn, means that most of the tospace reserve will be wasted, reducing GC efficiency. Measured nursery survival rates in the literature vary from 20% to less than 3% for

allocation-heavy code in the SML/NJ compiler.

Guan, Srisa-an, & Jia [GSaJ09] measured several nursery sizing policies for a generational collector, and found that performance varied widely in some cases. Throughput varied by up to 36%, and MMU graphs showed widely divergent profiles for some benchmarks. Even the same benchmark can "prefer" different policies depending on workloads. For example, jbb2005-23whs achieved 30% MMU at two orders of magnitude finer granularity with an Appel nursery (HA) versus the other policies. In contrast, for jbb2005-8whs, at roughly 1 second granularity, the default adaptive policy achieved roughly 85% MMU versus roughly 15% MMU for the HA policy.

One approach to avoid the overhead of the tospace reserve is to simply reduce its size. This gives objects more time to die and makes minor collection more efficient. If the reserve turns out to be too small, a backup strategy is needed, such as compacting collection. Velasco, Olcoz, & Tirado [VOT04] measured average GC time reduction of 17% for a 2x heap, and a reduction of heap usage ranging from 19% to 40%. Overall execution time speedup was 2% on average for SPECjvm98.

Similar approaches have been explored by McGachey & Hosking [MH06], as well as Tong & Lau [TL10].

Demers et al [DWH$^+$90] detailed schemes for obtaining some of the benefits of generational collection in the context of a conservative collector. They also note the possibility of using application-provided hints (in the form of a logical timestamp) to guide collector effort. Their chosen example is to do a partial collection between two compiler phases. On that benchmark, they observed that hints produced a speedup relative to the baseline generational collector, but the non-generational collector was still faster. Their conclusion was that "with the right hints, at no loss of working set,

collection time can be greatly improved." Subheaps would enable compiler authors to experiment with domain-specific collection heuristics, such as inter-pass partial collections.

Demers et al also introduced the possibility of leaving objects marked between collections. The implementation in this dissertation uses this "sticky mark bits" technique to implement generational collection for subheaps. Subsequent work has leveraged the same idea to either reduce redundant tracing [CM15, Ric16] or improve parallel tracing [CP15, RN13].

The **Mapping Collector** [WK08] uses virtual memory to compact the tenured space in a generational collector without copying or updating pointers. This relies on the same basic phenomenon that subheaps take advantage of: objects tend to die in clumps. The Mapping Collector (MC) identifies completely-dead pages, and remaps the underlying storage to permit indefinite bump-pointer allocation. Address space is never reused, but with 64-bit pointers, virtual memory is not a scarce resource. Reclaiming chunks of memory with little live data is a trick shared by G1 and C4 [DFHP04, TIW11]; MC adopts the simpler approach of reclaiming only completely-dead pages. Their experimental evaluation shows this causes only mild fragmentation. Because MC doesn't modify live objects, it need not synchronize with the mutator to reclaim space. Concurrent marking is demarcated by stop-the-world root scans. MC provides significant improvements to throughput and MMU compared to alternative compaction algorithms such as the Compressor [KP06]. Max pause times for the concurrent MC were roughly 1 ms per 3 MB of heap.

6.2.3. Partitioned Collectors

Another category of collectors partitions the heap in ways beyond the standard generational design. These designs generally use fixed, automated, program-independent partitioning schemes, whereas subheaps provide arbitrary partionings but require programmer guidance.

Bishop developed a partitioned collector for an early capability system with a single paged linear address space, called **ORSLA** [Bis77]. Bishop's design had the programmer place objects into separate "areas" which served as both the basis for efficient paging of small objects and of incrementalizing garbage collection in a large address space. Areas also supported space quotas. Each process owned multiple areas, including an area for activation records. In ORSLA, the user bore ultimate responsibility for invoking GC. However, the system implemented automatic inter-area reference tracking (with hardware assistance), and inter-area object movement was handled by the collector. An "object mover" helped maintain cycle-completeness without requiring simultaneous collection of cyclically linked areas.

In ORSLA, direct references between areas were normally forbidden; instead, an explicit indirection called an inter-area link served as the equivalent of a remembered set. However, Bishop observed that maintenance of inter-area links (and its associated overhead) was unnecessary for short-lived computations; for such situations, Bishop proposed "cables" between areas to permit direct linkage (and, of course, force simultaneous collection). Cables were automatically constructed from short-lived to long-lived areas. Users could also explicitly create cables between areas. The design for temporary subheaps from Section 2.7.1 is a simpler, less-general reinvention of Bishop's notion of cables.

Bishop suggested that directing collection effort towards pages seeing the highest rate of modification could improve the efficiency of collection. Given ORSLA's context as an operating-system level solution, combined with the collector's integration with hardware and virtual memory, made this an attractive point in the design space. Subheaps instead rely on user input to redirect collector effort, on the assumption that "regular" garbage collection provides sufficient performance in most situations.

The core mechanisms of subheaps bear striking resemblance to ORSLA's scheme for garbage collection. Novel elements of this dissertation include: the subheap API, including dynamically-scoped subheap activation and the design for condemned sets; integration of subheaps with generational garbage collection; investigation of write barriers and compiler-driven barrier optimization; lessons learned from deployment of subheaps; careful design for efficient amortization of both allocation and collection costs; and a full implementation, with performance analysis of subheaps across multiple programs and collector reference points.

The **Mark-Copy** collector by Sachindran & Moss [SM03] reduces the space overhead associated with semispace collection. Because of the standard space-time tradeoff with GCs, this also translates to faster collections in size-constrained heaps. The core idea is to divide the heap into numbered blocks, which can be evacuated, in-order, incrementally. To do this, MC constructs (at collection time) unidirectional remembered sets between blocks, using a full-heap scan. This scheme brings both lower space overhead and the potential for smaller pauses, at the cost of extra redundant work. When used in lieu of mark-sweep in a generational collector, MC usually brings small performance advantages due to increased mutator locality from copying.

The **Memory-Constrained Copying** collector (MC^2) [SMB04] enhances MC with incremental marking, logical block numbering, and interleaved nursery & generational

collection. By carefully scheduling its collection effort, MC^2 provides throughput comparable to mark-sweep while achieving significantly reduced pause times.

Generational GC designs focus on the youngest objects, but this risks promoting objects too soon. Focusing on the oldest objects is worse: they are often immortal, meaning tracing them is wasted effort. **Older-First** [SMM99] garbage collection focuses effort on the middle class. It does so by rotating a fixed-size collection increment through the heap, moving in turn from older to younger objects. Like subheaps, this means that its write barrier must remember more pointers.

Stefanović et al's [SHB+02] analysis of (Deferred) Older-First versus Appel-style generational collectors illuminated several trends that also apply to subheap collectors. The three biggest factors for Older-First were: more frequent collections, leading to higher costs for stack scanning; lowered pause times due to avoiding whole-mature-space collection; and lower total execution time, mainly due to reduction of GC work as evidenced by lower mark/cons ratio.

At the obvious cost of requiring programmer intervention, subheaps offer the following potential improvements upon Older-First collection. First, because subheaps are flexibly-sized instead of fixed-size, they may require fewer remembered pointers. Second, because subheap collection can be triggered based on specific program behavior, rather than on the weak generational hypothesis [JHM11], fewer objects need be unnecessarily copied out prematurely.

The **Beltway** framework [BJMM02] generalizes multi-space collectors, including semispace, generational, and Older-First designs. A Beltway design has one or more belts, each composed of an ordered sequence of increments. Allocations are directed into the last increment of a particular belt. Increments are collected in-order per belt.

Promotion policies determine where evacuated objects end up. Remembered sets must track pointers from old to young belts, and from young to old increments.

Jones & Ryder advocated for lifetime-aware garbage collection in the **LACE** [JR06, JR08] project. They observed that individual allocation sites almost always allocate objects with predictable lifetimes, and suggested orienting the heap towards object deaths rather than births. Doing so would avoid wasting effort on live objects, and speed the collection of dead objects: precisely the benefits conferred by subheaps. Unlike subheaps, LACE was envisioned primarily as an automatic system, driven by traces and/or static analysis.

The **Train** algorithm [HM92, SG95, Gar05] was designed to incrementally collect the mature space in a generational collector. Use of subheaps does not preclude use of the Train (or any other sophisticated design) for the default/mature space. Ideally, there would be little benefit in using the Train to collect other subheaps—if there is enough live data at the point of collection to make the Train worthwhile, the subheap is probably being activated and/or collected at the wrong time. I haven't yet thought about the possible complications of applying Train-like techniques within individual subheaps.

The Train algorithm works by dividing the mature space into fixed-size frames (cars) organized into ordered lists called trains. Any set of prefixes of cars within trains can be collected independently, thus bounding the work needed for a single round of collection. Subheaps, in contrast, are about making collection more efficient, reducing total GC work to be done, rather than limiting the amount of work to be done in any particular increment. Reduced GC pause times are simply a beneficial consequence of efficient collection.

A subtle but important difference between the Train algorithm and subheaps is that the former, used in its intended role as the Mature Object Space for a generational collector, has an invariant that roots are processed and redirected into young space before scavenging. Subheaps have no such invariant.

Like Beltway, each car/increment has a remembered set. (Subheaps, which also have remembered sets, are more akin to trains than cars). A key challenge for both Train and Beltway is handling cyclic structures which span cars/increments. In Beltway, the solution is to add an unbounded-size fallback space to hold such structures — which implies sacrificing incrementality. In Train, once a cyclic structure is confined to a single (unreferenced) train, the whole train can be collected. However, it can take many repeated object copies to arrange that state of affairs. There is a fundamental tension between the size of a car: smaller cars bring smaller pauses, but larger cars incur less recopying to handle cyclic structures. The way subheaps resolve this tension is to observe that, when a space is known to be nearly all dead objects, cost of collection is almost entirely independent of the size of the space.

Another traditional challenge for the Train algorithm is to limit the size of remembered sets. With subheaps, the programmer has the power and the responsibility to use subheaps in ways that do not lead to remembered set blowups. Garthwaite [Gar05] tackled this and similar issues of scalability.

Klock [KC11, KI11] presented an alternate take on scalable garbage collection. Unlike the Train algorithm, **Klock's regional garbage collector** provides provable, mutator-independent bounds on pause times.

Klock and Garthwaite tackled many similar issues. One common theme was establishing careful bounds on the cost of remembered set management, especially in the

presence of popular objects. Another was pacing garbage collector work to match mutator activity.

Klock notes that "exposing the notion of regions at a level visible to the application programmer" would be an interesting avenue for future research. His discussion touches on RBMM, pre-tenuring, and (implicitly) CBGC. Subheaps might provide a flexible framework for capturing the suggested benefits.

The **Garbage-First (G1)** design [DFHP04] shares common elements with subheaps, Klock's regional collector, and the Train algorithm. All three feature a space-partitioned heap with non-unidirectional remembered sets. Like Klock's regional collector, G1 uses a concurrent snapshot-at-the-beginning (SATB) marking pass to avoid retaining circular garbage in remembered sets. In G1, the results of concurrent marking are also used to prioritize collection of regions that are mostly-garbage. The less live data in a given region, the faster it will be to collect. This is the same basic insight that subheaps rely on to boost the efficiency of collection.

Subheaps differ from G1 in several ways. First is the granularity of regions vs subheaps. Regions (or blocks) are a fixed, physical division of the heap. Subheaps are a logical division; their granularity varies according to programmer-specified directives. When subheaps comprise multiple regions, they need not record remembered sets between those regions. This can potentially decrease time and space costs for both mutator and collector.

A subtle but closely related distinction concerns allocation destination. With G1, allocation switches to a new region when the previous one fills up. Subheaps have no maximum size, and allocation destination is under programmer control. This can also lead to smaller remembered sets.

G1 uses a points-into remembered set structure (in Klock's terminology), which has the downside of potentially quadratic space usage. In the paper's tests with SPECjbb, even after specialized handling to reduce the impact of popular objects, more than one fifth of the heap was devoted to storing remembered sets. This roughly matched the total volume of live objects. This suggests that space savings from shrinking remembered sets could be significant for some programs.

NG2C [BOF17] augments G1 with user-controlled fine grained pretenuring, and provides a profile-based analysis for automatically suggesting annotations. The most fundamental difference with subheaps is NG2C's lack of support for explicit reclamation. Explicit reclamation of arbitrary subheaps enables features like emulated reference counting. NG2C omits explicit reclamation, and in return naturally integrates with a highly concurrent collector. Because NG2C's "generations" do not maintain independent remembered sets, the underlying collector's write barrier applies as-is. A third difference is whether allocations are implicitly or explicitly directed to user-defined generations/subheaps. NG2C's decision to only pretenure at explicitly-annotated allocation sites arguably reduces modularity; it means, for example, that the client of a library cannot capture the library's allocations.

Several papers have observed that object lifetime and mutation patterns are correlated. Young objects are more heavily mutated, while older objects are more stable. Copying collection is well-suited to young objects, and reference counting suits older objects. **Ulterior Reference Counting** [BM03] combines a copying nursery with a reference-counted mature space. Pointers into the nursery must be remembered as usual. Pointers from the nursery to the mature RC space can be deferred or reference counted. Mutations in the RC space are logged. Object counts in the RC space are only reconciled when the nursery is collected. URC's willingness to scan the nursery's

live space stands in sharp contrast to subheaps, which strive to avoid making the cost of collection strongly dependent on the size of live data.

Morad et al [MHKS09] also combined generations and regions, augmenting an Appel-style generational collector with a region-based heap. In their scheme, long-lived objects were pretenured into regions to avoid the cost of repeated tracing in the mature space. Profiling was chosen over manual control or static analysis for choosing which objects to region-allocate. Each region held (all) objects from a single "nested" allocation site. Reference counts were maintained for regions rather than objects within regions; unlike subheaps, they forbade partial reclamation of regions. Their scheme required an additional write barrier only to keep accurate reference counts on minor collections. Major collections traced the whole heap, thereby also dealing with cyclic garbage across regions. In their terminology, the generational heap and region-based heap were each called "sub-heaps."

The **Age-Oriented Concurrent Collector** [PPB05] and the **generational sliding views** collector [AP03] both use a mark-sweep nursery with a reference-counted mature space, each concurrently collected. A mark-sweep nursery was chosen to avoid the complexities of concurrent copying. This dissertation only explores homogeneous subheaps; exploring heterogeneous memory management schemes such as these examples could be interesting future work.

Baecker [Bae72] explored the possibility of augmenting Algol 68's garbage collector with a region-like construct he called *areas*. The core benefits of such a scheme overlap with those of subheaps: efficient bulk reclamation and the ability to collect the most relevant subset of the heap. Baecker's work predates the development of generational collection. His paper identifies safe independent collection as a key problem (which would eventually be addressed by the invention of remembered sets and write

barriers). He also raised the potential for the cost of area management to outweigh the savings in garbage collection costs, a concern which has been borne out by our experimental analysis with subheaps.

6.2.4. Static Analysis for GC

Most work on garbage collection relies on generic heuristics and runtime interpositioning. Some work relies on static analysis of particular programs to make garbage collection more efficient.

Deca aims to reduce GC overhead in distributed data processing pipelines [LSZ+16]. It applies a code transformation, supported by static analysis, to enable bulk management of data, similar to region-based memory management. Rather than modify the garbage collector, their system represents objects in data processing pipelines via packed byte arrays. This brings three benefits: decreased GC cost, due to reduced object count and elimination of read/write barriers; increased data density and cache behavior, due to removal of pointers; and reduced serialization costs, because objects are represented in serialized form.

Deca's experimental evaluation exhibited exhilarating results: compared with the G1 garbage collector, Deca reduced GC costs from 8.8x to 90.5x, and improved end-to-end performance from 2.2x to 346x. The authors do not break down how much of their performance improvements originate from hardware effects or serialization avoidance versus reduced GC pressure. Comparing subheap's performance on similar programs would be quite interesting.

Connectivity-Based Garbage Collection uses a static analysis to determine a conservative partitioning of runtime objects, such that certain partitions are known

to never point to other partitions [HHDH02]. This permits sound collection of subsets of the heap without needing to record every partition-crossing reference. By forcing some partitions to be collected together, a connectivity-based collector can elide all write barriers and remembered sets entirely. This is precisely akin to a generational collector's requirement to collect the nursery before the old space in exchange for a unidirectional remembered set.

Viewed through the lens of their work, subheaps allow the programmer to exploit some of the benefits of connectivity-based collection. Programmers get the task of identifying fruitful partitions, while the implementation repurposes well-researched infrastructure for generational garbage collection. CBGC uses topological ordering to avoid the need for write barriers or remembered sets, which are potentially costly in time and space, respectively. Subheaps make it the programmer's responsibility to choose productive and efficient partitionings of the heap. CBGC uses static analysis to determine allocation placement; subheaps extend programmer choices through dynamic slices of program execution.

Ruggieri & Murtagh proposed a static analysis to identify object lifetimes which could be linked to procedure activation records [RM88]. They relied on inter- and intra-procedural data-flow analysis rather than the type-based approach of Tofte & Talpin. In their scheme, each procedure call could create a "sub-heap" to hold objects which would have been statically shown to die before the procedure call returns.

6.2.5. Program-Specific Garbage Collection

Some collectors discussed thus far have been designed to work well with particular classes of programs (such as distributed big data computations with Yak) or program features (such as phase recognition in MicroPhase). Some collectors go a step further,

and provide ways to customize their behavior to an individual program. In contrast to work such as subheaps that proposes a richer interface between the programmer and the collector, program-specific GC works without human interaction, usually driven by static analysis or data collected from GC traces.

Object colocation [GM04] is related to **pretenuring** [Har00] and connectivity-based collection. The common idea is to allocate objects to different spaces in order to improve GC efficiency. Such techniques could be applied to subheaps which contain a mix of live and dead objects, reducing the live data size at collection time. Both are, effectively, forms of program-specific garbage collection.

In the context of a generational collector, pretenuring [Har00] uses static or dynamic analysis to decide, on a per-call-site basis, whether to allocate into the nursery or the mature space. Intuitively, allocating long-lived objects into the mature space avoids both the cost of write barriers and the cost of copying out of the nursery. Colocation [GM04] makes the decision on a per-object-allocated basis; the same call site can allocate into the mature space or nursery, depending on what other objects it will be connected to. Viewed through the lens of colocation, subheap allocation allows call sites to vary allocation choices, but generally in a more coarse-grained way. Similarly to colocation, and unlike regions or CBGC, the only negative consequence of "bad" subheap usage is degraded performance and not compromised program correctness.

In the ideal case, colocation selects exactly those objects that would survive a nursery collection. With subheaps, the goal is to select only those objects that will not survive subheap reclamation. In either case, the effect in the limit is the same: subheap/nursery reclamations can be free because everything they contain is dead.

The use of `coalloc` imposed a 1% average baseline performance penalty. Colocation

generally reduced the volume of data copied out of the nursery by 50-75%. However, the total benefits obtained were mixed. When augmenting a bounded-size nursery, colocation shaved an average of 30% from GC time in a 3x heap, and consistently improved total performance by about 3%. In an Appel-style nursery, the performance gain from colocation was drowned out by a slowdown caused by degraded locality in a mark-sweep mature space. With a copying old space, colocation had no appreciable net performance impact. Colocation required heuristics to prevent certain benchmarks from severely degrading performance.

Marion, Jones, & Ryder [MJR07] suggest a method for driving pretenuring: they combine a simple program analysis to identify coding idioms, and a database matching idioms to object lifetimes. Examples of such idioms include (i) classes with only final fields; (ii) immutable objects; or (iii) classes encoding reference cells. Observed reductions in GC time for some SPECjvm98 programs ranged from 6-77%.

Rather than building a novel collector to be more responsive to a range of programs, several authors have advocated for automatic switching between a diverse collection of off-the-shelf implementations. Proposed criteria for selecting the most appropriate design include profiling [FT00], machine learning [SBWC07], automated heuristics or explicit annotations [Pri01, SKB04, SK07], and accounting for program inputs [MZS09]. Less drastically, one might "customize" a single off-the-shelf collector by automatic parameter tuning, as done by Lengauer and Mössenböck [LM14].

Jacek et al [JCMM16] investigated the potential for collection schedule to optimize GC performance, using traces of the DaCapo benchmark suite. Their model showed average improvements of 10% in a 2x heap, with overall improvements between 5% and 20% for a generational collector. Since they did not consider customized heap partitioning schemes, the efficiency of their collections are limited by the live heap

(nursery) size at the time of collection, generally between 9MB and 11MB. Follow-on work by Jacek & Moss [JM19] investigated the potential for machine learning to improve the selection of GC collection trigger points.

6.2.6. Widening the GC Interface

The concept of augmenting the GC interface is not new to this work. However, the possibility is under-explored in the literature. This subsection covers the few projects which have experimented with user input to guide the garbage collector's actions.

Hayes proposed **key object** opportunism [Hay93], theorizing that the lifetimes of groups of objects could be tied to the lifetime of a single key object within the group, rather than being tied to program phase behavior. His primary heuristic for identifying key objects was those that appeared in remembered sets; other possibilities he considered included random selection, stack reachability, or programmer hints. Subheaps provide a means for programmers to take advantage of key objects.

Multi-Memory-Management (M^3) by Terei, Aiken, & Vitek [TAV14] was a significant spiritual inspiration. In particular, their proposed idea of widening the memory management interface to boost the efficiency of garbage collection is clearly reflected in subheaps. Their focus was to combine tracing and reference counting, relying on the programmer to identify a productive division of labor between the two schemes. Like subheaps, they argued for an opt-in approach, augmenting rather than replacing the underlying garbage collector. However, the details of the two efforts widely differ. Whereas subheaps use a dynamically-scoped runtime API to focus collection effort on logical subsets of the heap, they use a statically-scoped, type-driven approach to choose between collection strategies (tracing vs reference counting).

Customizable Memory Management [Att94, AFI98] (CMM) supported multiple

heaps with an interface roughly isomorphic to the basic interface sketched in Section 2.3 (sans the distinction between `condemn` and `collect`). Because CMM was implemented as a library-only solution, it did not—could not—make use of write barriers. Due to the limits of working without compiler support, CMM had to rely on either (fallible) user input or (expensive) cross-heap tracing to identify cross-heap pointers. Use of write barriers for subheaps improves safety, reduces programmer burden, and enables the hybrid tracing/reference-counting scheme of Section 3.5.

This dissertation explores homogeneous subheaps, in contrast to the heterogenous heaps explored by CMM. An earlier implementation of subheaps mirrored CMM's usage of C++ classes to implement multiple types of (sub)heap, specifically "coarse grained" subheaps which managed memory at the granularity of frames rather than lines. This scheme was abandoned in part because it produced significant fragmentation issues on the SAC benchmarks due to the inability for different granularities of subheap to share memory.

Project Snowflake [PVV+17] augmented a concurrent garbage collector with safe deletion of individual objects. The Snowflake programming model guarantees type and memory safety by making attempted use of a deleted object result in a thrown exception. However, careless use of `delete` can result in a program encountering unexpected exceptions. In contrast, the subheap API only affects performance, and cannot by itself alter program control flow.

Data Structure Aware collection, proposed by Cohen & Petrank [CP15], also gave the programmer a wider interface to GC. Their design brings together sticky mark bits[1] and BiBoP-style [DEB94, JHM11] type-segregated allocation. Programmers must identify data structure node types with annotations; node instances are then

[1]The sticky mark bit is called the member-bit in their work.

allocated on segregated pages, and stickily marked until the programmer signals their removal. Two major benefits accrue from this scheme. First, segregated contiguous allocation enhances locality for mutator and collector alike. Second, and more importantly, sticky-marked nodes can be treated as roots, providing a rich source of mostly-independent work packets for a parallel tracer.

Subheaps provide a flexible mechanism to segregate data structure nodes. In DSA, payloads are not segregated. With subheaps, the programmer could segregate all nodes together, or pursue a finer granularity of heap decomposition with nodes and payloads combined into many small subheaps. However, the interaction of subheaps, sticky marking, and parallel tracing remains entirely unexplored.

The **Deferred Collector** by Ricci [Ric16] makes GC more efficient by letting the programmer identify key objects. Ricci adds one function, `collectInfrequently(obj)`, to the runtime. The transitive closure of objects passed to that API are labeled as deferred. When the GC runs, it ignores deferred objects. Thus, when a large subgraph is protected by a deferred key object, the cost of tracing the subgraph can be amortized over many collections. Instead of adding a second API to notify the collector that a key object is about to die, Ricci simply ignores the deferred bits on every n^{th} collection.

Subheaps can provide similar benefits in theory, with two important disclaimers. First, subheaps currently provide no direct analogue of Ricci's infrequent collections; in practice, programmers would likely need to find points at which to (perhaps conditionally) trigger collection for designated objects. Second, the allocation-oriented nature of the subheap API may not be as convenient as Ricci's direct transitive-closure semantics. With Ricci's API, objects allocated from many places, at many different times, can be managed uniformly with a single API call. In contrast, the

subheap API might require much more careful management of subheap activations in order to physically segregate the objects that a transitive closure would identify.

Deallocation Hints by Reames and Necula [Rea13, RN13], focuses the collector's attention on objects that should soon die. This is in stark contrast to the prior two designs, which instead identify long-lived objects. Reames targets C and C++, in which most code already identifies (likely!) object lifetime end points with the `free()` function call.

In Reames's Hinted Collector, objects passed to `free()` are labeled as condemned but not immediately reclaimed. At collection time, every non-condemned object is assumed to be live and treated as a root. Like the Data Structure Aware collector, the root set is thus greatly expanded. A primary benefit of Deallocation Hints is to improve the performance of parallel marking. The key insight is that marking has serial complexity proportional to the reachable volume of live data, but parallel complexity proportional to the depth of the object graph.

As with DSA, subheaps (with the extension for short-lived subheaps from 2.7.1) can in theory provide similar abilities to redirect collector effort away from presumed-live data, but the interaction of subheaps and parallel marking has not yet been investigated.

These three collector designs can all be cast in a unified framework of **programmer-controlled mark bits**. The Hinted Collector allocates objects as marked, and allows the user to unmark candidates for reclamation. The Data Structure Aware collector segregates data structure nodes and stickily marks them. The Deferred Collector stickily marks the transitive closure of programmer-identified key objects. Note that while they use the same underlying mechanism, there is a distinction in purpose:

the Deferred collector speeds serial reclamation by amortizing the cost of marking stable subgraphs, while the other two improve parallel marking by providing a greatly expanded root set.

6.2.7. Others

Compact Regions [YCA+15] are a runtime-backed library for the Glasgow Haskell Compiler environment. Programmers modify their programs to make use of the library's API, which copies immutable objects to be arranged contiguously in memory. The stated purpose is to enable efficient network transmission of serialized data by avoiding the pointer-chasing of traditional serialization. To do so, compact regions must be self-contained; objects within a region can refer to other objects within the region, but cannot refer to objects outside of the region. As a pleasant side effect, this means that the garbage collector can skip over compact regions when processing the rest of the heap.

Like with subheaps, moving long-lived data from the mature space to a compact region can reduce GC costs. However, subheaps and compact regions differ in two important aspects. First, compact regions cannot contain mutable references (since mutation could introduce region-crossing pointers). Second, like Tofte/Talpin-style regions, objects within a compact region are only collected en masse with their containing region; unlike subheaps, there is no reclamation of space from dead objects within a compact region.

The **Clustered Collector** [CM15] dynamically identifies "head" objects along with disjoint sets reachable from each head. Head objects are similar to key objects, in that reachability of the head implies reachability of the cluster. However, unlike key objects, non-reachability of the head does not imply non-reachability of the cluster.

The Clustered Collector intermittently runs a dynamic analysis to identify clusters. In their prototype, cluster identification is several times more expensive than a full heap trace. Clusters do not grow once identified, but can be dissolved by writes that would shrink the cluster. Undissolved clusters need not be re-traced at each collection. In a sense, the Clustered Collector dynamically identifies fine-grained sub-generations. Like generations, clusters have remembered sets and can reduce tracing effort wasted on stable data.

Some key differences with subheaps include: automated runtime analysis vs programmatic source-embedded API; dissolving clusters for reduced floating garbage vs deferring subheap collection for efficiency; size-limited clusters vs dynamically-scoped swathes of allocations.

Firefox partitions its JavaScript heap into "**compartments**" [WGW+11, WLBF16]. Segregation of objects into compartments is done automatically, via script document origin, and cross-compartment references are handled via wrappers (which help enforce web-specific access policies) instead of remembered sets.

Contaminated GC [CPC00] focuses on object death points rather than age. It essentially dynamically determines the appropriate region to own each object.

Baker [Bak92] proposed a scheme called **lazy allocation**. The idea is to do all allocation on the runtime stack, evacuating live objects at method return and heap writes. Subheaps apply a similar principle, but at coarser programmer controlled granularity, and reserve stack allocation for allocations with statically-known lifetime. This avoids the need for evacuation or read/write barriers for the stack (assuming stacks are not themselves heap-allocated).

Corry extends on Baker's ideas in several ways with **Optimistic Stack Alloca-**

tion [Cor06, Cor04]. Corry splits the traditionally unified Algol-like stack into separate control and data stacks. He suggests triggering deallocation (from the data stack) at loop boundaries instead of at call/return. Doing so provides better support for factory methods and constructors, and reduces the need for inlining. Corry also avoids the need for Baker's read barrier by scanning stack frames.

6.2.8. GC Scheduling

Subheap collection is usually explicitly invoked. This touches on the subfield of garbage collection scheduling, results from which suggest the potential for subheaps to improve the performance of garbage collected systems.

Perhaps the simplest and most widely-available API for programmers to influence GC is the `System.gc()` API found in Java. However, it provides the programmer no guarantees or even mental model for what portion of the heap it will collect. An application that intends to trigger a minor collection but instead gets a major collection is unlikely to achieve its performance goals. This uncertainty and imprecision severely limits the usefulness of `System.gc()` in practice.

Terei & Levy explored the impact of programmatic GC scheduling with **Blade** [TL15]. They extended the garbage collector with a very simple API: a callback for the user to request deferral of garbage collection, and a function to trigger a (whole-heap) GC. Deferred GC allows domain-specific coordination logic to run, improving system performance. HTTP servers can notify load balancers that they will be temporarily out of service while GC takes place. In a distributed system, follower nodes can coordinate with the leader to avoid correlated pauses that can prevent the system from reaching consensus. Their evaluation showed huge improvements in worst-case request latency, on the order of 100x, without degrading throughput. Other big

data systems such as **Cayuga** [DGP+07] have also found benefits from the ability to explicitly kick off GC on a per-thread basis.

There is also work on having the runtime pick opportune moments to trigger garbage collection. Prior coverage of related work on generational collectors touched on some examples, mostly focused on improving overall work efficiency by collecting when the nursery is mostly-dead. Some work has investigated scheduling GC collections to avoid interference and reduce overheads in Big Data workloads [MAHK16, MHAK15]. Their results show improvements in both throughput and tail latencies.

Finally, some work has looked at GC scheduling with an eye towards latency rather than throughput. For example, to maintain responsiveness and avoid dropped frames, V8 schedules increments of collection work to occur in the spare milliseconds between frame rendering [DEE+16]. Subheaps give programmers more control over when to start collections, but do not yet allow programmers to schedule partial collection increments, as would likely be required to explore this sort of latency oriented domain-specific optimization.

CHAPTER 7 : Conclusion

Tracing garbage collection and manual memory management seem to be diametrically opposed: one provides safety, modularity, and decent performance without human input; the other provides control and performance for a wider range of programs. Subheaps represent one particular hybrid between these two extremes, for enhancing programmer control over tracing garbage collection without sacrificing safety.

Of course, the concrete design for subheaps explored in this dissertation is only an early foray into the design space. Subheaps are more broadly an idea about how to expand the "range" of a GC by careful focusing of collection effort. The implementation and evaluation sections have revealed both strengths and weaknesses of the concrete design explored in this dissertation. The future work section casts some light on how the underlying idea might evolve—in design and implementation—to ameliorate those shortcomings. These threads may now be joined to support an informed perspective on subheaps. We begin by summarizing the benefits and drawbacks to the implemented version of subheaps.

Benefits For certain important classes of programs, such as software caches, subheaps can provide large reductions in GC overhead and divorce GC costs from heap size. Subheaps allow programmers to treat garbage collection costs as *partially* deterministic (with important caveats) rather than completely nondeterministic. A consequence of such control is that minor alterations to source programs—such as the insertion or removal of a call to `subheapCollect`—can produce drastic changes to GC costs, for better or worse. Finally, by managing space at the granularity of lines, subheaps can be used in situations where other region-flavored allocators would fail due to excessive space overhead.

The reasons behind the sharp division in performance on caches between subheaps and non-augmented tracing GC, illustrated in Section 4.4, are worth exploring. Caches have several features that make them well-suited to subheap management. First, isolation between cache entries minimizes remembered set maintenance costs and eliminates the possibility of creating subheap-cyclic garbage. Second, the lifetime of a cache entry is well-defined; this makes it relatively obvious how and where to modify the cache to use subheaps. Third, cache entry lifetimes are not statically predictable, which makes it difficult for traditional heuristic approaches like generational collection to work efficiently. Fourth, the line-based granularity of subheaps minimizes space overheads for cache entries that may vary widely in size. Caches also have aspects favoring subheaps which are not related to their heap structure. Traditional workloads have fixed heap requirements, so the only effect of overprovisioning the heap is to reduce GC throughput overhead. In contrast, extra space can be used to improve the behavior of the cache itself, creating an impetus to operate with a tight heap. Subheaps thus resolve the tension between GC throughput and cache hit rate.

Costs Subheaps impose a variety of costs for the four primary constituents of a language ecosystem: language designers, language implementors, library authors, and programmer-users. Designers must account for the tensions between subheaps and certain language features, particularly mutable globals, shared-memory multithreading, and sophisticated control flow. Language implementors face extra costs, such as being forced to adopt data flow analysis to scrub stack slots before reclamation, and to implement the static analysis and optimizations needed for subheap write barriers. While sometimes inconvenient, these costs must be paid only once, and can be amortized over all users of subheaps. In contrast, some costs recur for all users. Both library authors and their consumers must contend with the runtime costs of the

subheap write barrier and the potential costs in human effort of deciding how and where to make use of subheaps. Any design and implementation for a subheap-like system must carefully balance best-case versus common-case costs and overheads. In particular, making "regular" tracing collection slower to speed up user-triggered collections is a very risky proposition.

There is also a cost in human effort to measure the benefit (or lack thereof) for envisioned usages of subheaps. For example, the Reynolds2 benchmark (Section 4.3.2) exhibits a "sweet spot" of subheap sizing. It would currently be the human's responsibility to experimentally find this favorable configuration. Lastly, subheaps require that programs be modified to use the subheap API. Through the API, the choice of when, where, and whether to collect can be driven by arbitrary Turing-complete code fragments. The burden of wisely applying this power, in trading off performance versus clarity, modularity, and maintainability, rests on programmers.

In contrast to the success of subheaps for software caches, it is instructive to consider the failure of subheaps to improve the SAC library for self adjusting computation. SAC features heap structures—cyclic data structures with heterogeneous object lifetimes—that prevent subheaps from operating with minimal overhead. SAC also involves code features, such as intricate use of higher-order recursive functions, that complicate the task of orchestrating the proper intended usage of the subheap API.

Alternatives Subheaps do not exist in a vacuum. Individual performance problems that might be solved with subheaps can be resolved, or at least ameliorated, with other approaches. Examples include overprovisioning the heap to reduce throughput costs, or reducing latency via concurrent collection. Part of the appeal of subheaps is to provide a single principled mechanism rather than relying on a collection of

disparate approaches. The tradeoff for such generality is that subheaps are often not the lowest-effort solution to any particular isolated GC problem:

- Throughput problems caused by GC thrashing are exacerbated by small heap sizes. Often, the cheapest and easiest solution to reduce the runtime cost of GC is to simply provision more memory.

- Subheaps must conservatively update remembered sets; thus even if subheap-crossing pointers die by the time collection is triggered, the cost of recording intermediate states can outweigh the savings from cheaper collections. In contrast, memory management techniques based on static analysis need not dynamically monitor intermediate program states.

- Finally, latency issues with a stop-the-world GC model tend to be binary: partial reductions in GC load imply that unacceptable pauses merely occur less often, rather than not at all. While subheaps can eliminate GC work for very simple programs, doing the same for large and complex programs does not seem feasible. In contrast, concurrent collectors can more easily control latency by pacing the mutator.

An interesting direction for future work would be to combine subheaps with complementary approaches. For example, a system that relies primarily on substructural types for memory management, such as Rust, might find it easier to gain benefits from use of subheaps because superfluous short-lived data would be "filtered out" by the language's static discipline.

A Snapshot of Opinion This dissertation has explored the ideas behind, the implementation of, and the consequences from using subheaps. Subheaps were developed in pursuit of an ambitious goal: a general-purpose GC augmentation to make safe

languages performance-competitive with their unsafe counterparts. While subheaps have demonstrated strong results for some important classes of programs, it remains unclear whether the crown has yet been won. I believe that most implementors of new languages, given the costs and benefits outlined above, would make an engineering decision not to adopt subheaps *in their current form*. This is in no small part due to the minimialist focus of the design for subheaps explored in this dissertation. Of more interest to potential adopters is whether a spiritual successor to subheaps might prove sufficiently useful. Here I remain optimistic that future research could alter the tradeoffs involved to make usage in practice, well, practical.

The Future Expansions of the subheap API might lead to improvements for programs not examined in this dissertation. One example would be to experiment with support for evacuation between subheaps. This could reduce instances of wasted work in repeated collection of a subheap with mixed-lifetime objects. Such extensions would also enable more direct comparisons of the efficacy of subheaps versus related work such as Yak.

The desire to better model or duplicate existing GC designs could also motivate other API variants. The model of subheaps discussed so far is "flat" in that subheaps are each separate entities, with no notion of heirarchy or grouping. The related work section pointed out that many pieces of related work depend on known relations between different pieces of the heap. Future work might provide a way to encode such relations to increase subheap's flexibility, allowing higher level policies like generational collection to be composed and configured as needed. Extending the API to make subheap management easier, such as by declaring logical groupings of subheaps, would likewise have the potential to improve flexibility, performance, and debuggability. A key challenge would be to balance the dynamic flexibility offered by subheaps with

the desire to reduce overhead by leveraging statically-known invariants.

Another interesting point for future exploration would concern how objects are routed into different subheaps. A key design point of the current design for subheaps is to establish a regime of dynamic scoping for the active allocator. This makes it easy for the client of a library to capture allocations made by an oblivious library in a coarse-grained manner. Future work might investigate how to better leverage knowledge held by library authors, and how to coordinate knowledge across boundaries in a minimally invasive way.

The version of subheaps presented in this document is type-agnostic, meaning that the subheap mechanism is unaffected by the types assigned to (or inferred for) program values. This is in contrast to work on region-based memory management, which drives allocation decisions *through* the type system, or work like M^3, which derives allocation decisions *from* (programmer annotations on) types. Future work on subheaps might explore variants which are not type-agnostic. Relying on type information to drive subheaps could reduce programmer burden and increase the analyzability of specific subheap configurations. In contrast to type inference for region-based memory management, which must generally compute conservative static approximations to value lifetimes, subheaps could make use of best-effort data from dynamic observations to help identify profitable subheap configurations.

APPENDIX

Segregated-Metadata Write Barrier An earlier iteration of the subheap prototype, discussed in Section 2.7.3, represented the object-to-subheap mapping in per-line metadata instead of storing subheap identifiers in object headers. This produced a significantly more costly barrier, commemorated in Figure 30.

A.1. The Fundamental Bottleneck of Garbage Collection

Computer systems get huge speedups on common-case workloads via clever tricks such as caching, branch prediction, and speculative execution. Likewise, garbage collectors speed up common-case workloads via clever tricks such as parallel marking and generational collection. Sadly, these tricks are not complementary: when tracing happens, it tends to be slow.

The heart of any garbage collector is the marking loop.[1] A recursive exploration of most object heaps generates an "unfriendly" pattern of memory accesses: uncacheable, hard to prefetch [CHV04, GBF07], and only sometimes amenable to parallel speedups. Marking shows significant degradation in instruction-level throughput under high-frequency profiling [YBM15]. Allocation, in contrast, has an easy-to-handle sequential write pattern. The specific penalty for tracing relative to allocation depends on details of hardware and heap layout, but historically it has been nearly an order of magnitude slower [BCR03a, BCM04a, BCM04b, XSaJ08].

This asymmetry is partially counterbalanced by smarter heuristics, such as generational collection, which decouple GC throughput from tracing throughput in the

[1] Even in reference counting collectors, pause times are dominated by recursive unmarking of large structures.

```
        movq    %rsi, %r15
        movq    %rdi, %r14
        movq    %r14, %rcx
        shrq    $11, %rcx
        movq    heap_metadata_arr(%rip), %rbx
        movabsq $68719476720, %r13      # imm = 0xFFFFFFFF0
        andq    %r13, %rcx
        movb    8(%rbx,%rcx), %al
        movq    (%rbx,%rcx), %r12
        cmpb    $1, %al
        je      .LBB61_2
#                       ^^ fast path insn 11
        movzbl  %al, %edi
        movq    %r12, %rsi
        movq    %r14, %rdx
        callq   heap_for_slowpath
        subq    $8, %rsp
        movq    %rax, %r12
.LBB61_2:
        movq    %r15, %rax
        shrq    $11, %rax
        andq    %r13, %rax
        movb    8(%rbx,%rax), %cl
        movq    (%rbx,%rax), %rax
        cmpb    $1, %cl
        je      .LBB61_4
        movzbl  %cl, %edi
        movq    %rax, %rsi
        movq    %r15, %rdx
        callq   heap_for_slowpath
        subq    $8, %rsp
.LBB61_4:                               # %heap_for_wb.exit25
        testq   %r12, %r12
        je      .LBB61_7
        cmpq    %rax, %r12
        je      .LBB61_7
#                         ^^ fast path insn 22
        movq    %r12, %rdi
        movq    %r15, %rsi
        callq   subheap_write_barrier_slowpath
        subq    $8, %rsp
.LBB61_7:                               # %cleanup
        movq    %r14, (%r15)
#                            ^^ fast path insn 23
```

Figure 30: Costlier subheap write barrier (asm).

common case. When a generational nursery is collected with little live data, the ratio of reclaimed data to traced data can be quite large. In one study, nursery survival rates for Java application benchmarks averaged 8.5% [BGH+06]. This low survival rate effectively permits GC throughput to be 12 times faster than the limit imposed by the cost of tracing.

On the flip side, repeated tracing of long-lived objects can lead to asymptotic slowdowns. This is most often seen in tight heaps [BM08]. Absolute mark/cons ratios can vary by orders of magnitude (0.03 to 19.44) in standard benchmark suites [SMS+12]. Observed GC time usually tracks variation in mark/cons ratios [SHB+02, FMB03, BHM+07]; Section 4 explores this connection, and how it can fail, in more detail.

When it comes time to collect the mature space, and concerns over latency come into play, nursery survival rates are irrelevant. With a simple stop-the-world collector, tackling the mature space incurs pause times proportional to the live data in the heap. Since a significant fraction of the mature space tends to remain live, the cost of a full-heap collection is proportional to heap size.

The precise cost depends on both tracing rate and heap residency statistics. But an estimate of **one second pauses per gigabyte of heap space** seems to be supported by experimental results from the last decade; in practice, things are rarely better and often worse [AS16, DSaC02, GTSS11, GKS06, KC11, LP06, Mül14, NFX+16, SMB04, SHB+02].

Meanwhile, heap sizes have grown just as furiously as allocation rates. The reason is simple: RAM sizes have grown exponentially. In 2019, desktops have dozens of gigabytes of RAM, and servers can have hundreds of gigabytes or even terabytes. Pauses to fully collect such heaps can destroy interactivity in desktops and interfere

with distributed algorithms in the cloud, since a multi-minute pause is not appreciably different than a crashed node. And yet, even as heap sizes rise, there is simultaneous pressure to expand the reach of garbage collection into latency-sensitive domains, demanding ever more stringent pause limits of milliseconds or even microseconds.

A great deal of great work in the literature has gone into addressing pause times. Incremental collection can spread out the work of a major collection into multiple smaller pauses. Concurrent collection can perform work on a background thread, hopefully allowing the mutator to go about its business unimpeded by garbage collection pauses. However, such techniques generally impose significant losses in throughput. And they do not erase the fundamental tension: allocating is much easier than tracing. This tension creates a lurking performance cliff.

Given some amount of spare resources, in the form of extra cores and memory, a concurrent collector will be able to recycle memory at some particular sustainable rate. By dint of the space-time tradeoff, either increasing allocation rate or decreasing heap size will increase collector workload. When the collection workload pushes past the collector's throughput limit, the collector has no choice but to pause the mutator until memory can be reclaimed. If this occurs, the illusion of "free" garbage collection is shattered. So the worst-case performance of concurrent collection is difficult to reason about in a principled way, because small shifts in heap size or allocation rate can raise pause times by orders of magnitude [KC11, PD00]. Some concurrent collectors will "pace" the mutator to ensure that it doesn't overwhelm the collector. This avoids the latency cliff in exchange for accentuating throughput overhead.

BIBLIOGRAPHY

[ABBT06] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. [An Experimental Analysis of Self-Adjusting Computation]. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 96–107, New York, NY, USA, 2006. ACM.

[Aca05] Umut Acar. [Self-Adjusting Computation]. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[Ada07] Sébastien Adam. [Bounded Frame, Cycle, and Large Object Handling in Generational Older-First Garbage Collection]. Master's thesis, Université du Québec À Montréal, 2007.

[ADM98] Ole Agesen, David Detlefs, and J. Eliot Moss. [Garbage Collection and Local Variable Type-precision and Liveness in Java Virtual Machines]. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 269–279, New York, NY, USA, 1998. ACM.

[AFI98] Giuseppe Attardi, Tito Flagella, and Pietro Iglio. [A Customisable Memory Management Framework for C++]. *Software: Practice and Experience*, 28(11):1143–1184, 1998.

[AFL95] Alexander Aiken, Manuel Fähndrich, and Raph Levien. [Better Static Memory Management: Improving Region-based Analysis of Higher-order Languages]. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 174–185, New York, NY, USA, 1995. ACM.

[AHK+11] Martin Aigner, Andreas Haas, Christoph M. Kirsch, Michael Lippautz, Ana Sokolova, Stephanie Stroka, and Andreas Unterweger. [Short-term Memory for Self-collecting Mutators]. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 99–108, New York, NY, USA, 2011. ACM.

[AHKS10] Martin Aigner, Andreas Haas, Christoph M Kirsch, and Ana Sokolova. [Short-term memory for self-collecting mutators-revised version]. Technical report, Tech. Rep. 2010-06, Department of Computer Sciences, University of Salzburg, 2010.

[AP03] Hezi Azatchi and Erez Petrank. [Integrating Generations with Advanced Reference Counting Garbage Collectors]. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 185–199, Berlin, Heidelberg, 2003. Springer-Verlag.

[App89] A. W. Appel. [Simple Generational Garbage Collection and Fast Allocation]. *Softw. Pract. Exper.*, 19(2):171–183, February 1989.

[AS16] Khaled Alnowaiser and Jeremy Singer. *Topology-Aware Parallelism for NUMA Copying Collectors*, pages 191–205. Springer International Publishing, Cham, 2016.

[Att94] Giuseppe Attardi. [A Customizable Memory Management Framework]. In *Proceedings of the 6th USENIX C++ Technical Conference*, USENIX C++ Technical Conference, Berkeley, CA, USA, 1994. USENIX Association.

[Bae72] H. D. Baecker. [On a Missing Mode in ALGOL 68]. *SIGPLAN Not.*, 7(12):20–30, December 1972.

[Bak92] Henry G. Baker. [CONS Should Not CONS Its Arguments, or, a Lazy Alloc is a Smart Alloc]. *SIGPLAN Not.*, 27(3):24–34, March 1992.

[BCM04a] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. [Myths and Realities: The Performance Impact of Garbage Collection]. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM.

[BCM04b] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. [Oil and Water? High Performance Garbage Collection in Java with MMTk]. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[BCR03a] David F. Bacon, Perry Cheng, and V. T. Rajan. [Controlling Fragmentation and Space Consumption in the Metronome, a Real-time Garbage Collector for Java]. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 81–92, New York, NY, USA, 2003. ACM.

[BCR03b] David F. Bacon, Perry Cheng, and V. T. Rajan. [A Real-time Garbage Collector with Low Overhead and Consistent Utilization]. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Pro-*

gramming Languages, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[BCR04] David F. Bacon, Perry Cheng, and V. T. Rajan. [A Unified Theory of Garbage Collection]. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 50–68, New York, NY, USA, 2004. ACM.

[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.

[BHM+07] Stephen M. Blackburn, Matthew Hertz, Kathryn S. McKinley, J. Eliot B. Moss, and Ting Yang. [Profile-based Pretenuring]. *ACM Trans. Program. Lang. Syst.*, 29(1), January 2007.

[Bis77] Peter Boehler Bishop. [Computer systems with a very large address space and garbage collection]. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, Boston, MA, USA, 1977.

[BJMM02] Stephen M Blackburn, Richard Jones, Kathryn S. McKinley, and J Eliot B Moss. [Beltway: Getting Around Garbage Collection Gridlock]. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 153–164, New York, NY, USA, 2002. ACM.

[BM03] Stephen M. Blackburn and Kathryn S. McKinley. [Ulterior Reference Counting: Fast Garbage Collection Without a Long Wait]. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 344–358, New York, NY, USA, 2003. ACM.

[BM08] Stephen M. Blackburn and Kathryn S. McKinley. [Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance]. In *Proceedings of the 29th ACM SIGPLAN Conference*

on Programming Language Design and Implementation, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.

[Bob80] Daniel G. Bobrow. [Managing Reentrant Structures Using Reference Counts]. *ACM Trans. Program. Lang. Syst.*, 2(3):269–273, July 1980.

[Boe] Hans-J. Boehm. [An Artificial Garbage Collection Benchmark]. `http://www.hboehm.info/gc/gc_bench.html`.

[BOF17] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. [NG2C: Pretenuring Garbage Collection with Dynamic Generations for HotSpot Big Data Applications]. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 2–13, New York, NY, USA, 2017. ACM.

[BPSa+19] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. [Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications]. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 28:1–28:16, New York, NY, USA, 2019. ACM.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. [From Region Inference to Von Neumann Machines via Region Representation Inference]. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 171–183, New York, NY, USA, 1996. ACM.

[BVEB07] Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and Koen Bosschere. [GCH: Hints for Triggering Garbage Collections]. In Per Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 74–94. Springer-Verlag, Berlin, Heidelberg, 2007.

[BVEDB05] Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. [Garbage Collection Hints]. In *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'05, pages 233–248, Berlin, Heidelberg, 2005. Springer-Verlag.

[BZ93] David A. Barrett and Benjamin G. Zorn. [Using Lifetime Predictors to Improve Memory Allocation Performance]. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 187–196, New York, NY, USA, 1993. ACM.

[BZM02] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. [Reconsidering Custom Memory Allocation]. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 1–12, New York, NY, USA, 2002. ACM.

[CFD+17] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. [Orca: GC and Type System Co-design for Actor Languages]. *Proc. ACM Program. Lang.*, 1(OOPSLA):72:1–72:28, October 2017.

[CHL98] Perry Cheng, Robert Harper, and Peter Lee. [Generational Stack Collection and Profile-driven Pretenuring]. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 162–173, New York, NY, USA, 1998. ACM.

[CHV04] Chen-Yong Cher, Antony L. Hosking, and T. N. Vijaykumar. [Software Prefetching for Mark-sweep Garbage Collection: Hardware Analysis and Software Redesign]. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 199–210, New York, NY, USA, 2004. ACM.

[CM15] Cody Cutler and Robert Morris. [Reducing Pause Times with Clustered Collection]. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 131–142, New York, NY, USA, 2015. ACM.

[Cor04] Erik Corry. [Stack Allocation for Object-Oriented Languages]. Master's thesis, Department of Computer Science - Daimi, University of Aarhus, 2004.

[Cor06] Erik Corry. [Optimistic Stack Allocation for Java-like Languages]. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 162–173, New York, NY, USA, 2006. ACM.

[CP15] Nachshon Cohen and Erez Petrank. [Data Structure Aware Garbage Collector]. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 28–40, New York, NY, USA, 2015. ACM.

[CPC00] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. [Contaminated Garbage Collection]. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 264–273, New York, NY, USA, 2000. ACM.

[CR04] Sigmund Cherem and Radu Rugina. [Region Analysis and Transformation for Java Programs]. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 85–96, New York, NY, USA, 2004. ACM.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. [An Efficient, Incremental, Automatic Garbage Collector]. *Commun. ACM*, 19(9):522–526, September 1976.

[DEB94] R. Kent Dybvig, David Eby, and Carl Bruggeman. [Don't Stop the BiBOP: Flexible and Efficient Storage Management for Dynamically-Typed Languages]. Technical report, Indiana University Computer Science Department, 1994.

[DEE+16] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. [Idle Time Garbage Collection Scheduling]. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 570–583, New York, NY, USA, 2016. ACM.

[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.

[DGK+02] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. [Thread-local Heaps for Java]. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 76–87, New York, NY, USA, 2002. ACM.

[DGP+07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. [Cayuga: A General Purpose Event Monitoring System.]. In *CIDR*, pages 412–422. www.cidrdb.org, 2007.

[DSaC02] L. Dykstra, W. Srisa-an, and J. M. Chang. [An Analysis of the Garbage Collection Performance in Sun's HotSpot Java Virtual Machine]. In *Conference Proceedings of the IEEE International Performance, Computing, and Communications Conference (Cat. No.02CH37326)*, pages 335–339, 2002.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. [Combining Generational and Conservative Garbage Collection: Framework and Implementations]. In *Proceedings of the*

17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM.

[DZSO05] Chen Ding, Chengliang Zhang, Xipeng Shen, and Mitsunori Ogihara. [Gated Memory Control for Memory Monitoring, Leak Detection and Garbage Collection]. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 62–67, New York, NY, USA, 2005. ACM.

[FCD+18] Juliana Franco, Sylvan Clebsch, Sophia Drossopoulou, Jan Vitek, and Tobias Wrigstad. [Correctness of a Concurrent Object Collector for Actor Languages]. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 885–911, Cham, 2018. Springer International Publishing.

[FCJH16] Henrique Ferreiro, Laura Castro, Vladimir Janjic, and Kevin Hammond. [Kindergarten Cop: Dynamic Nursery Resizing for GHC]. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 56–66, New York, NY, USA, 2016. ACM.

[Flu07] Matthew Fluet. [Monadic and Substructural Type Systems for Region-Based Memory Management]. PhD thesis, Cornell University, 2007.

[FMA06] Matthew Fluet, Greg Morrisett, and Amal Ahmed. [Linear Regions Are All You Need]. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 7–21, Berlin, Heidelberg, 2006. Springer-Verlag.

[FMB03] Kenneth W. Fiduk, Kathryn S. McKinley, and Stephen M. Blackburn. [Not Just Yet - Giving Objects Time to Die for Garbage Collection]. Technical report, The University of Texas at Austin, 2003.

[FT00] Robert Fitzgerald and David Tarditi. [The Case for Profile-directed Selection of Garbage Collectors]. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, pages 111–120, New York, NY, USA, 2000. ACM.

[GA98] David Gay and Alex Aiken. [Memory Management with Explicit Regions]. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 313–323, New York, NY, USA, 1998. ACM.

[GA01] David Gay and Alex Aiken. [Language Support for Regions]. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language*

Design and Implementation, PLDI '01, pages 70–80, New York, NY, USA, 2001. ACM.

[Gab] Richard P. Gabriel. [The Rise of Worse Is Better]. `https://www.dreamsongs.com/RiseOfWorseIsBetter.html`.

[Gar05] Alexander T. Garthwaite. [Making the Trains Run on Time]. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 2005.

[Gay98] David Gay. [Memory Management with Explicit Regions]. PhD thesis, University of California, Berkeley, CA, USA, 1998.

[GBF07] Robin Garner, Stephen M. Blackburn, and Daniel Frampton. [Effective Prefetch for Mark-sweep Garbage Collection]. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 43–54, New York, NY, USA, 2007. ACM.

[GGS+15] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingan, Derek Murray, Steven Hand, and Michael Isard. [Broom: Sweeping out Garbage Collection from Big Data Systems]. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 2–2, Berkeley, CA, USA, 2015. USENIX Association.

[GI18] Hugo Musso Gualandi and Roberto Ierusalimschy. [Pallene: A Statically Typed Companion Language for Lua]. In *Proceedings of the XXII Brazilian Symposium on Programming Languages*, SBLP '18, pages 19–26, New York, NY, USA, 2018. ACM.

[GKS06] Giorgos Gousios, Vassilios Karakoidas, and Diomidis Spinellis. [Tuning Java's Memory Manager for High Performance Server Applications]. In Alexios Zavras, editor, *Proceedings of the 5th International System Administration and Network Engineering Conference SANE 06*, pages 69–83. NLUUG, Stichting SANE, May 2006.

[GM04] Samuel Z. Guyer and Kathryn S. McKinley. [Finding Your Cronies: Static Analysis for Dynamic Object Colocation]. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 237–250, New York, NY, USA, 2004. ACM.

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. [Region-based Memory Management in Cyclone]. In *Proceedings of the ACM SIGPLAN 2002 Conference on Pro-*

gramming Language Design and Implementation, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM.

[Gou18] Isaac Gouy. [The Computer Language Benchmarks Game]. `https://benchmarksgame-team.pages.debian.net/`, 2018.

[Gro03] Dan Grossman. [Safe Programming at the C Level of Abstraction]. PhD thesis, Cornell University, Ithaca, NY, USA, 2003.

[GSaJ09] Xiaohua Guan, Witawas Srisa-an, and Chenghuan Jia. [Investigating the Effects of Using Different Nursery Sizing Policies on Performance]. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 59–68, New York, NY, USA, 2009. ACM.

[GTSS11] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. [Assessing the Scalability of Garbage Collectors on Many Cores]. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11, pages 7:1–7:5, New York, NY, USA, 2011. ACM.

[HA08] Matthew A. Hammer and Umut A. Acar. [Memory Management for Self-Adjusting Computation]. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 51–60, New York, NY, USA, 2008. ACM.

[HAC09] Matthew A. Hammer, Umut A. Acar, and Yan Chen. [CEAL: A C-based Language for Self-Adjusting Computation]. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 25–37, New York, NY, USA, 2009. ACM.

[Han77] David R Hanson. [Storage Management for an Implementation of SNOBOL4]. *Software: Practice and Experience*, 7(2):179–192, 1977.

[Han90] D. R. Hanson. [Fast Allocation and Deallocation of Memory Based on Object Lifetimes]. *Software: Practice and Experience*, 20(1):5–12, 1990.

[Har00] Timothy L. Harris. [Dynamic Adaptive Pre-tenuring]. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, pages 127–136, New York, NY, USA, 2000. ACM.

[Har06] Tim Harris. [Leaky Regions: Linking Reclamation Hints to Program Structure]. Technical report, Microsoft Research, June 2006.

[Hay93] Barry Hayes. [Key Objects in Garbage Collection]. PhD thesis, Stanford University, Stanford, CA, USA, 1993. UMI Order No. GAX93-17774.

[Hay97] Barry Hayes. [Ephemerons: A New Finalization Mechanism]. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 176–183, New York, NY, USA, 1997. ACM.

[HBM+06] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. [Generating Object Lifetime Traces with Merlin]. *ACM Trans. Program. Lang. Syst.*, 28(3):476–516, May 2006.

[HDH03] Martin Hirzel, Amer Diwan, and Matthew Hertz. [Connectivity-based Garbage Collection]. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 359–373, New York, NY, USA, 2003. ACM.

[HDH+18] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Monal Narasimhamurthy, and Dimitrios J. Economou. [Fungi: Typed Incremental Computation with Names]. *CoRR*, abs/1808.07826, 2018.

[HET02] Niels Hallenberg, Martin Elsman, and Mads Tofte. [Combining Region Inference and Garbage Collection]. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 141–152, New York, NY, USA, 2002. ACM.

[HHDH02] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. [Understanding the Connectivity of Heap Objects]. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 36–49, New York, NY, USA, 2002. ACM.

[Hir04] Martin Hirzel. [Connectivity-based Garbage Collection]. PhD thesis, University of Colorado, Boulder, CO, USA, 2004.

[HKHF14] Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. [Adapton: composable, demand-driven incremental computation]. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 156–166, 2014.

[HM92] Richard L. Hudson and J. Eliot B. Moss. [Incremental Collection of Mature Objects]. In *Proceedings of the International Workshop on Mem-*

ory Management, IWMM '92, pages 388–403, London, UK, UK, 1992. Springer-Verlag.

[HMGJ04] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. [Experience with Safe Manual Memory-management in Cyclone]. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 73–84, New York, NY, USA, 2004. ACM.

[HMN01] Fritz Henglein, Henning Makholm, and Henning Niss. [A Direct Approach to Control-Flow Sensitive Region-based Memory Management]. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 175–186. ACM, 2001.

[IKN09] Hiroshi Inoue, Hideaki Komatsu, and Toshio Nakatani. [A Study of Memory Management for Web-based Applications on Multicore Processors]. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 386–396, New York, NY, USA, 2009. ACM.

[JCMM16] Nicholas Jacek, Meng-Chieh Chiu, Benjamin Marlin, and Eliot Moss. [Assessing the Limits of Program-specific Garbage Collection Performance]. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 584–598, New York, NY, USA, 2016. ACM.

[JDK] [Java HotSpot VM Options]. `http://www.oracle.com/technetwork/articles/java/vmoptions-jsp-140102.html`.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[JM19] Nicholas Jacek and J. Eliot B. Moss. [Learning when to Garbage Collect with Random Forests]. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019, pages 53–63, New York, NY, USA, 2019. ACM.

[JR06] Richard Jones and Chris Ryder. [Garbage Collection Should Be Lifetime Aware]. In Olivier Zendra, editor, *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*, pages 182–196, Nantes, France, July 2006.

[JR08] Richard E. Jones and Chris Ryder. [A Study of Java Object Demographics]. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 121–130, New York, NY, USA, 2008. ACM.

[KC11] Felix S. Klock, II and William D. Clinger. [Bounded-latency Regional Garbage Collection]. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 73–84, New York, NY, USA, 2011. ACM.

[KI11] Felix S Klock II. [Scalable garbage collection via remembered set summarization and refinement]. PhD thesis, Northeastern University, 2011.

[KP06] Haim Kermany and Erez Petrank. [The Compressor: Concurrent, Incremental, and Parallel Compaction]. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, New York, NY, USA, 2006. ACM.

[Kus15] Bradley C. Kuszmaul. [SuperMalloc: A Super Fast Multithreaded Malloc for 64-bit Machines]. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 41–55, New York, NY, USA, 2015. ACM.

[Laf88] Yves Lafont. [The linear abstract machine]. *Theoretical Computer Science*, 59(1):157 – 180, 1988.

[Lev14] Jacob Leverich. [Mutilate]. `https://github.com/leverich/mutilate`, 2014.

[LH83] Henry Lieberman and Carl Hewitt. [A Real-time Garbage Collector Based on the Lifetimes of Objects]. *Commun. ACM*, 26(6):419–429, June 1983.

[LK14] Jacob Leverich and Christos Kozyrakis. [Reconciling high server utilization and sub-millisecond quality-of-service]. In *Proceedings of the Ninth European Conference on Computer Systems*, page 4. ACM, 2014.

[LM14] Philipp Lengauer and Hanspeter Mössenböck. [The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors]. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 111–122, New York, NY, USA, 2014. ACM.

[LP06] Yossi Levanoni and Erez Petrank. [An On-the-fly Reference-counting Garbage Collector for Java]. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, January 2006.

[LSZ+16] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. [Lifetime-based Memory Management for Distributed Data Processing Systems]. *Proc. VLDB Endow.*, 9(12):936–947, August 2016.

[LW10] Ruy Ley-Wild. [Programmable Self-Adjusting Computation]. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2010.

[MAHK16] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. [Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications]. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 457–471, New York, NY, USA, 2016. ACM.

[McC60] John McCarthy. [Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I]. *Commun. ACM*, 3(4):184–195, April 1960.

[Mem18] [Memcached]. `https://memcached.org`, 2018.

[MH06] Phil McGachey and Antony L. Hosking. [Reducing Generational Copy Reserve Overhead with Fallback Compaction]. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 17–28, New York, NY, USA, 2006. ACM.

[MHAK15] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. [Trash Day: Coordinating Garbage Collection in Distributed Systems]. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[MHKS09] Ronny Morad, Martin Hirzel, Kelliot K. Kolodner, and Mooly Sagiv. [Efficient Memory Management for Long-Lived Objects]. Technical report, IBM Research Division, 2009.

[MJR07] Sebastien Marion, Richard Jones, and Chris Ryder. [Decrypting the Java Gene Pool]. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 67–78, New York, NY, USA, 2007. ACM.

[Mol15] Matthew Robert Mole. [A study of thread-local garbage collection for multi-core systems]. PhD thesis, University of Kent, 2015.

[Moo84] David A. Moon. [Garbage Collection in a Large LISP System]. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM.

[Mül14] Andreas Müller. [Controlling GC pauses with the Garbage-First Collector]. `http://blog.mgm-tp.com/2014/04/controlling-gc-pauses-with-g1-collector/`, 2014.

[MZS09] Feng Mao, Eddy Z. Zhang, and Xipeng Shen. [Influence of Program Inputs on the Selection of Garbage Collectors]. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 91–100, New York, NY, USA, 2009. ACM.

[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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 349–365, GA, 2016. USENIX Association.

[NGB16] Diogenes Nunez, Samuel Z. Guyer, and Emery D. Berger. [Prioritized Garbage Collection: Explicit GC Support for Software Caches]. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 695–710, New York, NY, USA, 2016. ACM.

[PD00] Tony Printezis and David Detlefs. [A Generational Mostly-concurrent Garbage Collector]. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, pages 143–154, New York, NY, USA, 2000. ACM.

[PG02] Tony Printezis and Alex Garthwaite. [Visualising the Train Garbage Collector]. In *ACM SIGPLAN Notices*, volume 38, pages 50–63. ACM, 2002.

[PPB05] Harel Paz, Erez Petrank, and Stephen M. Blackburn. [Age-Oriented Concurrent Garbage Collection]. In *Proceedings of the 14th International Conference on Compiler Construction*, CC'05, pages 121–136, Berlin, Heidelberg, 2005. Springer-Verlag.

[Pri01] Tony Printezis. [Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment]. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, Berkeley, CA, USA, 2001. USENIX Association.

[PVV+17] Matthew Parkinson, Dimitrios Vytiniotis, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Dylan McDermott, Aaron Blankstein, and Jonathan Balkind. [Project Snowflake: Non-blocking Safe Manual Memory Management in .NET]. *Proc. ACM Program. Lang.*, 1(OOPSLA):95:1–95:25, October 2017.

[PZM+10] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. [Schism: Fragmentation-tolerant Real-time Garbage Collection]. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM.

[Rea13] Philip Reames. [Hinted Collection]. Master's thesis, University of California, Berkeley, 2013.

[RGH78] G. D. Ripley, R. E. Griswold, and D. R. Hanson. [Performance of Storage Management in an Implementation of SNOBOL4]. *IEEE Trans. Softw. Eng.*, 4(2):130–137, March 1978.

[Ric16] Nathan P. Ricci. [Determining When Objects Die to Improve Garbage Collection]. PhD thesis, Tufts University, Medford, MA, USA, 2016.

[RKP09] Y. Roh, J. Kim, and K. H. Park. [A Phase-Adaptive Garbage Collector Using Dynamic Heap Partitioning and Opportunistic Collection]. *IEICE Transactions on Information and Systems*, 92:2053–2063, 2009.

[RM88] C. Ruggieri and T. P. Murtagh. [Lifetime Analysis of Dynamically Allocated Objects]. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 285–293, New York, NY, USA, 1988. ACM.

[RMAB16] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. [Hierarchical Memory Management for Parallel Programs]. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 392–406, New York, NY, USA, 2016. ACM.

[RN13] Philip Reames and George Necula. [Towards Hinted Collection: Annotations for Decreasing Garbage Collector Pause Times]. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[SA00] Zhong Shao and Andrew W. Appel. [Efficient and Safe-for-space Closure Conversion]. *ACM Trans. Program. Lang. Syst.*, 22(1):129–161, January 2000.

[SBF+10] Jennifer B. Sartor, Stephen M. Blackburn, Daniel Frampton, Martin Hirzel, and Kathryn S. McKinley. [Z-rays: Divide Arrays and Conquer Speed and Flexibility]. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 471–482, New York, NY, USA, 2010. ACM.

[SBM14] Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. [Fast Conservative Garbage Collection]. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 121–139, New York, NY, USA, 2014. ACM.

[SBWC07] Jeremy Singer, Gavin Brown, Ian Watson, and John Cavazos. [Intelligent Selection of Application-Specific Garbage Collectors]. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 91–102, New York, NY, USA, 2007. ACM.

[SBYM13] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. [Taking off the Gloves with Reference Counting Immix]. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 93–110, New York, NY, USA, 2013. ACM.

[SG95] Jacob Seligmann and Steffen Grarup. [Incremental Mature Garbage Collection Using the Train Algorithm]. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 235–252, London, UK, UK, 1995. Springer-Verlag.

[Sha15] Rifat Shahriyar. [High Performance Reference Counting and Conservative Garbage Collection]. PhD thesis, The Australian National University, 2015.

[SHB+02] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. [Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine]. In *Proceedings of the 2002*

Workshop on Memory System Performance, MSP '02, pages 25–36, New York, NY, USA, 2002. ACM.

[SHM+06] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. [Safe Manual Memory Management in Cyclone]. *Sci. Comput. Program.*, 62(2):122–144, October 2006.

[SJBL10] Jeremy Singer, Richard E. Jones, Gavin Brown, and Mikel Luján. [The Economics of Garbage Collection]. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 103–112, New York, NY, USA, 2010. ACM.

[Sjö14] Andreas Sjöberg. [Evaluating and improving remembered sets in the HotSpot G1 garbage collector]. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2014.

[SK07] Sunil Soman and Chandra Krintz. [Application-Specific Garbage Collection]. *Journal of Systems and Software*, 80(7):1037–1056, 2007.

[SKB04] Sunil Soman, Chandra Krintz, and David F. Bacon. [Dynamic Selection of Application-Specific Garbage Collectors]. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 49–60, New York, NY, USA, 2004. ACM.

[SM03] Narendran Sachindran and J. Eliot B. Moss. [Mark-copy: Fast Copying GC with Less Space Overhead]. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 326–343, New York, NY, USA, 2003. ACM.

[SMB04] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. [MC^2: High-performance Garbage Collection for Memory-constrained Environments]. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 81–98, New York, NY, USA, 2004. ACM.

[SMM99] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. [Age-based Garbage Collection]. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 370–381, New York, NY, USA, 1999. ACM.

[SMS+12] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z. Guyer. [New Scala() Instance of

Java: A Comparison of the Memory Behaviour of Java and Scala Programs]. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 97–108, New York, NY, USA, 2012. ACM.

[SN18] Salvatore Sanfilippo and Pieter Noordhuis. [Redis]. `https://redis.io/`, 2018.

[Ste99] Darko Stefanović. [Properties of Age-based Automatic Memory Reclamation Algorithms]. PhD thesis, University of Massachusetts, Amherst, MA, USA, 1999.

[SWB+15] Codruţ Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. [Safe and Efficient Hybrid Memory Management for Java]. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 81–92, New York, NY, USA, 2015. ACM.

[TAV14] David Terei, Alex Aiken, and Jan Vitek. [M^3: High-performance Memory Management from Off-the-shelf Components]. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 3–13, New York, NY, USA, 2014. ACM.

[TBEH04] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. [A Retrospective on Region-based Memory Management]. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.

[TIW11] Gil Tene, Balaji Iyengar, and Michael Wolf. [C4: The Continuously Concurrent Compacting Collector]. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM.

[TL10] Liangliang Tong and Francis C. M. Lau. [Exploiting Memory Usage Patterns to Improve Garbage Collections in Java]. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 39–48, New York, NY, USA, 2010. ACM.

[TL15] David Terei and Amit A. Levy. [Blade: A Data Center Garbage Collector]. *CoRR*, abs/1504.02578, 2015.

[Tov12] Jesse A Tov. [Practical Programming with Substructural Types]. PhD thesis, Northeastern University, Boston, MA, USA, 2012.

[TT97] Mads Tofte and Jean-Pierre Talpin. [Region-based Memory Management]. *Information and Computation*, 132(2):109–176, 1997.

[UJR14] Tomoharu Ugawa, Richard E. Jones, and Carl G. Ritson. [Reference Object Processing in On-the-fly Garbage Collection]. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 59–69, New York, NY, USA, 2014. ACM.

[Ung84] David Ungar. [Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm]. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM.

[UOO11] Katsuhiro Ueno, Atsushi Ohori, and Toshiaki Otomo. [An Efficient Non-moving Garbage Collector for Functional Languages]. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 196–208, New York, NY, USA, 2011. ACM.

[Vej94] Magnus Vejlstrup. [Multiplicity Inference]. Master's thesis, University of Copenhagen, København, Denmark, 1994.

[VG17] Raoul L. Veroy and Samuel Z. Guyer. [Garbology: A Study of How Java Objects Die]. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 168–179, New York, NY, USA, 2017. ACM.

[VOT04] José Manuel Velasco, Katzalin Olcoz, and Francisco Tirado. *Adaptive Tuning of Reserved Space in an Appel Collector*, pages 542–558. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[WCM00] David Walker, Karl Crary, and Greg Morrisett. [Typed Memory Management via Static Capabilities]. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, July 2000.

[Wee06] Stephen Weeks. [Whole-program Compilation in MLton]. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 1–1, New York, NY, USA, 2006. ACM.

[WGW+11] Gregor Wagner, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. [Compartmental Memory Management in a Modern Web Browser]. In *Proceedings of the International Symposium on Memory*

Management, ISMM '11, pages 119–128, New York, NY, USA, 2011. ACM.

[WK08] Michal Wegiel and Chandra Krintz. [The Mapping Collector: Virtual Memory Support for Generational, Parallel, and Concurrent Compaction]. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 91–102, New York, NY, USA, 2008. ACM.

[WLBF16] Gregor Wagner, Per Larsen, Stefan Brunthaler, and Michael Franz. [Thinking Inside the Box: Compartmentalized Garbage Collection]. *ACM Trans. Program. Lang. Syst.*, 38(3):9:1–9:37, April 2016.

[WM89] P. R. Wilson and T. G. Moher. [Design of the Opportunistic Garbage Collector]. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 23–35, New York, NY, USA, 1989. ACM.

[XGD+15] Shijie Xu, Qi Guo, Gerhard Dueck, David Bremner, and Yang Wang. [Metis: A Smart Memory Allocator Using Historical Reclamation Information]. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '15, pages 6:1–6:9, New York, NY, USA, 2015. ACM.

[XSaJ07] Feng Xian, Witawas Srisa-an, and Hong Jiang. [Microphase: An Approach to Proactively Invoking Garbage Collection for Improved Performance]. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 77–96, New York, NY, USA, 2007. ACM.

[XSaJ08] Feng Xian, Witawas Srisa-an, and Hong Jiang. [Garbage Collection: Java Application Servers' Achilles Heel]. *Science of Computer Programming*, 70(2-3):89–110, 2008.

[XSaJJ07] Feng Xian, Witawas Srisa-an, ChengHuan Jia, and Hong Jiang. [AS-GC: An Efficient Generational Garbage Collector for Java Application Servers]. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 126–150, Berlin, Heidelberg, 2007. Springer-Verlag.

[Xu13] Guoqing Xu. [Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs]. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Program-*

ming Systems Languages & Applications, OOPSLA '13, pages 111–130, New York, NY, USA, 2013. ACM.

[YBM15] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. [Computer Performance Microscopy with Shim]. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 170–184, New York, NY, USA, 2015. ACM.

[YCA+15] Edward Z. Yang, Giovanni Campagna, Ömer S. Ağacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. [Efficient Communication and Collection with Compact Normal Forms]. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 362–374, New York, NY, USA, 2015. ACM.

[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.