M^3: High-Performance Memory Management from Off-the-Shelf Components

David Terei
Stanford University
dterei@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Jan Vitek
Purdue University
jv@cs.purdue.edu

Abstract

Real-world garbage collectors in managed languages are complex. We investigate whether this complexity is really necessary and show that by having a different (but wider) interface between the collector and the developer, we can achieve high performance with off-the-shelf components for real applications. We propose to assemble a memory manager out of multiple, simple collection strategies and to expose the choice of where to use those strategies in the program to the developer. We describe and evaluate an instantiation of our design for C. Our prototype allows developers to choose on a per-type basis whether data should be reference counted or reclaimed by a tracing collector. While neither strategy is optimised, our empirical data shows that we can achieve performance that is competitive with hand-tuned C code for real-world applications.

Categories and Subject Descriptors D.3.4 Programming Languages [*Processors*]: Memory management (garbage collection); D.3.3 Programming Languages [*Language Constructs and Features*]: Dynamic storage management

General Terms Algorithms, Design, Experimentation, Languages, Performance

Keywords Memory Management; Garbage Collection; Tracing; Mark-Sweep; Reference Counting

1. Introduction

Automatic memory management, as supported by modern managed languages such as Java, Ruby and Go, offers a great deal of safety and productivity to developers. Through a simple interface, an entire class of difficult bugs is removed, improving security and reliability. Moreover, the best collectors are competitive, both in throughput and latency, with explicit memory management. However, the complexity of highly-tuned collectors and the engineering effort involved in creating them is staggering. For Java, hundreds of man years were invested into the various collectors that are part of the Hotspot virtual machine. One reason for this complexity is that most memory management strategies have pathologies, particular workloads that will make them perform sub-optimally. To avoid these, best-of-breed collectors incorporate sophisticated optimisations designed to reduce the likelihood of triggering worst case behaviour.

The price for all this is a substantial engineering cost, a cost that puts high-performance memory management out of the reach of many systems. Languages that are developed by small communities, such as Ruby, Python or R, young industrial languages such as Go, or languages such as C for which automated memory management is not the preferred route, cannot afford such sophisticated solutions. Implementations of these languages are forced to get by with basic reference counting or mark-sweep collectors. This complexity also makes tuning a modern collector to work well across a wide range of benchmarks, or to a specific application, a difficult task.

In this paper, we investigate whether this complexity is really necessary. We show that through a different (but wider) interface between the developer and garbage collector we can achieve high performance from off-the-shelf components. We modify the developer's memory management interface in two ways. Firstly, we allow multiple memory management strategies to co-exist in the same program, and secondly we give developers control of the policy decision of which program values are managed by which strategy. We refer to this interface as a multi-memory-management (M^3) system.

While this extension adds some burden at the language level, it allows for a drastic reduction in the complexity of the underlying runtime system. Instead of highly tuned collectors that require substantial compiler support, off-the-shelf collector designs can be used. Furthermore, the design we are proposing is completely opt-in. If developers specify no annotations, then a default collector manages all memory, essentially reducing to the current state of affairs.

We have implemented a prototype M^3 system for C that provides developers the choice between a naive reference counting collector, or a basic mark-sweep collector. Despite the simplicity of the components, we are able to achieve close to, or in one case better than, the performance of explicit memory management. We chose C for two reasons. First, it allows us to compare directly with carefully written manual memory management for performance-oriented applications. Second, our implementation is an existence proof that it is straightforward to build an M^3 prototype from scratch from off-the-shelf components.

Our work also suggests that through this wider interface, higher performance with less effort is achievable for language implementations willing to ask developers for more information. By allowing developers to guide the memory management policy at the language level, the collector strategy can be customized to the specific application, improving performance and avoiding any worst case behaviour that a fixed policy collector will have.

ISMM'14, June 12, 2014, Edinburgh, United Kingdom.

ACM 978-1-4503-2921-7/14/06. . . $15.00.
http://dx.doi.org/10.1145/2602988.2602995

We do not expect M^3 to be universally applicable, but believe that in the hands of more experienced developers building systems where performance is a concern, it can be simpler and achieve stronger results than current approaches. These are situations where a great deal of times is already spent tweaking the exposed garbage collector and run-time settings on deployment.

In addition to describing the design of our system, we conduct case studies of three programs and evaluate the performance of our system on each. Our contributions are as follows:

- We identify the benefits in extending the memory management interface and the requirements a new interface should satisfy.
- We detail one specific design that satisfies this interface.
- We conduct an evaluation of this design on three programs: Memcached, MOSS and a synthetic web middleware.

The paper is organized as follows. In Section 2, we motivate the problem with one particular program we believe is ill-served by current garbage collectors. In Section 3 we detail the requirements of an M^3 system and describe one design point in the space. In Section 4 we present three case studies, describing how to use an M^3 system with them and evaluate the performance impact. We describe related work in Section 5 and conclude in Section 6.

2. Background

In this section we provide some background on the performance characteristics of modern memory management systems and some further indications that a simpler implementation that provides more control to developers would have advantages.

Firstly, designers of automatic memory management implementations are interested in providing collectors that work well across a broad range of client systems and metrics of success are usually expressed in terms of average behaviour across a set of standard benchmarks [13]. And, on average modern collectors work very well. However, previous work has shown that the efficacy of a memory management system is highly dependent on the behaviour of the application and available resources [4, 19, 32]. For example, Soman et al. [32] showed in their work on the dynamic selection of the best garbage collector for a specific application that no single collector was optimal for all benchmarks. In fact, the optimal collector often varies over the life-time of the application. All of this leads to considerable variance in collector performance over the set of applications that developers care about.

Anecdotally we have heard stories of software projects that discovered after writing a large amount of code in a particular language with a managed implementation that the collector performed poorly in at least some (and sometimes, many) of the project's important use cases. Unfortunately, these projects had few options for addressing the problem, as memory managers generally expose control over only a few decisions (e.g., the size of the generations in a generational collector). Short of the hugely expensive task of rewriting the code to target a different platform with a different garbage collector, these projects are effectively stuck with serious performance issues they cannot address.

We argue that a collector built from off-the-shelf components but with a wider interface between the developer and the memory manager would help address some of these issues. By giving developers a simpler system with easier-to-understand performance, as well as more control, developers can better reason in advance about how a proposed system would perform and better address problems as they arise by changing the collector's policy.

2.1 An Example: Memcached

In this section we use Memcached as an example of the challenges facing modern memory management systems. It is also one of our case studies. Memcached is a high-performance, distributed, in-memory key-value store, widely used for the task of caching temporary data in modern web architectures [2]. For example, Facebook and Twitter make extensive use of the technology to reduce load on relational database servers and rely on a 99% hit rate to scale to the massive user bases that they serve [3]. Memcached uses an event-based architecture, scaling to a large number of cores. The server is accessed over the network using a client library. Efficient memory management is critical as Memcached deals with very large heap sizes and is used in performance sensitive situations where 99th percentile latency matters. Being able to provide consistent performance at all times and with high heap utilization is a necessity.

```
typedef struct {
    uint size;       // sizes of items
    void *free;      // free list
    uint free_size;  // total free items in list
    void **slabs;    // array of slab pointers
    uint slab_size;  // # of allocated slabs
    uint slab_limit; // size of slabs array
} slabclass_t;

// global holding our slab classes
static slabclass_t slabclass[MAX_SLAB_CLASSES];
```

Figure 1. Type and management of slab classes in Memcached.

Memcached is a challenging program for a garbage collector due to its absolute emphasis on performance and the unpredictable lifetimes of items in the system. Key-value pairs in the system are only deallocated for one of three reasons:

1. The client issues a `delete` command
2. The client issues a `set` command to update an existing item. Internally Memcached treats this as a `delete` and `new` command, never updating items in place.
3. The maximum memory usage is hit and Memcached frees the least-recently-used key-value pair to make room for the next one.

These properties generally make applying the generational hypothesis efficiently very difficult, as the lifetimes are either unpredictable or the oldest items in the system are being deallocated. Heap sizes in the 10's of GB and a requirement for low latency make the problem even more difficult. We believe these characteristics have prevented systems like Memcached from moving to safer, more productive languages.

In fact, one of our coauthors tried writing a Memcached-like system in the garbage-collected Go programming language [22] and ran into exactly these problems. By the time the problems were apparent, it was too late to consider rewriting the entire code base, so the system was modified to use a custom allocator and reference counting scheme written using unsafe primitives. This doubled the size of the code base and introduced more bugs than had been encountered in the entire development process up to that point. In this situation a simpler collector with the usual safety guarantees but more performance transparency and developer control would have been much better than the ultimate solution.

Memcached is written in C and uses explicit memory management and a few custom allocators to improve performance and handle fragmentation. For the bulk of memory, the key-value store itself, it uses a slab allocator with a number of fixed sized allocation classes for meeting requests, trading external for internal fragmentation, giving $O(1)$ allocation and deallocation routines. The code for a slab class is shown in Figure 1.

While challenging, we believe that software such as Memcached would be suitable for automatic memory management if the developer was able to express the program's properties to the memory manager. In our proposed design, the developer can specify that the memory associated with the key-value store is managed by a reference counting system while the rest of the system is managed by a tracing collector. This design is already partially captured in the Memcached code base, with manual reference counting being performed on key-value items.

3. Design

In this section we explore the design space for M^3 systems and offer one concrete design. An M^3 system must make choices on the following axes:

- Memory management strategies: multiple approaches for allocation and deallocation of memory.
- Composability: rules and restrictions for composing strategies in the same program.
- Granularity: the data items to which a strategy can be attached.
- Staging: the times at which a strategy can be selected.

These four dimensions map out the main design choices that must be addressed by a multi-memory-management proposal. We contend that with the right design, an M^3 system can be built from off-the-shelf components while still allowing developers to achieve the performance characteristics that they desire.

It is worth pointing out the relation between M^3 systems and the policies implemented by various generational garbage collectors. Generational collectors offer more than one strategy for allocating and recovering memory, e.g., a young generation using a semi-space copying collector with a bump-point allocator and an old generation managed using a mark-sweep collector with free-lists for allocation. However, the decision of when to move memory from one generation to the other is not exposed to the developer and each object can only change policy once. Instead, after-the-fact administrators will attempt to tweak a few knobs such as generation sizing to improve throughput or latency. This work generalizes these ideas and gives developers control over the choice of strategy.

3.1 Our Design

We present a concrete design of an M^3 system inspired by our case studies. Our design targets C and strives for simplicity, both in its implementation and cognitive complexity for the developer:

- Two memory management strategies are supported: naive reference counting and tracing garbage collection.
- The strategies can be freely composed in the same computation.
- The granularity of the strategies is at the type level. We expect the common case to be that all values of the same type are managed by the same policy, but developers can choose per allocation site.
- Strategies are selected at allocation and stay in place for the lifetime of the data.

3.1.1 Granularity: Using Types

Types are a natural place to attach memory management choices and they provide a hint to the compiler to generate efficient code for memory accesses. In our design, each type defaults to tracing collection, but all types have two implicit variants that give the developer control. For example, a type `stats` can be used as follows:

	Tracing	Ref. Counting
Strategy	Batch	Incremental
Mutation cost	None	High
Throughput	Low	High
Pauses	Long	Short
Cycles?	Yes	No

Table 1. Tracing Vs. Reference Counting. Taken from Bacon et al. [7].

	Tracing	Ref. Counting
Num. of In-Bound Refs.	Large	Small
Num. of Out-Bound Refs.	Small	Large
Expected Lifetime	Short	Long
Mutation Rate of Refs.	High	Low
Utilisation of Memory	Low	High

Table 2. Ideal properties for a node to perform well with Tracing Vs. Reference Counting

```
stats     st;  // default to tracing collector
rc::stats st;  // use of reference counting
gc::stats st;  // use of tracing collector
```

Types already capture commonality in the code and this commonality usually extends to the best way to manage the underlying memory. While for some very common types, such as primitive types like `int` and `char*`, a single ideal memory management strategy doesn't exist for all values, simple type aliasing functionality is sufficient to deal with this, to allow further specialisation by use-case. We will discuss the use of types further in Section 3.1.3.

3.1.2 Memory Management Strategies

While there are countless variations on how memory can be reclaimed in a system we eventually settled on offering the choice between the two primary viewpoints in today's systems: reference counting and tracing.

The reduction of our initial scope to these two strategies is not surprising in hind-sight, especially when viewing these two strategies as duals of one-another as suggested by Bacon et al. [7]. The strength and weaknesses of each one complements the other, as shown in Table 1 where we summarise the trade-offs of each strategy and in Table 2 where we specify the kinds of nodes in a heap that work best with one strategy.

3.1.3 Implementation Considerations

For an implementation of our design a number of questions arise, primarily on what memory management code to generate for a function and how to deal with pointers crossing from one heap to the other. A schematic diagram of our design is in Figure 2. Here we outline the scope of a complete compiler, however, our current prototype does not automate everything. We also do not evaluate the performance of pointers crossing heap boundaries thoroughly at this time. Please refer to Section 4.1 for an overview of the current prototype.

Code Generation Naive reference counting requires insertion of code around pointer operations to atomically update reference counts. A tracing garbage collector does not require any barriers around pointer accesses. The simplest strategy for combining the two is to emit checks around all pointer operations. These checks perform a switch on both where the pointer resides and where it points to select the appropriate code to execute. The four choices and the corresponding actions are outlined in Table 3. While these checks are expensive, we do not expect it to be the common case:

Stored To	Points To	Action
Traced Heap	Traced Heap	Nothing
Traced Heap	RC Heap	Add to remembered set
RC Heap	RC Heap	Perform reference counting
RC Heap	Traced Heap	Perform reference counting

Table 3. Decision of code to execute for creation of new pointers

instead we have found that a division of the types into the reference counted or traced heap can be easily found that minimises or eliminates the need for the general case. However, the general case does provide a simple and safe programming model for the developer.

To improve the performance of the common case, we attack it as a simple optimisation problem: how to specialise code to one memory management strategy? Currently in our prototype this is done with a very simple optimiser: we look at all uses of a type in the code base and if it is only ever used as a traced type then we can specialise all the code for that type to tracing. If it is used as both a traced and reference counted type then we optionally issue a warning and leave the code unspecialised. This design requires whole program compilation. A straightforward extension would be to add per-module declarations that certain types are only used with a specific memory management strategy, enabling separate compilation.

Heap Boundaries The second issue that we must deal with is how to handle pointers that reside in one heap but point to values in the other heap. This boundary crossing problem is exactly the same issue dealt with in generational collectors where we must be aware of all pointers from the old generation pointing into the young generation. We have two cases to handle:

1. Pointers in the traced heap (*TH*) pointing to the reference counted heap (*RC*); and
2. Pointers in the reference counted heap pointing to the traced heap.

TH to RC Pointers For the first case, we simply update the tracing collector to be aware of reference counting and perform the appropriate decrements as a finalization step on collected objects that point into the reference counted heap. While this approach has a cost (the tracing collector now exhibits some of the behaviour of a reference counting implementation, where tracing dead objects and freeing is a potentially unbounded operation) we do not expect this to be an issue in practice. Our assumption is that pointers crossing heap boundaries will be rare and that a large number of them is a sign of an inappropriate decision by the developer on how to specialise the types.

RC to TH Pointers For the second case, we adopt a similar solution to generational collectors and make use of a remembered set for augmenting the root set of the tracing collector. We can do this efficiently as we are expecting most code to be specialised to a specific memory management strategy and as such will not pay the cost of a software or hardware write barrier for all heap operations. Instead we can simply insert the appropriate code where needed. There is a significant complication compared to the typical use of remembered sets in generational garbage collectors, which is that our remembered set can never be cleared. In a generational collector, the remembered set grows between old generation collections but is reset on each full collection. As we never wish to scan the reference counted heap, we must instead both add entries to and remove entries from our remembered sets and in the worst case they can grow unbounded. As before, we argue that such a large number of pointer crossings is a sign of an inappropriate decision by the developer on how to specialise the types.

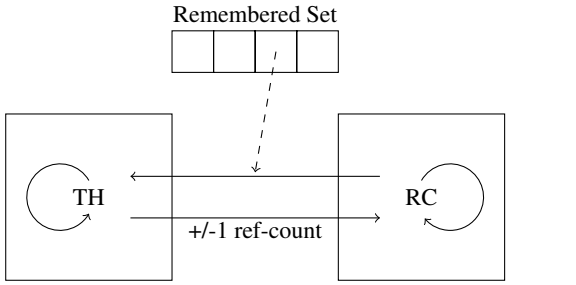

Figure 2. Schematic of our Multi-Memory-Management Design

4. Case Studies

In order to both shape our design and evaluate it, we undertook case studies of a number of programs, the results of which we explore in this section. The studies include Memcached, a high-performance in-memory key-value store, MOSS, a plagiarism detection tool and finally a synthetic example that models the behaviour of a typical middleware service in a modern web stack. For each of these studies, we specify the heap organization, how we divided it up in our system and the results of a performance evaluation.

As our implementation is for C, we evaluate each of our case studies against an explicit memory management version. This sets an accurate and very competitive baseline. The small delta between the explicit memory management version and our M^3 version shows the performance achievable with our approach. It also suggests that M^3 could be used by developers of currently explicit memory management systems to improve safety while retaining performance.

4.1 Implementation

As we have set out to explore the viability of a memory management system using off-the-shelf components, we have pursued a simple implementation. Despite this, even in its current state it is able to show competitive performance.

For our mark-sweep tracing collector we use the Boehm-Weiser garbage collector [14, 15]. For our reference counting implementation we utilize C++11's `shared_ptr` feature [25]. To handle pointers from the traced to RC heap we manually add a finalizer onto the traced object at allocation, where the finalizer knows how to decrement the RC pointer. In a complete implementation, the tracing collector would instead perform the decrement directly to the RC object during the sweep phase.

The Boehm-Weiser collector provides a mature implementation that supports parallel and incremental collection [16]. It is a conservative collector designed to operate against C/C++ programs. We use the collector in parallel mode with incremental collection disabled to keep it as simple as possible.

Our reference counting implementation using `shared_ptr`'s provides a naive reference counting collector. Shared pointers implement reference counting by adding a second level of indirection, with the reference count being stored at this second level and the managed object left unmodified. This is costly in terms of the second indirection we pay on these pointers but provides a very simple and easy way to integrate it with the rest of the system since the underlying representation of values isn't changed. This also allows easy handling of interior pointers, as a `shared_ptr` acts as a fat pointer, retaining both a pointer to the start of the object and one to the current offset.

Unless otherwise noted, memory for the mark-sweep heap is allocated using the allocator provided by the Boehm-Weiser collector while memory for the reference counted heap is allocated using

the `malloc` and `free` routines provided by GNU C library (glibc) 2.18.1 [1].

4.2 Memcached

Memcached is a high-performance, distributed, in-memory key-value store that is widely used for the task of caching temporary data in modern web architectures. We introduced Memcached in Section 2.1.

The heap of Memcached can be largely divided into two distinct components: the management of the key-value pairs themselves and the rest. The key-value pairs are managed by a typical slab-allocator design, giving $O(1)$ allocation and deallocation behaviour by trading external fragmentation for internal fragmentation. The rest consists of thread data structures, configuration details, server statistics, and connection and buffer handling. These are all managed by a variety of free-list allocators for each type. Some client commands, such as statistics collection, also generate variable and short lived data. A schematic of the heap organization can be seen in Figure 3.

4.2.1 Applying a Multi-Memory-Management System

We split the heap of Memcached in a simple way: memory retrieved from the slab allocator for use in storing key-value pairs is reference counted, while the rest of the system is traced. We achieve this by having the slab allocator return reference counted pointers and using a destructor method for them that returns memory to the appropriate slab.

This design removes the bulk of the memory from consideration by the tracing collector, which allows us to scale to very large heap sizes with high utilization while maintaining low latency and high throughput. We remove the use of free-lists for managing the rest of the memory such as connections and buffers, instead simply relying on the tracing collector. The modifications needed to the types and allocation routines to achieve this are shown in Figure 4.

The traced and reference counted heaps are largely separated in this design, with only one pointer potentially existing per connection from the traced heap to the reference counted heap. Each connection object holds a pointer to an item in the key-value store that corresponds to the item the connection is currently processing for either a `get` or `set` request. As such, the number of these pointers is in practice bound to a fairly low number as it isn't reasonable to expect a Memcached server to deal with much in excess of a few thousand connections.

4.2.2 Evaluation

We evaluate the results of our system with Memcached using two different metrics, the total throughput of the server and the worst-case latency. For Memcached we evaluate three different variants; firstly, the original, explicit memory management version; secondly, a fully traced version; and thirdly, the M^3 version. We use two servers for the evaluation, one running Memcached and one running the client. Both are 12 core 2.27GHz Intel Xeon L5640 machines connected via 10 gigabit Ethernet.

The throughput results are presented in Table 4. In this setup we initially load 100,000 key-value pairs into the server and then perform as many `get` requests as possible for a 5 minute period. For the tracing garbage collector, we invoke a collection every 10 seconds. The small heap size however makes this very cheap and indeed we can increase the frequency further with no impact. The main point is to measure the cost of using reference counting in our M^3 system.

The results demonstrate that all three versions are able to achieve the same level of throughput with approximately 1.73M requests per second.

```
// new aliases for RC.
typedef char* slab_t;
typedef slab_t* slablist_t;

typedef struct {
    uint size;        // sizes of items
    slab_t free;      // free list
    uint free_size;   // total free items in list
    slablist_t slabs; // array of slab pointers
    uint slab_size;   // # of allocated slabs
    uint slab_limit;  // size of slabs array
} slabclass_t;

// global holding our slab classes
static slabclass_t slabclass[MAX_SLAB_CLASSES];

// slab allocation
static int slabs_alloc(const unsigned int id) {
    // declare reference counted allocation
    rc::slab_t ptr;

    slabclass_t *p = slabclass[id];
    if ((mem_malloced + len > mem_limit && p->slab_size > 0) ||
        (grow_slab_list(id) == 0) ||
        ((ptr = malloc((size_t)len)) == 0)) {
        MEMCACHED_SLABS_SLABCLASS_ALLOCATE_FAILED(id);
        return 0;
    }
    p->slabs[p->slab_size++] = ptr;
    mem_malloced += len;
    MEMCACHED_SLABS_SLABCLASS_ALLOCATE(id);
    return 1;
}
```

Figure 4. Modifications to the slab Type's and allocation routines. Compare with Figure 1.

Memcached Version	Request Per Second
Explicit	1,728,240
Full Tracing	1,722,577
M^3	1,730,996

Table 4. Throughput performance of the various versions of Memcached. We performed `get` requests with a 50MB heap and manually invoked the tracing collector every 10 seconds.

For measuring latency we model the behaviour of Memcached in a production environment under increasing heap sizes. To do this, we utilise a third machine to generate load, performing 400,000 requests per second with a mixture of 90% `get` and 10% `set` requests. Performing this over a 7 minute window grows the heap in a linear fashion from nothing to 4GB. Our original client now functions as a latency sampler, sending 5,000 `get` requests per second during the experiment and recording the 20 worst latency samples. A graph of the additions into this set over the course of the experiment is shown in Figure 5. Here we let the tracing collector decide on its own when to collect. The 5 worst samples are shown in Table 5. Here we see where our M^3 system can bring real benefits, with latency largely matching the performance of explicit memory management and greatly improving on the full tracing design. It is also worth pointing out that our evaluation only went to a heap of 4GB, a small size when modern servers come with 40GB - 1TB of memory.

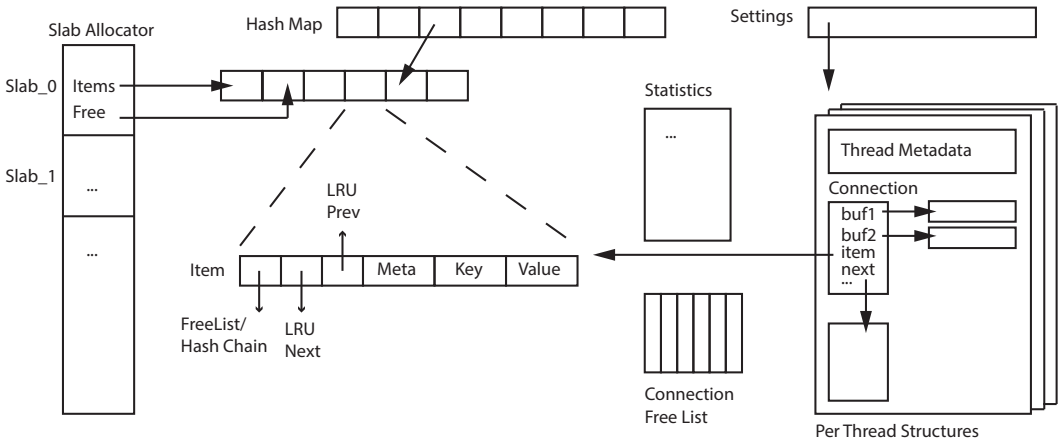

Figure 3. Memcached Heap Organization

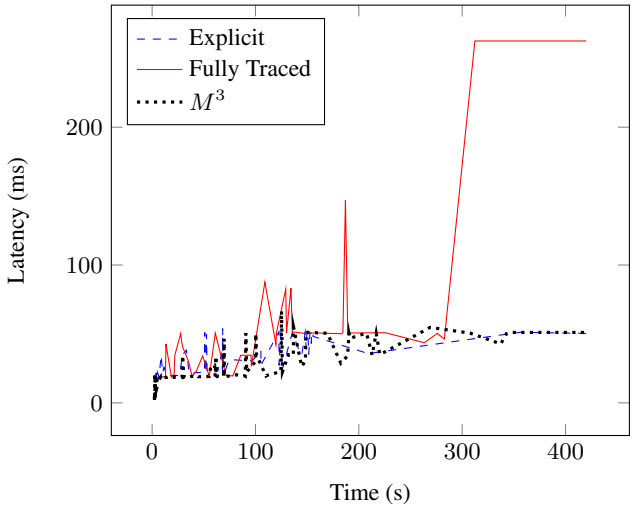

Figure 5. Latency of worst response-time as we slowly increase the heap of Memcached by performing a mix of 90% `get` requests and 10% `set` requests at an approximate rate of 400,000 requests per second.

Memcached Version	5	4	3	2	1
Explicit	51	51	54	54	66
Full Tracing	83	88	147	263	263
M^3	51	51	51	51	53

Table 5. The 5 worst latency measurements (ms) for different versions of Memcached. These were taken over the course of 7 minutes as we increased the heap from nothing to 4GB.

4.3 MOSS

MOSS is a widely-used plagiarism detection tool in use since 1994 for evaluating the similarity of programs [30]. It is written in C and uses Gay and Aiken's region memory manager [20, 21].

What we are proposing is less work for the developer than using regions, which have been used by systems developers for decades in C/C++ (often under different names, usually "arenas" or "zones"). So there is plenty of evidence that developers can deal with this. The main savings is that the developer does not need to specify where data should be freed, which is typically the developer's responsibility in region-based systems. A cost is that you do not get the locality benefits of regions—we're not doing anything about fragmentation. As future work, regions could be incorporated as another choice of memory management policy into our approach.

MOSS's heap organization consists of the following principle regions:

1. **database**: A potentially very large array storing information about the various passages of program text that MOSS is processing. An internally linked list is threaded through the array as well for various ordering operations.
2. **index**: A secondary array that stores a searchable index for efficiently looking up the passages in the text database. It is sized at 1/8th of the text database.
3. **files**: Stores information about the various files being processed.
4. **matches**: Stores information on matches among files; largely a collection of arrays and linked lists.
5. **temporary regions**: A variety of short lived regions are created for operations such as sorting subsets of much larger arrays and linked lists.

The database, index and files regions all have a lifetime equal to the program itself and as such only ever grow in size and require allocation but not deallocation. The matches and various temporary regions are much shorter lived, being created and destroyed throughout the execution. A diagram of MOSS's heap organization can be seen in Figure 6.

4.3.1 Applying a Multi-Memory-Management System

We simply put the text database into the reference counted heap and the rest is put under the control of the tracing collector. Similar to Memcached, the bulk of the data is stored in the text database and little mutation occurs here except for some re-ordering of passages using the internal linked list. It is not useful moving the index to the reference counted heap as the indexes are stored as relative array offsets, not actual pointers, so the tracing collector can deal efficiently with the index regardless of its size. This requires creating a new alias for the type involved, `passage`, but is an easy change to make given the isolated and limited way the database is used.

With this division we only end up with a few instances of pointers crossing heap boundaries. For the traced to reference counted direction, which can be handled very efficiently, a pointer for each

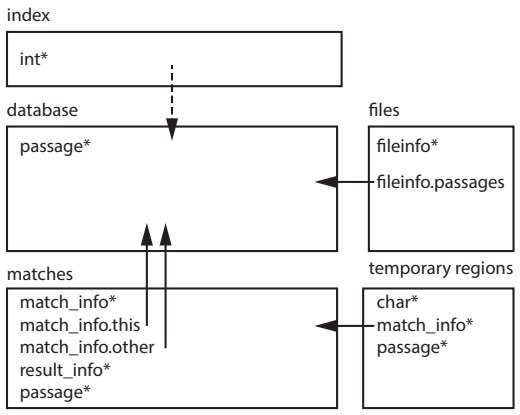

Figure 6. MOSS Heap Organization. Solid lines represent pointers, dotted lines implicit pointers through array offsets. The types stored in each region are also listed.

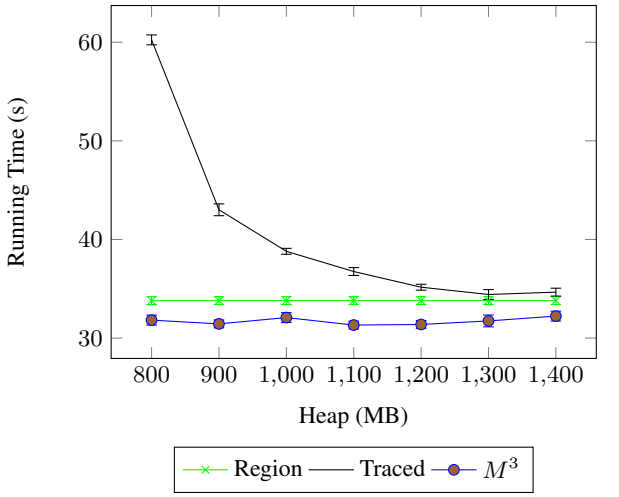

Figure 7. Performance of the various versions of MOSS

file read points to its corresponding passages array in the database. Secondly, for each match we find between two files, a pointer is created to each passage. No pointers from the reference counted heap to the traced heap are ever created, as while `passage` values exist in both, the existing code already copies such values from the reference counted database so that it can perform isolated mutation.

4.3.2 Evaluation

The results from an evaluation of MOSS under various configurations are presented in Figure 7. The evaluation is performed by running MOSS over two different versions of a large source code repository consisting of 1,255 files and 845,122 lines of code. Processing all of this causes MOSS to allocate 2,441MB of memory. For the versions of MOSS using a collector, we run the experiment over a range of heap sizes, from 800MB, the minimum that all versions can use, up to 1,400MB. For the M^3 version, we size the reference counted heap at a fixed 670MB, the maximum size it needs at any point, and the remaining is assigned to the tracing collector.

We see that at all heap sizes the M^3 version of MOSS achieves the strongest results with a running time of 31.82 seconds on average across all heap sizes. The region version of MOSS has a running time of 33.4 seconds and the traced version of MOSS has a running time between 34.6 - 60.24 seconds. The M^3 variant surprisingly achieves a faster time than the explicit, region version. This appears to be due to the cost of creating many small temporary regions compared to the performance of parallel mark-sweep.

The M^3 variant is unaffected by the heap size in the range shown as with the passage database removed from the tracing collectors heap, its collection policy triggers a collection less frequently and each collection is faster. With the collector set to trigger at 85% heap utilization, the traced version of MOSS with an 800MB heap is constantly at that limit since the passage database is 670MB in size, occupying 83.75% of the heap. With the M^3 version, the tracing collector runs without regard for the passage database since it resides in the reference counted heap, allowing it to make better policy decisions. This leads the traced version to run the garbage collector 86 times compared to the M^3 version where it runs only 17 times. While tweaking the policy of the tracing collector could help, this is exactly what we have achieved in the M^3 version! Also, even at a 900MB heap when the traced version runs with a comparable number of collections (15), the higher cost of collection for tracing the passage database means the traced version has a running time of 44.4 seconds.

4.4 Synthetic Middleware

Our third case study is a synthetic program that models the behaviour of a typical middleware service in a distributed web stack. This example is distilled from what is known about a persistent memory management performance issue in a commercial Internet service.

The basic design is a request-response architecture that further generates RPC calls to backend services. Our program has a number of parameters that can be tuned, all demonstrating various challenges for memory management systems:

1. **Allocation Per Request**: We perform a certain amount of allocation for every request received. Currently we use a tunable normal distribution for deciding the amount.
2. **RPC Delay**: We model the RPC calls by inserting delays in processing. In the production system, the RPC backend is a separate system.
3. **Long Lived Global State**: For each request that comes in we choose a random user id to assign the request to. For each user in the system we keep statistics at the service on the number and type of requests performed. The number of users in the system is tunable.

We believe these parameters accurately model the challenges of automated memory management in modern systems code and highlight the complex policy decisions that collectors need to deal with. The program causes problems for most generational collectors in the following way:

1. Firstly, the data allocated per request should ideally never be promoted to the old generation. It is by its nature short lived and bounded by the number of connections and requests that a server will handle at any time. However, due to the 99th percentile latencies present in a distributed system, it is extremely hard to size generations correctly to capture these properties.
2. Secondly, as the amount allocated per request can vary, this presents a problem when it is promoted to the old generation due to fragmentation and the use of free-lists in nearly all old generations.

3. Finally, these two properties over time cause the old generation to fill up with fragmented and ill-aged data, eventually triggering a worst-case stop the world compaction collection in many collector designs (e.g., HotSpot JVM's CMS collector).

The difficulty of getting the aging policy right through the knobs provided is key to this issue. The commercial Internet company, for example, runs a number of their services with 12GB young generations compared to 4GB old generations in an attempt to never have memory incorrectly promoted. While this helps, in the presence of huge variance in 99th percentile latencies and high throughput it only delays the inevitable.

Unfortunately due to limitations with our implementation we are unable to demonstrate the effects of varying the 99th percentile latency of RPC calls and the amount of data allocated per request. This is because the effect of these parameters, incorrect promotion of memory and fragmentation of the old generation, are only applicable in a generational system with two styles of collection and allocation. While the Boehm-Weiser collector has some generational behaviour in its incremental collection implementation, this provides neither a separate allocation mechanism nor tracing behaviour. Despite these limitations, we are able to show an improvement for our approach over pure tracing or reference counting.

4.4.1 Applying a Multi-Memory-Management System

While we cannot illustrate the specific problem outlined above because we do not have a generational tracing collector, the synthetic middleware is still an instructive benchmark. We apply our M^3 design in a straightforward manner: short-lived data associated with requests are managed by the tracing collector, as are connections themselves, while the longer lived global state is managed by the reference counted heap.

While one could object that a generational collector with a tracing, bump-pointer allocator for the young generation and a reference counting collector for the old generation would also suffice, we disagree. This would only hold if the aging policy could be tuned exactly right, a hard problem we believe when considering the variability of 99th percentile latency in these systems. Even if the aging and generations can be adjusted to prevent premature aging, it is likely that this configuration is non-optimal for the bulk of data that should be handled by the tracing collector that has a narrow distribution of life-times.

The key point here is the difficulty of tuning a program with a fixed collection policy. Rather than struggle to do that after the program has been developed with limited tools (e.g., setting the generation sizes), we are advocating allowing developers to use their knowledge of the application-specific characteristics to set the overall collection policy. That is, instead of a more complex collector to handle this workload, we show that our M^3 system can achieve high performance from our simple implementation with only a small amount of developer input.

4.4.2 Evaluation

As with Memcached, we evaluate our M^3 system for this case study on two metrics: throughput and latency. We use the same setup as before, two machines, one running the synthetic middleware service and the other the client for load generation and sampling. Both machines are 2.27GHz Intel Xeon L5640 connected via 10 gigabit Ethernet.

For throughput we evaluate two different configurations, one with no bytes allocated per request and another with a fixed 10 bytes allocated per request. These results can be seen in Figure 8. In both configurations the explicit memory management, full tracing and M^3 versions achieve the same level of performance at 1.86M req/s. The full reference counting version however only achieves 93% of the performance in the first configuration and 90% in the second configuration. It is worth highlighting the minimal impact on performance when we are allocating per-request for the tracing collector configurations. The work of the mark phase is proportional to the amount of live data and in this system that is bounded by the number of live requests we are dealing with. While the sweep phase is proportional to the allocation rate, the cost is far cheaper than marking due to the memory access patterns.

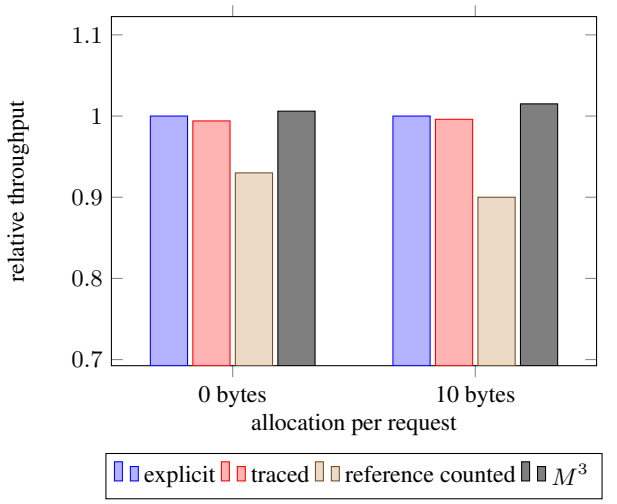

Figure 8. Throughput performance of various versions of our synthetic middleware service. Performance is relative to the explicit memory management version and consists of two different configuration. Firstly, with no allocation per requests and secondly, with a fixed 10 bytes allocated per request.

For evaluating the impact on latency we collected the 10 worst latency samples across a range of sizes for the number of users (or long lived data) in the system. We utilize three machines, one to run the service, one to generate load of 300,000 req/s across 2,000 connections and the final machine for sampling latency by generating 2,000 req/s. We run the experiment for 10 minutes, invoking the tracing garbage collector every 1 minute to evaluate pause times.

The result of the worst recorded latency measure is shown in Figure 9. As expected the latency of the traced version gets progressively worse as we increase the number of users in the system. The other three versions all have acceptable latency profiles, staying flat across the range of heap sizes. The explicit memory management version achieves a worst case of 2.5ms, the reference counted version a worst case of 3.1ms and the M^3 version a worst case of 4.5ms. While the M^3 version achieves the worst of the three, it is within acceptable bounds and its throughput performance is 10% greater than the reference counted version. Critically though, we have as developers chosen just one point in the possible space of policies for the M^3 version. A developer valuing latency more than throughput could easily reference count the entire heap.

Our results demonstrate that the M^3 version is able to achieve better latency than the traced version and better throughput than the reference counted version. Indeed, it is within a small constant latency overhead of the throughput and latency performance of the explicit memory management version.

4.5 Summary

In all three of our case studies, the M^3 system performs well, with a flat latency profile and throughput equal to the explicit memory management or traced version. We believe the key conclusions from these results are:

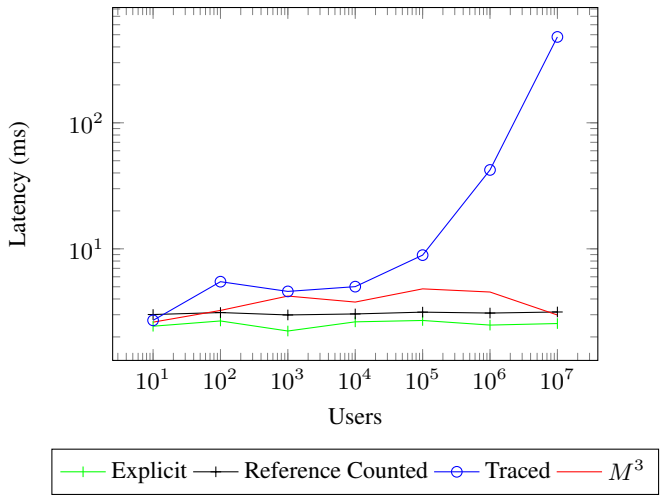

Figure 9. Latency of worst response-time as we increase the amount of users in the system. This corresponds to more permanent, long-lived data.

1. An M^3 system can achieve great performance. In each of our case studies, our M^3 version achieves as good throughput as the traced version and a flat latency profile very close to the behaviour of the reference counted version.
2. A little developer control goes a long way. We are able to achieve the competitive performance that we do with a very simple implementation that uses off-the-shelf components.
3. An M^3 system allows the policy to be expressed by the developer so that the collector can be customized to the program and done so after initial program development. While some may object that the right traditional collector design would achieve similar results to our M^3 design, designing a policy that is exactly right for all programs is at least very difficult; certainly such a policy has yet to be demonstrated.

5. Related Work

Customisable Memory Management Our closest related work is by Attardi, Flagella and Idlio on their Customisable Memory Management framework for C++ [5, 6]. Their work is motivated by performance and the observation that there is no single ideal collector. Like us, their design gives developers the ability to customize the memory management policy by exposing multiple heaps and allowing developers to choose, on a type-by-type bases, which heap is responsible for values of that type. They make use of C++ features such as inheritance and operator overloading to provide a framework that can be customised and extended for mixing various garbage collection strategies. Unlike our work however, they only consider tracing collectors, looking at abstracting a mark-sweep and semi-space collector for example. They also take the approach of tracing across heap boundaries. Tracing heap A will trace the reachable set of all pointers starting from the roots of A, regardless of if traced objects reside in a different heap. Their system also doesn't consider language level support, instead designing a conservative collector that can work with the existing C++ language.

Memory Management Toolkits There is a rich history of work in providing customizable memory allocators through either a richer interface than `malloc` or composable layers of simple allocation routines [8, 33]. They are similar to M^3 in their widening of the interface between the developer and memory manager. However, they only deal with allocation and unsafe interfaces while our focus is on safe memory management and choosing between reclamation strategies.

Similar work has been done for managed languages, providing toolkits of composable code that can be used to easily build garbage collectors [11, 24]. We believe this work is complementary to ours as it would allow rapid experimentation of different collector strategies. It also somewhat addresses the complexity of modern garbage collectors. Our work differs though in it focus and advocating of a wider interface between the developer and memory manager, something not explicitly considered by these toolkits.

Garbage Collection for C/C++ The Boehm-Weiser collector [14, 15] for C/C++ has been designed with the assumption that it does not control the entire heap of a program. The collector provides considerable flexibility, safely allowing calls to the libc `malloc` and `free` API's, as well as the ability to add individual pointers to the root set. The Boehm-Weiser collector essentially provides developers two choices of memory management policy: a value can be placed in the explicitly managed heap or in the GC-managed heap. There is no language-level support and so the developer must be very careful not to store the only pointer to a live object in the traced heap into the explicitly managed heap.

The work of Ellis and Detlefs [18] provides a design of a language extension to C++ for allowing garbage collection to be safely integrated with the language. They propose dealing with GC pointers stored in the explicitly managed heap in the same manner as us, by using write barriers and remembered sets. Broader in scope, it also includes the design of a safe subset of C++ that can support garbage collection efficiently. This proposal has apparently never been implemented or evaluated.

Hinted Collection Reames and Necula have recently published work [29] on *hinted collection*, where the developer can provide deallocation hints. In their system, a deallocation hint acts as a performance optimization and has no impact on the correctness of a program. The collector simply uses the hints to optimize the ordering and scope for tracing the heap, achieving a reduction in pause times. They have a single tracing collector managing the whole heap, rather than exposing different strategies with different trade-offs as we propose. The work is complementary to ours and could be used to further optimize the behaviour of the tracing collector in our system.

Pretenuring Pretenuring [10, 12, 17, 27] allocates objects directly into the old generation or permanent space in a generation collector. The performance benefits can be significant if the right objects are pretenured, as it avoids costs associated with promoting them from the young generation. The overall memory management policy is still fixed. Pretenuring is also done through profiler information, either online or pre-recorded, and doesn't attempt to widen the garbage collection interface for developers.

Garbage Collection Selection Work has been done on the automatic selection of a garbage collection algorithm for a particular program [19, 32]. Based on characteristics of the program (e.g., obtained by profiling) a specific GC algorithm is selected. In some cases, the choice can be changed dynamically at certain safe execution points. These works are motivated by the problem that no single GC provides the best results for all programs. While we believe M^3 provides a plausible solution to this issue, our motivation is in exploring the benefits of using several simple strategies to avoid the complexity associated with modern garbage collectors. In approach, this line of work exposes no control to the developer and does not allow different GC designs to be mixed within the same program.

Reference Counting and Tracing Collectors Previous work has explored the relationship between reference counting and tracing

collectors. The work of Bacon, Cheng and Rajan [7] showed that tracing and reference counting can be seen as duals of each other. Tracing finds live data and is batch driven in nature, while reference counting finds dead data and is incremental in nature. Various collector designs can be seen as combining elements of both to achieve low latency and high throughput.

The work of Blackburn and McKinley on a garbage collection design called 'Ulterior Reference Counting' [9] explores a generational collector that traces the young generation and reference counts the old generation. It is likely that some of our case studies would perform well with their collector. However, we have no doubt that there are other realistic systems where their approach would not perform especially well. Our main point is that the problem of efficient memory management can be simplified and more easily achieved if developers are given a modicum of control over setting the collection policy.

The state-of-the-art for pure reference counting collectors, such as the work of Shahriyar et al. [31] shows very promising results with a single collector being competitive across a range of benchmarks. This work though focuses on pushing the performance of reference counting collectors and as such greatly increases their complexity over naive reference counting, counter to our own aim. The work also doesn't look at latency, a principle concern in two of our case studies.

Safe Explicit Memory Management The prior work of Jim et al. with their safe dialect of C, the Cyclone programming language [23, 26], addresses the problem of safe memory management with strong developer control. Their work adopts a region-based type system to provide static safety for memory management. This type system provides a unifying framework for several forms of memory management, including stack allocation, arena regions, reference counting and tracing collection.

Recently, Mozilla has started working on a new programming language, Rust [28], that is similar in some ways to the memory management design of Cyclone. It uses a linear type system to provide a number of different management policies. These include unique pointers, reference counted pointers and traced pointers, allowing for stack allocation, reference counting and tracing.

These systems all differ from our work in their emphasis on a strong static typing discipline with the memory management strategy encoded into pointer types. An advantage of this approach is that it can express more sophisticated strategies with finer granularity of control. Our approach is very different as we are motivated by exploring the power of using off-the-shelf components as a response to increased complexity in runtimes and compilers. Part of our motivation is also to explore solutions for existing languages without such complex type systems as deployed by Cyclone and Rust.

6. Conclusion

Real-world garbage collectors in managed languages are becoming increasingly complex. We investigated whether this complexity is really necessary and show that by having a different (but wider) interface between the collector and the developer, we can get high performance with off-the-shelf components for real applications. Our interface, M^3, provides developers with the choice of multiple memory management strategies that can coexist, allowing them to select the best combination of policies for their program and change that choice at any time. We do not expect M^3 to be universally applicable, but believe that in the hands of experienced developers dealing with performance sensitive code it can be simpler and achieve stronger results than current approaches. To investigate the feasibility of M^3 we conducted case studies of three different programs: Memcached, MOSS and a synthetic middleware. For all three we achieved performance results equal to or better than any of the single memory management strategies we had available and were competitive with explicit memory management. Finally, we remark that while our design used a mark-sweep and reference counting collector, we do not believe this is necessary for an M^3 system. Instead, systems should be designed with trade-offs between memory managers carefully chosen and their integration well managed.

Acknowledgments

This work is funded by the DARPA Clean-Slate Design of Resilient, Adaptive, Secure Hosts (CRASH) program and by a gift from Google.

References

[1] GNU C Library, 2.18.1. `https://www.gnu.org/software/libc/`, 2013.

[2] Memcached. `http://memcached.org/`, 2014.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12. ACM, 2012.

[4] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith. A Comparative Evaluation of Parallel Garbage Collector Implementations. In *Workshop on Languages and Compilers for Parallel Computing*, LCPC'01, 2001.

[5] G. Attardi and T. Flagella. A Customizable Memory Management Framework. In *USENIX C++ Conference*, 1994.

[6] G. Attardi, T. Flagella, and P. Iglio. A Customisable Memory Management Framework for C++. In *Software Practice and Experience*, SPE'98, 1998.

[7] D. F. Bacon, P. Cheng, and V. Rajan. A Unified Theory of Garbage Collection. In *Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA'04. ACM SIGPLAN, 2004.

[8] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing High-Performance Memory Allocators. In *Conference on Programming Language Design and Implementation*, PLDI'01. ACM SIGPLAN, 2001.

[9] S. M. Blackburn and K. S. McKinley. Ulterior Reference Counting: Fast Garbage Collection without a Long Wait. In *Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA'03. ACM SIGPLAN, 2003.

[10] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for Java. In *Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA'01. ACM SIGPLAN, 2001.

[11] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with JMTk. In *International Conference on Software Engineering*, ICSE'04. IEEE, 2004.

[12] S. M. Blackburn, M. Hertz, K. McKinley, J. E. B. Moss, and T. Yang. Profile-Based Pretenuring. In *Transactions on Programming Languages and Systems*, TPLS'07. ACM SIGPLAN, 2007.

[13] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM*, 2008.

[14] H. Boehm. Space Efficient Conservative Garbage Collection. In *Conference on Programming Language Design and Implementation*, PLDI'93. ACM SIGPLAN, 1993.

[15] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. In *Software Practice and Experience*, SPE'88, 1988.

[16] H. Boehm, A. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Conference on Programming Language Design and Implementation*, PLDI'91. ACM SIGPLAN, 1991.

[17] P. Cheng, R. Harper, and P. Lee. Generational Stack Collection and Profile-Driven Pretenuring. In *Conference on Programming Language Design and Implementation*, PLDI'98. ACM SIGPLAN, 1998.

[18] J. R. Ellis and D. L. Detlefs. Safe, Efficient Garbage Collection for C++. In *C++ Technical Conference*, CTEC'94. USENIX, 1994.

[19] R. Fitzgerald and D. Tarditi. The Case for Profile-Directed Selection of Garbage Collectors. In *International Symposium on Memory Management*, ISMM'00. ACM SIGPLAN, 2000.

[20] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Conference on Programming Language Design and Implementation*, PLDI'98. ACM SIGPLAN, 1998.

[21] D. Gay and A. Aiken. Language Support for Regions. In *Conference on Programming Language Design and Implementation*, PLDI'01. ACM SIGPLAN, 2001.

[22] Google. The Go Programming Language. `http://golang.org/`, 2014.

[23] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based Memory Management in Cyclone. In *Conference on Programming Language Design and Implementation*, PLDI'02. ACM SIGPLAN, 2002.

[24] R. L. Hudson, J. E. Moss, A. Diwan, and C. F. Weight. A Language-Independent Garbage Collector Toolkit. Technical report, Amherst, MA, USA, 1991.

[25] International Organization for Standards. Programming Language C++. ISO/IEC 14882:2011(E), 2011.

[26] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, ATC'02. USENIX, 2002.

[27] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic Object Sampling for Pretenuring. In *International Symposium on Memory Management*, ISMM'04. ACM SIGPLAN, 2004.

[28] Mozilla. The Rust Programming Language. `http://www.rust-lang.org/`, 2014.

[29] P. Reames and G. Necula. Towards Hinted Collection. In *International Symposium on Memory Management*, ISMM'13. ACM SIGPLAN, 2013.

[30] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *International Conference on Management of Data*, COMAD'03. ACM SIGMOD, 2003.

[31] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley. Taking Off the Gloves with Reference Counting Immix. In *Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA'13. ACM SIGPLAN, 2013.

[32] S. Soman, C. Krintz, and D. F. Bacon. Dynamic Selection of Application-Specific Garbage Collectors. In *International Symposium on Memory Management*, ISMM'04. ACM SIGPLAN, 2004.

[33] K.-P. Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice and Experience*, v26:1–18, 1996.