Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Modern garbage collector for HashLink and its formal verification

Author:
Aurel Bílý

Supervisor:
Prof. Susan Eisenbach

Second Marker:
Prof. Sophia Drossopoulou

June 15, 2020

Abstract

Haxe [1] is a high-level programming language that compiles code into a number of targets, including other programming languages and bytecode formats. HashLink [2] is one of these targets: a virtual machine dedicated to Haxe. There is no manual memory management in Haxe, and all of its targets, including HashLink, must have a garbage collector (GC). HashLink's current GC implementation, a variant of mark-and-sweep, is slow, as shown in GC-bound benchmarks.

Immix [3] is a GC algorithm originally developed for Jikes RVM [4]. It is a good alternative to mark-and-sweep because it requires minimal changes to the interface, it has a very fast allocation algorithm, and its collection is optimised for CPU memory caching. We implement an Immix-based GC for HashLink.

We demonstrate significant performance improvements of HashLink with the new GC in a variety of benchmarks. We also formalise our GC algorithm with an abstract model defined in Coq [5], and then prove that our implementation correctly implements this model using VST-Floyd [6] and CompCert [7]. We make the connection between the abstract model and VST-Floyd assertions in a novel way, suitable to complex programs with a large state.

Acknowledgements

Firstly, I would like to thank Lily for her never-ending support and patience.

I would like to thank my supervisor, Susan, for the great advice and intellectually stimulating discussions over the years, as well as for allowing me to pursue a project close to my heart.

I am also grateful to the welcoming Haxe developer team, without whom this project would not be possible.

Last but not least, thank you, my friends, for the great 4 years.

Contents

> “Ph’nglui mglw’nafh Cthulhu R’lyeh wgah’nagl fhtagn.”

H. P. Lovecraft, The Call of Cthulhu, 1928

Chapter 1

Introduction

1.1 Haxe

Haxe [1] is an open-source programming language created by Nicolas Cannasse in 2006. It is a general-purpose, high-level language that is rapidly evolving thanks to an active team of developers. Its primary selling point is its ability to transpile[1] into many different targets from the same codebase. At the time of writing, Haxe supports the following targets:

- **Programming languages**: ActionScript 3.0, C++, C#, Java, JavaScript, Lua, PHP, Python
- **Bytecode**: Flash SWF, JVM
- **Dedicated virtual machine bytecode**: Neko, HashLink

Haxe is popular among video game developers due to its portability, the availability of high-quality game engines, and its good performance. Performance is critically important for any interactive experience. Currently, the most performant Haxe target is C++, allowing the generated code to be optimised by mainstream C++ compilers, such as GCC [8] or Clang [9]. However, C++ is among Haxe's older targets and it suffers from slower compilation times.

1.2 Garbage collection

As is the case for the majority of modern high-level programming languages, Haxe assumes a garbage-collected environment. Programmers cannot, in general, affect how memory is allocated or managed during the execution of their programs.

[1] "Transpilation", at least in the Haxe community, generally refers to compilation where the output is another high-level programming language, rather than bytecode or machine code. As such, given the Haxe targets, Haxe is both a transpiler and a compiler.

Garbage-collected environments are arguably less error-prone and easier to understand, but in the case of Haxe this is a necessity—most of Haxe targets are garbage-collected and do not have any manual memory management interface.

For the C++ and HashLink targets, the garbage collector is part of the runtime that is bundled together with the compiled program. C++ does not have a garbage collector (by design[2]), so this functionality is provided by the "hxcpp" library.

1.3 HashLink

The HashLink [2] target was introduced in 2015 as a high-performance target with low compilation times. It is a virtual machine created specifically to run bytecode produced by the Haxe compiler. As a result, it requires a garbage collector to manage the dynamic memory allocated at runtime. The current implementation is a very simple, stop-the-world, "mark-and-not-sweep" collector that is rather slow when compared to other Haxe targets in GC-bound benchmarks[3].

1.4 JIT, HL/C, and libhl

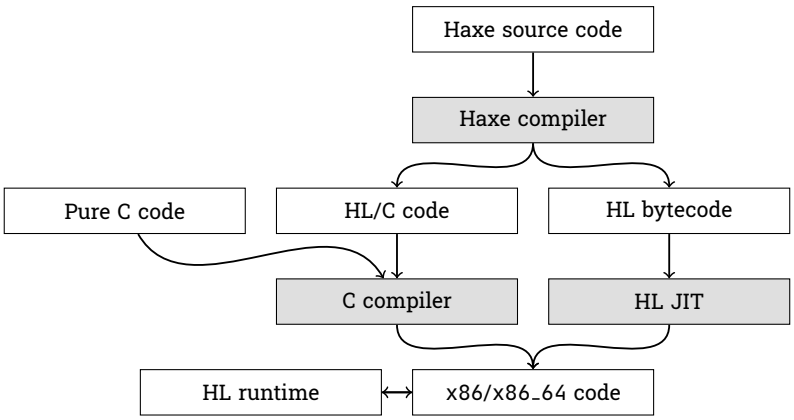

Figure 1.1: Haxe, HashLink, and HL/C compilation phases. Shaded blocks indicate processes, unshaded blocks represent artifacts.

HashLink itself also consists of two "sub-targets": the bytecode-executing virtual machine ("JIT") and a library written in C that allows code compiled by Haxe to be further compiled by regular C compilers while still using most of the virtual

[2] Although there are proposals such as N2310 or N2670 that would enable easier integration of automatic garbage collectors. However, these proposals are not implemented on major compilers at the time of writing. See `http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2310.pdf` and `http://open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2670.htm`.

[3] As can be seen from the benchmark results at `https://benchs.haxe.org/`.

machine runtime ("HL/C"). The virtual machine currently targets Intel `x86` and `x86_64` architectures [10], where the bytecode is compiled into machine code just before execution starts.

The garbage collector is part of the runtime that is dynamically or statically loaded to HashLink applications. In the JIT case, the runtime is loaded by the VM executable. In the HL/C case, the runtime is linked during compilation so that its functions can be called by the C code.

Finally, HashLink can be used as a standalone library called `libhl` used only with (or in addition to) code written directly in C, without a Haxe transpilation step. This is currently not a priority application for HashLink, but the API is designed with this in mind. See Figure 1.1 for an overview of the compilation phases of Haxe and HashLink.

1.5 Project goals

The primary goal for this project was to significantly improve or even replace the garbage collector for the HashLink VM. With an improved GC, HashLink can become much more convenient as a target for developers, as it can reach a greater runtime performance in addition to its already fast compilation times and native integration. To be able to empirically determine whether the project was successful in terms of performance, developing a new benchmark suite was also a subgoal.

Additionally, if HashLink proves to be fast, reliable, and easily maintainable, it may in the future be a way to "bootstrap" Haxe—making the compiler compile itself via the HashLink target while maintaining its current speed.

The secondary goal for this project was to formally verify the GC implementation, by modeling its behaviour in higher-order logic using the Coq proof assistant [5], then connecting this model to the C implementation using CompCert [7] and VST-Floyd [6].

1.6 Outline

In Chapter 2, we briefly describe some existing approaches to garbage collection, as well as their advantages and disadvantages. We also describe some of the literature related to formal, mechanised verification of garbage collectors and C code. In Chapter 3, we introduce the specifics of a GC implementation for HashLink, including a number of low-level details and various optimisations. In Chapter 4, we provide empirical data showing the performance of the Immix-based collector compared to HashLink's original collector, as well as comparisons of the Immix-based HashLink target and the other Haxe targets. In Chapter 5, we describe an abstract model of our Immix-based collector. In Chapter 6, we discuss in detail the approach taken for the formal verification of the collector, using the abstract

model. Finally, in Chapter 7, we conclude by suggesting possible future goals of the new Immix-based implementation and its extensions.

Chapter 2

Background

2.1 Introduction

In this chapter, we describe some of the existing literature related to the project. In Section 2.2, we explore the basic ideas of garbage collection, how to evaluate GC performance, and list some GC algorithms. In Section 2.3, some considerations specific to garbage collection in Haxe and HashLink are discussed. In Section 2.4, we discuss some approaches to evaluating GC performance in practice with benchmarks. In Section 2.5, we note a number of methods used to verify implementations of garbage collectors and other complex projects. Finally, in Section 2.6, we summarise the relevant findings and emphasise the project challenges.

2.2 Garbage collection

It is increasingly common to use programming languages with garbage collection facilities. The popular representatives of the two paradigms are Java, which is garbage-collected, and C/C++, which are manually managed. Of these, Java is gaining in relative popularity due to lower development costs and higher general productivity [11].

The benefits are easy to describe: the developer generally only needs to know when a particular object needs to be allocated, but keeping track of its lifetime throughout the application, and knowing when it is no longer needed by any other part of the application is much more difficult. Garbage-collected environments expose an API which only provides object allocation. Deallocation of objects happens automatically, without the developer's explicit requests. In many cases, even though the garbage collector knows when an object is being deallocated, the exact moment when this happens depends largely on memory usage and allocation patterns, and as a result, it is usually not even possible to trigger additional functionality on deallocation. This is the case in both Java and Haxe, although HashLink as a library allows such finalisation functionality.

Ideally, a garbage collector would deallocate objects soon after they are used for the last time, in order to decrease memory footprint. However, this property, "liveness", is hard to determine and depends on the specifics of the program[1]. Instead, GCs use one of two approximations of liveness, forming the two major GC classes:

- **Tracing**: objects are assumed to be live if they are reachable by following pointers from one of the GC "roots".
- **Reference counting**: an incoming reference counter is maintained on every object; an object is assumed to be live if this counter is non-zero.

Although both classes have their advantages and disadvantages, and although reference counting has recently become comparable in performance to tracing collection due to Shahriyar [13], in this project we will focus on tracing garbage collection. Switching to a reference-counted collector would be a large change from the current approach, and would require changes to the GC interface. See Section 2.3.3 for further discussion.

2.2.1 Performance metrics

The operation of a GC has an impact on the performance of its host application. Different GC algorithms affect the application in different ways, so the choice of GC depends heavily on the type of application. The primary metrics which concern us in this project are:

- **Throughput**: the speed at which the application can allocate new objects.
- **Pause time**: the time it takes for the GC to complete a collection cycle, during which the application is paused.
- **Memory overhead**: the amount of memory dedicated to the GC internals and space used for GC bookkeeping.

Other metrics which are less relevant to HashLink's applications:

- **Promptness**: the time it takes for an object to be collected after it becomes unreachable. There are no finalisers in Haxe, so this is not of crucial importance.
- **Completeness**: whether the collector collects all unreachable objects. GC schemes aim to be complete in general, otherwise running out of memory would quickly become an issue. However, some schemes are "conservative" rather than "precise" and may result in false-positive identification of pointers, for example when integer values happen to fall into the address range of

[1] In general, determining liveness of objects in a program would amount to solving the halting problem. [12]

a GC. Such memory is then not collected, even though it is not used by the application. A low rate of false positives is desirable, but a non-zero amount is probably acceptable. Additionally, completeness may be impossible to reach in HL/C due to the uncooperative nature of the C language (see Section 3.5.1).

2.2.2 Tracing garbage collection

The most commonly used category of GC in production environments is tracing GC. The algorithms in this category include mark-and-sweep, mark-and-compact, and semispace copying, briefly described in the following subsections.

Mark-and-sweep

With mark-and-sweep algorithms, first introduced by McCarthy [14], a collection cycle consists of tracing all objects reachable from the root set. Objects reached in this "marking phase" are marked, usually by setting a designated bit in the object header or a separate bitmap containing the mark bits for a larger number of objects. The next phase is the "sweeping phase", which iterates through all objects on the heap and deallocates them if their mark bit is not set.

A slightly different, slower alternative is the "mark-and-not-sweep"[2] algorithm, currently used in HashLink. In this variation, there is no sweep phase. Instead, objects on the heap are iterated during allocation and space used by unmarked objects may be used. One possible advantage of this method is that it is simpler to implement.

Mark-and-compact

One problem introduced by the simple sweeping method of mark-and-sweep GC is fragmentation—after some time, the heap contains old objects of various sizes spread far apart in addition to new objects allocated in the gaps. This decreases object locality, i.e. objects which are close to one another in terms of pointer distance may end up far apart in memory, which in turn decreases the usefulness of a CPU cache.

Mark-and-compact algorithms [15, 16] address this by compacting marked objects into a smaller part of the heap, then updating pointers to compacted objects in a separate phase. The order of compacted objects varies from algorithm to algorithm—some aim to increase object locality, some maintain the original allocation order.

[2] Also known as "lazy sweeping" in the literature.

Semispace copying

Mark-and-compact algorithms result in lower fragmentation and fast allocation but have high collection costs. Semispace algorithms, first described by Fenichel and Yochelson [17], achieve good performance all around in return for a much higher memory usage. The heap is split into two memory regions and live objects are copied from one space to the other during the collection phase. This enables the GC to compact and reorder live objects arbitrarily, without requiring a separate pass to update pointers. Because it is theoretically possible that a collection cycle collects no objects, and because the two semispaces trade places each collection cycle, they are identically sized by necessity, and so a semispace collector requires at least double the amount of memory of a non-semispace collector.

2.2.3 Generational GC

The "weak generational hypothesis" states that most objects "die young" (i.e. become unreachable soon after allocation). This appears to be generally true regardless of the application or language used. Studies on the Java DaCapo benchmark suite have shown that between 65% and 96% of objects survive less than 64 kilobytes of GC activity [18].

As a result, it is beneficial for a GC to focus its attention on young objects, and try to collect young objects much more often than older objects. This is the core idea of generational GC algorithms (first due to Lieberman and Hewitt [19]), which separate the heap into at least 2 generations. Collecting the young generation, also called the nursery, constitutes a "minor" collection. Tuning the sizes of the individual heap generations is important for achieving optimal performance. Object age itself is typically measured by counting the number of collection cycles a particular object survives, since measuring actual GC activity in terms of bytes allocated is problematic for multi-threaded collectors.

2.2.4 Concurrent GC

The algorithms described so far do not address the issue of threaded applications and concurrency in the GC itself. A single collector thread is easier to reason about, but a GC will reach much better performance on a multicore system if the collection of dead objects is done by multiple threads. Some GC implementations, as is the case for HashLink, are "stop-the-world", i.e. all the application threads must be paused before collection actually takes place. This is the primary cause of pauses and is problematic for interactive applications.

When implementing concurrent GC algorithms, data races must be considered. Certain problems can be solved with atomic read/write operations as implemented on modern multicore architectures. Additionally, some objects might impose write barriers—synchronisation points used when writing into certain objects. An ex-

ample of a high performance collector using these techniques is the Shenandoah GC [20].

A GC that is not concurrent, but uses multiple threads to decrease the application pause during collection cycles is called a "parallel" collector.

2.2.5 Modern architectures

In addition to supporting concurrent execution, modern processor architectures have a large number of features that can drastically improve execution time when used properly. These features include cache lines, cache invalidation, and memory prefetching. All aim to decrease the cost of accessing the main system memory, since such operations tend to be orders of magnitude slower than operations contained within a core and its own cached memory.

As a result, some of the more recent improvements in garbage collectors come not from more complex algorithms but rather a more clever memory layout that can better cooperate with the CPU.

2.2.6 Immix

The Immix collector [3] is a parallel collector based on mark-and-sweep with a modified thread-local allocation algorithm and a heap layout more optimised for modern CPU caches. The heap is organised into large blocks containing lines which then contain actual objects. The line size is chosen to more or less align with the cache line sizes of modern CPU architectures, such as `x86_64`. Objects are not collected unless all objects in a line are unreachable, and surprisingly this coarser granularity leads to performance improvements.

Immix is described in further detail in Chapter 3, since our implementation is based on it.

Modified Immix collectors

The latest GC used in Haxe's C++ target is a "generational Immix" collector with many other differences from the original design. The generational part means that the heap is additionally split into generations based on object age—a younger nursery which uses fast bump allocation and an older generation that is an Immix space.

Shahriyar et al. [21] introduce multiple modifications to Immix. These include "Sticky Immix", a generational modification; and "RC Immix", a hybrid reference-counting Immix collector. The latter is available in Jikes RVM [4], an open-source Java VM implementation.

2.3 Haxe, HashLink, HL/C

2.3.1 GC for immutable data

Haxe is primarily an object-oriented language whose semantics should seem familiar to programmers of Java or C++. However, partly due to its roots in the functional programming language OCaml, the type system of Haxe has certain kinds of types which are immutable after instantiation. These are called algebraic data types (ADTs), or simply "enums" in Haxe terminology.

Immutable data are a familiar concept in functional languages such as Haskell or OCaml. A well-designed GC system can take advantage of immutability—in the case of a generational collector, older objects can never point to younger objects.

If the overhead of implementing a separate collection strategy for immutable data is not too high (both in terms of implementation time and performance costs), it is also important to make sure the interface between the two data categories is seamless. Ueno et al. [22] demonstrate a similar endeavour, aiming for memory interoperability between C and ML (a functional language). Their key idea is to avoid moving or compacting allocated memory, thereby better facilitating immutable data in the scheme.

2.3.2 Dynamic types

Although Haxe is a statically typed language, it has some features to allow easier integration when compiling to dynamically typed languages. The primary example of this is the `Dynamic` type, which is similar to values in JavaScript: it may be `null`, any primitive value (Boolean, integer, etc), or an object with fields created dynamically at runtime.

Using `Dynamic` is usually not a good solution in idiomatic Haxe code[3], but it is nonetheless a part of the type system that cannot simply be turned off. As such, HashLink, being a Haxe target, must support `Dynamic` at runtime.

For the collector, this means that certain objects will not have static type information, which is particularly relevant when tracing any referenced objects. Modifying the type information at runtime could cause difficulties to fully concurrent collectors. Since fields can be added at runtime, `Dynamic` instances can grow in size from the time they were initially allocated.

2.3.3 GC interface

As mentioned in the introduction, HashLink can be used as a standalone C library. The library should be able to provide a garbage-collected environment with minimum effort on the part of the programmers. To this end, a tracing GC seems to be a

[3]The Haxe manual recommends minimising the use of `Dynamic`. See `https://haxe.org/manual/types-dynamic.html`.

good choice: objects are allocated with a call to a particular function, and from that point on, any references to that object should be automatically detected by the GC, either by examining pointers of referent objects or by examining thread stacks.

This simple interface would need to be modified for a reference-counting GC implementation as well as a concurrent GC such as Shenandoah [20]. The operation of setting an object field to point to another object can no longer happen without the knowledge of the GC. The reference count has to be updated and/or write barriers need to be checked.

2.3.4 HashLink JIT

When using HashLink as a virtual machine for Haxe-produced bytecode, HashLink uses "just-in-time" compilation, where the bytecode is converted to x86 or x86_64 machine code and loaded directly as executable memory. This process is fast and avoids the low performance and memory behaviour of a purely interpreter-based VM. However, it is more difficult to debug. Hence the focus of GC work should be on HL/C. After confirming that a GC works well and is apparently bug-free, it can be integrated into HL JIT later.

2.4 Benchmarks

It is also important to consider how to evaluate performance metrics in practice. GC research often focuses on Java runtimes. Due to its popularity, there are a number of benchmark suites designed to emulate large, realistic Java applications using memory in various allocation patterns. These include the DaCapo suite [23] or the SpecJVM suites [24].

Unfortunately, such complete benchmarks are not available for Haxe. At the start of this project, there was a small number of GC-bound microbenchmarks in addition to various unit tests and problematic cases demonstrated in GitHub issues by users of the HashLink target. Since "toy benchmarks", or microbenchmarks are widely considered inadequate at reporting useful information about the performance of a GC [25, 26], these are of limited usefulness but still clearly demonstrate that the C++ and JavaScript targets have better collection schemes, the latter due to the heavily optimised V8 runtime.

To avoid the pitfalls of microbenchmarks, a good benchmark should be large enough for the GC to reach a steady state, that is a state after the VM is "warmed up", and ideally after the heap has gone through at least one major collection. A steady state is more indicative of the long-term behaviour of a collector.

A good benchmark should emulate allocation patterns of a realistic application. This involves examining a template application's behaviour in detail, recording all the allocation requests, their sizes and times, the object relationships in terms of pointers, as well as any modifications of object fields that could result in objects

being unreachable. This type of activity logging is not yet available for HashLink. In implementing such a logger, it may be useful to refer to Elephant Tracks [27], a tool with the same goal implemented for Java.

Fortunately, there are a number of fully developed video games created with HashLink, which can serve as very realistic benchmarks with the needed allocation and usage patterns.

2.4.1 GC tuning

GC "tuning" refers to adjusting the parameters of a collector to improve its performance for a particular application. Some parameters for the collectors described so far include: number of generations, sizes of memory regions dedicated to particular types of objects, total heap size, collection schedule, and heuristic thresholds.

Tuning is an application-specific process and parameters that achieve very good performance with one application may produce terrible results with another. Even after tuning, results depend on a variety of external factors, such as the machine workload, which is especially relevant in server applications. Nevertheless, a well-tuned collector can be a good demonstration of its own strengths. The process of tuning is not trivial, and it may be useful to consider machine-assisted tuning techniques, such as the ones presented by Lengauer and Mössenböck [28], who demonstrate an experimental result of a 77% improvement of GC runtime for some applications after tuning. It is important to note that most GC tuning is still a manual process.

A default GC collection schedule is one in which a major GC collection happens only when the heap is exhausted. By contrast, Jacek et al. [29] provide an alternative route, trying to find a best case scenario for a particular collector running a particular application. This is an offline method that attempts to find the optimal collection schedule from a trace of the collector behaviour, where the trace contains timestamps of GC-related events such as allocations. Program analysis is not a part of the method, so it could be applicable to any language if the required traces could be generated.

2.5 Verification

Garbage collectors form a crucial component in many systems, and due to their heavy focus on good performance and necessary interactions with low-level memory representation, their code is generally not trivial to understand and debug. As such, even for applications which are not critical, it is a worthy endeavour to formally verify the functionality of a GC as compiled, with respect to a higher-order logic model of that collector.

By formally verifying software, we aim to have a specification in a simpler, but formal language, and ensure that the implementation satisfies this specification.

Such a formal language may be higher-order logic (HOL), as implemented in the proof assistants Coq [5] or Isabelle [30].

One possible method for the verification is to use Coq to describe the abstract GC model in combination with CompCert [7], the verified C compiler, and VST-Floyd [6], a tool for proving C programs using the Verifiable C separation logic. This method was employed for the (almost) complete end-to-end verification of a TCP/IP echo server by Koh et al. [31]

Hawblitzel and Petrank [32] show an alternate approach using Boogie [33] and Z3 [34], proving the correctness of a mark-and-sweep collector and a semispace copying collector. Ericsson et al. [35] prove an incremental collector for CakeML using the HOL4 theorem prover.

2.6 Summary

In the sections above, a number of possible approaches to the various goals of this project have been outlined. In Chapters 3 to 6, we elaborate on the approaches actually taken.

The GC method forming the basis of the implementation is Immix [3], a variant of mark-and-sweep. This method was chosen due to its similarity to mark-and-sweep in terms of the interface exposed to the implementing system (i.e. HashLink), and because of its great performance characteristics, as demonstrated by its various implementations in Java virtual machines.

Formal verification of the collector is modelled after Koh et al. [31], even though this publication verifies a networking server rather than a GC. This approach was deemed more appropriate to avoid modelling large parts of HashLink itself.

Some sources of difficulty have also been outlined in the sections above. To summarise:

- **Lack of realistic GC benchmarks for Haxe.**
- **Lack of previous work**: the current HashLink GC is rudimentary and most of the work will need to be done from scratch.
- **High performance requirements**: many possible applications of HashLink would be bound by its slow GC (for example, using it to bootstrap the Haxe compiler), so the new GC should be heavily optimised.
- **Complex type system**: `Dynamic` instances need to be handled carefully. Immutable ADTs may be handled separately as an optional optimisation.
- **Formal verification**: formal verification of large codebases is far from trivial and often individual functions require hundreds of lines of Coq proofs.

Chapter 3

Immix-based garbage collector for HashLink

3.1 Introduction

In this chapter, we provide a high-level description of Immix in Section 3.2. We then describe our implementation of a garbage collector for HashLink based on Immix in Section 3.3. In Section 3.4, we discuss the low-level details of some optimisations. Finally, in Section 3.5, we list the key differences between the original Immix publication and the collector implemented for HashLink.

3.2 Immix

Immix [3] is a "mark-region" collector originally designed and benchmarked on the Java research virtual machine, Jikes RVM [4]. It divides the memory into fixed-size regions called blocks. Blocks are either free and owned by the collector itself, or in use and owned by a particular thread. Threads allocate objects inside blocks using fast bump allocation[1] until they run out of space, at which point they recycle one of their previously used blocks, or take a new one from the collector.

Giving blocks to threads addresses one possible source of slowdown during allocation—namely, the cost of synchronisation of multiple threads. With thread-local blocks, allocation in most cases does not require synchronisation, only in the worst-case scenario when a new block is required, in which case the global free block pool must be locked. Note that block ownership only implies that a thread has the exclusive right to allocate data in the owned block. Data may still be arbitrarily shared across threads, so it is possible that threads modify the object

[1] Also known as linear allocation, it consists of simply keeping track of the last allocation position (the cursor) and the end of the current span of free space (the limit). Objects that fit within this space are allocated at the cursor, which is then moved ("bumped") by the size of the object.

data contained in another thread's blocks. To avoid data races in the application code, additional synchronisation primitives of the runtime must be used.

Immix additionally splits blocks into lines, the size of which is designed to align with the size of CPU cache lines. During the marking phase, the collector marks objects as any mark-and-sweep collector, but it also marks the lines to which the objects belong. The majority of objects are smaller than one line, so the line marking is coarser-grained than object marking. The purpose of line marks is to help the collector quickly identify which parts of a block are free. Rather than walking through the block and looking for gaps of a particular size, the collector can instead look for any line-sized gap according to a bytemap[2] containing the line marks.

If a medium object[3] cannot be allocated immediately, a fresh block is used to avoid a potentially expensive gap search. See Section 3.3.9 for the full allocation algorithm.

With the allocation scheme described above, objects are allowed to span lines, but not blocks. Because objects can span lines, lines immediately after a marked line are considered implicitly occupied. As is common in many collectors [36, 37, 38], large objects are treated differently and are allocated into a global free list. For Immix, this threshold is 25% of the block size, i.e. 8 kilobytes, a value which was determined empirically.

Because objects are only allocated in line-sized gaps or larger, there is a possible memory overhead due to live objects in a line with dead objects. The benefits are that this scheme decreases fragmentation and collection time.

To avoid lines with very few live objects, Immix can also opportunistically compact the heap. Compaction may happen during the marking phase and is triggered by heuristics based on fragmentation statistics. The most fragmented blocks are good candidates for compaction, although it is possible the process is terminated due to insufficient space, since the statistics only provide estimates of fragmentation based on line usage, not real space usage.

Differences between the original Immix publication described above and the collector implemented for HashLink are discussed in Section 3.5.

3.3 Implementation details

In this section we describe the collector implementation for HashLink. Figure 3.1 shows an overview of memory areas, block types, and transition paths between block types.

Throughout this section and the implementation itself, we assume a 64-bit architecture, such as `x86_64`. A "word" is understood to be 8 bytes in size, large

[2] A bytemap is chosen rather than a bitmap to allow parallel, multi-threaded marking without race conditions.

[3] A "medium object" in Immix is any object larger that one line.

enough to fit a pointer. A 32-bit implementation would result in several changes to the specific memory layouts, but the overall structure of the collector remains the same.

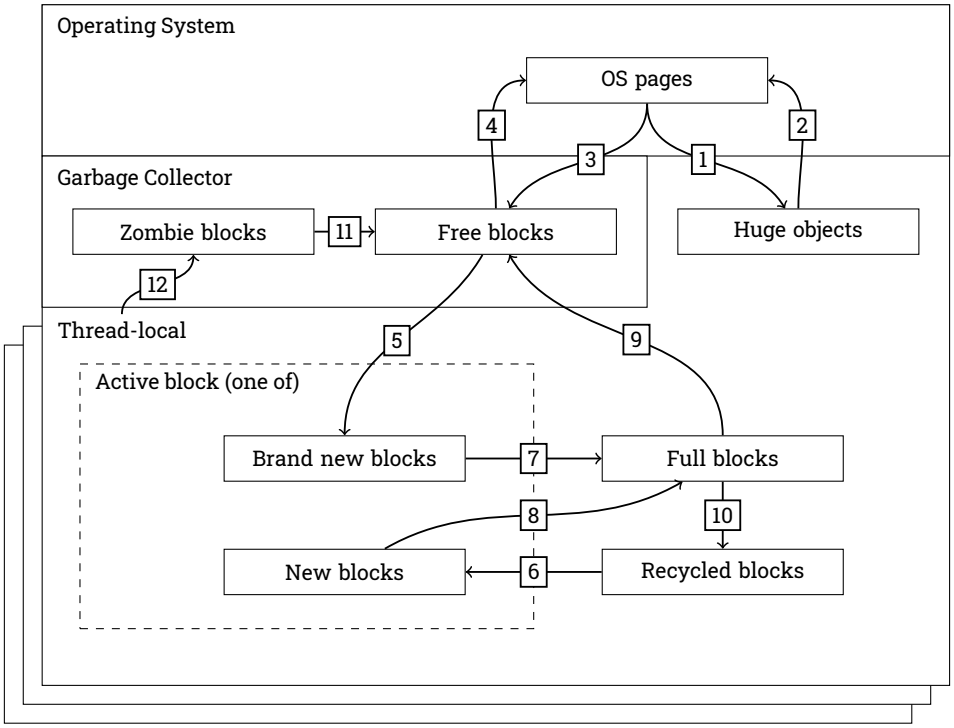

Figure 3.1: Memory organisation. The numbered transition paths are explained in Section 3.3.6.

3.3.1 Memory organisation and sizes

The size of lines was chosen to correspond to the size of CPU cache lines on major CPU architectures [10]. Block and page sizes were derived to obtain a good fit of block headers and object metadata.

- **Line size**: 128 bytes (corresponds to 16 words)
- **Block size**: 64 KiB = 65,536 bytes (accommodates 442 lines and metadata)
- **Normal page size**: 4 MiB = 4,194,304 bytes (accommodates 64 blocks)

3.3.2 Global OS page management

The collector directly interacts with the operating system in the methods `gc_alloc_os_memory` and `gc_free_os_memory`. All allocated memory regions must

be page-aligned. This is usually true of pointers returned by the underlying function `mmap`[4], although sometimes a realignment procedure is necessary. The realignment procedure is not modified from the current HashLink implementation, so we skip its explanation here.

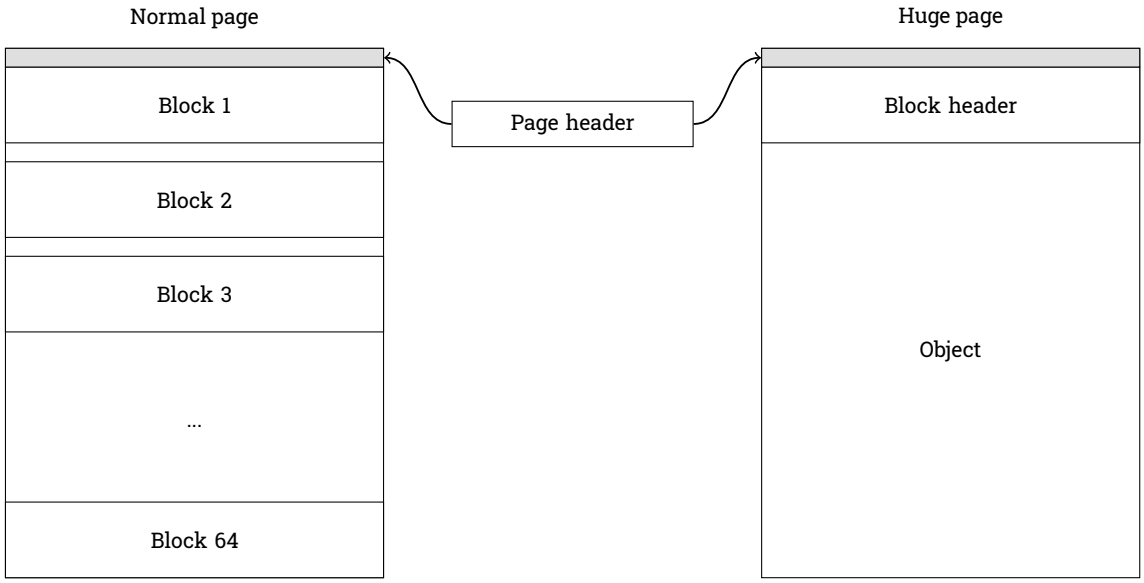

Figure 3.2: Page layout.

3.3.3 GC page management

"Raw" pages allocated from the OS are initialised by the GC in one of two ways. Normal pages are always 4 MiB in size and consist of blocks. Huge pages have variable sizes, which may exceed 4 MiB, and accommodate a single large object, prefixed by a mostly empty block header so that huge objects can be treated just like other objects.

Both page kinds share a common header structure. This header is embedded inside the block header of the first block of the page. See Figure 3.2 for a visualisation. Due to the alignment requirements for allocated pages, the page header corresponding to a particular object or block can be obtained by discarding a number of the least-significant bits of the address (21 in the implementation).

The GC keeps track of all allocated pages in two doubly-linked lists, one for each page kind. These lists are used when scanning pages during the sweep phase. All allocated pages also participate in a dynamically growing hash map, which allows the GC to quickly check if a particular pointer belongs to the collector-managed memory or not.

[4] `mmap` can be used to allocate raw pages of memory from the operating system. See `https://linux.die.net/man/3/mmap` for complete documentation.

Pages are obtained using the methods `gc_alloc_page_normal` and `gc_alloc_page_huge`. Pages are deallocated using `gc_free_page_memory`.

3.3.4 Free block pool

The free block pool is a singly-linked list of blocks, common to all threads. No live objects should be present in the blocks in the pool. Whenever a thread requires a new block, it must first obtain the lock which protects the pool. The head of the free block pool list is assigned to the thread, reducing the pool size by one, then the lock is released. If the pool is empty when a block is requested, a GC cycle is triggered. If the pool remains empty after the cycle, a new normal page is allocated, which adds 64 blocks to the free block pool.

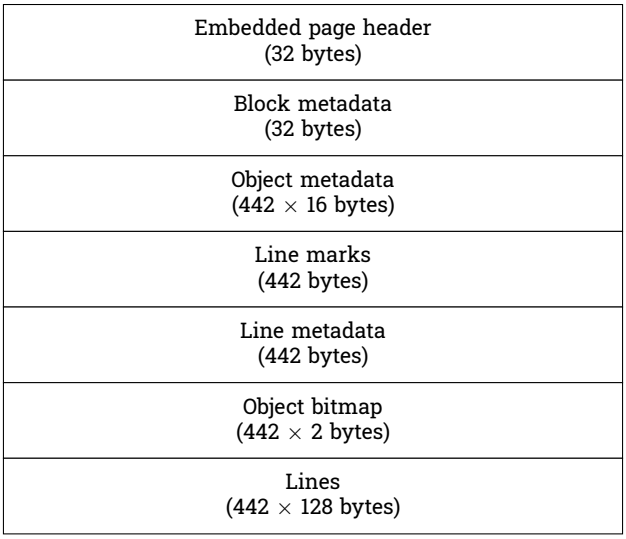

Figure 3.3: Block layout. A full block spans 64 KiB. Padding is not shown.

3.3.5 Blocks

Blocks consist of 442 lines containing allocated data, and a header that contains:

- the block kind;
- pointers to the previous and the next block in a list; and
- metadata for all objects of the block, including mark bits, object sizes, and object kinds.

Depending on the block kind, the previous/next pointers have different meanings, as shown in Table 3.1.

Kind	List kind	List meaning
Free	Singly-linked	Global free block pool
New	None	None
Full	Doubly-linked	Thread-local list of full blocks
Recycled	Doubly-linked	Thread-local list of recycled blocks
Zombie	Doubly-linked	Global list of zombie blocks
Brand new	None	None

Table 3.1: List of block kinds and their list participation.

3.3.6 Block lifetime

Blocks change their kind as the garbage collector operates. Here is the list of all block kind transitions. The numbers refer to the paths shown in Figure 3.1.

- **1,2**: Huge objects are allocated and released separately from other blocks, being accommodated in OS pages of their own.
- **3**: Blocks are initially formed as **Brand new** blocks from a normal page taken from the OS.
- **4**: Free blocks may be returned to the OS if the heap is underutilised.
- **5,6**: A **Brand new** block may be taken by a thread and used as its active allocation block. If there are any blocks available in a thread's recycled block list, they are used instead (re-marked as **New**).
- **7,8**: When the allocation cursor reaches the end of a block, the block is marked as **Full** and added to the thread's list of full blocks.
- **9,10**: During the sweep phase, the GC scans all full blocks. Full blocks which have at least one free line after sweeping are marked as **Recycled** and are moved into the thread's list of recycled blocks. Full blocks which are completely free are instead put back into the free block pool.
- **11,12**: When a thread stops executing, all of its blocks are marked **Zombie** and given over to the GC. These may eventually become empty and be added to the global free block pool.

3.3.7 Objects

Objects are separated into two parts, stored in different sections of a block: the metadata and the object data. The application can only directly access object data. Object data is word-aligned and in most cases consists of a type pointer followed by field data.

The metadata overhead is 1 or 2 bytes per object. In Figure 3.4:

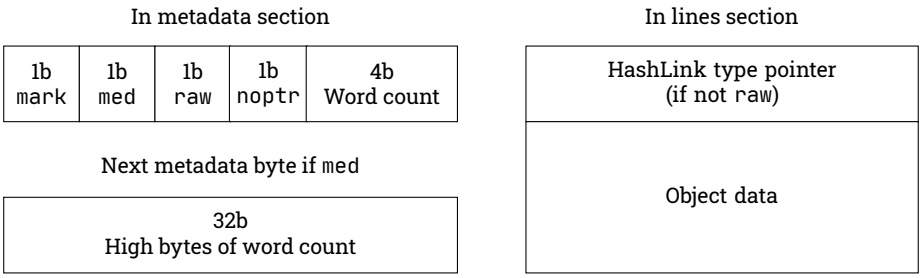

Figure 3.4: Object layout.

- **Mark bit** (`mark`): used during the marking phase to check if an object was previously marked[5].
- **Medium-size bit** (`med`): indicates that the object is larger than 16 words (i.e. 1 line) in size.
- **Raw bit** (`raw`): indicates that the object does not start with a HashLink type pointer.
- **No-pointer bit** (`noptr`): indicates that the object fields contain no pointers and should not be scanned.
- **Small-size words**: 4 least-significant bits of object size in words.

If the medium-size bit is set, then the following 4 metadata bytes contains the next bits of the object size in words. Because a zero-word allocation is invalid, the size in words is first decremented by one before storage, allowing objects of 16 words in size, inclusive, to be stored without setting the medium-size bit. Larger objects are stored using separate pages, and their size is calculated from the page size stored in the page header.

3.3.8 Allocation: fast path

Under normal conditions, the majority of allocations will follow the fast path. In the fast path, a thread has a thread-local block from which it can allocate objects, and it has an allocation cursor and an allocation limit in that block. If there is enough space to accommodate the allocation request, the allocation cursor is incremented by the object size (rounded up to a multiple of the word size) and the original cursor position is returned. In pseudo-code, this can be expressed as:

```
def bump_alloc(size):
  obj = initialise(cursor)
  cursor += size
```

[5] In the implementation, the meaning of the mark bit changes with every cycle so it need not be reset.

```
  return obj

def alloc_fast_path(size):
  if cursor + size < limit:
    return bump_alloc(size)
  return alloc_slow_path(size)
```

3.3.9 Allocation: slow path

If there is not enough space to accommodate the allocation request, the action taken depends on the size of the object.

For small objects (size less than or equal to one line) a "gap search" is performed in the active block. This is a linear search through the line markings which were set in the last sweep cycle by the GC. If a gap is not found, a block is recycled if available, or a fresh block is taken from the global pool.

For medium objects (size less than a quarter of a block) the search is skipped. Instead, a block is recycled if available, or a fresh block is taken from the global pool.

For large objects a "huge" page is allocated directly to hold the object.

The full allocation algorithm can thus be summarised as:

```
def alloc(size):
  // fast path
  if cursor + size < limit:
    return bump_alloc(size)
  // slow path
  cursor = line_align(cursor)
  if size <= LINE_SIZE:
    if gap_search(current_block):
      return bump_alloc(size)
    full_list.append(current_block)
    if recyclable_blocks:
      current_block = recyclable_blocks.pop()
      if gap_search(current_block):
        return bump_alloc(size)
    current_block = free_pool.pop()
    return bump_alloc(size)
  else if size <= MEDIUM_SIZE:
    current_block = free_pool.pop()
    return bump_alloc(size)
  else:
    return gc_alloc_page_huge(size)
```

3.3.10 Object initialisation

No matter which allocation path an object takes, the garbage collector is only responsible for memory allocation and system initialisation (according to Jones [12]). This means that the pointer returned from an allocation request has its metadata set in the block header, allowing the GC to identify the object's size at a later time, but not much more than that.

In case of a Haxe to HashLink compilation, type safety and secondary initialisation is ensured by subsequent operations on the allocated objects. In the case of C code interacting with `libhl`, this burden is on the C code developer.

3.3.11 Marking

A collection cycle starts with the mark phase. The GC maintains a dynamically growing stack of pointers called the "mark stack". The mark stack is filled first by following the GC roots, which are application-specific and include references to static instances and various pieces of the standard library (when compiling from Haxe).

The mark stack is then supplemented with the pointers found by scanning the thread stacks and registers. A standard register-spilling procedure based on `setjmp`[6] is used. The thread context is saved into a `jmpbuf` using a `setjmp` call, then the buffer is scanned at word-aligned increments. On all tested platforms, the values found in the `jmpbuf` correspond to the values contained in the CPU registers before the `setjmp` call.

The mark phase then enters a tight marking loop, removing pointers from the top of the stack one at a time, scanning them according to their type information (if any), and adding any found, not-yet-marked references. This results in a depth-first scanning order. Care has been taken to reduce the number of branches in the marking loop, in order to avoid branch prediction misses and to enable good CPU pipelining. See Section 3.4.2 for more detail.

As per [3], lines are marked in the per-block line mark bytemap when scanning an object rather than when adding it to the mark stack.

3.3.12 Sweeping

In the sweep phase, the collector iterates through all allocated pages. Based on the bitmap of used blocks stored in the page header, it scans blocks that might need to be swept. If the line marks (marked in the mark phase) indicate that a block is not used anymore, it is put back into the free block pool. If the line marks indicate that there is some usable space and the block is considered **Full**, it is moved to the thread's recycle list instead. See Section 3.3.6 for more detail.

[6] See `https://linux.die.net/man/3/setjmp` for complete documentation.

At the end of the sweep phase, the GC has statistics on the overall usage of the heap. These are used as heuristics to decide whether the heap should be grown by allocating additional pages. This aims to prevent heap thrashing, which is a situation in which the GC is invoked multiple times in short succession because it never frees enough memory for the demands of the application.

3.3.13 Thread cooperation

Threads in HashLink are assumed to be cooperative. The GC is a "stop-the-world" collector, meaning that it requires all threads to reach a safe stopping point before a collection cycle can begin. Such stopping points occur in all allocation requests, which happen very often.

This also implies that a thread performing a particularly long operation without any allocations may cause the other threads to halt. However, there are no plans to change this aspect of HashLink at the moment.

When a thread exits, the objects it allocated may still be visible to other threads. In this case, all of the thread's blocks are marked **Zombie** and cannot be used for fresh allocations until all of the objects they contained are cleared. See Section 3.3.6 for more detail.

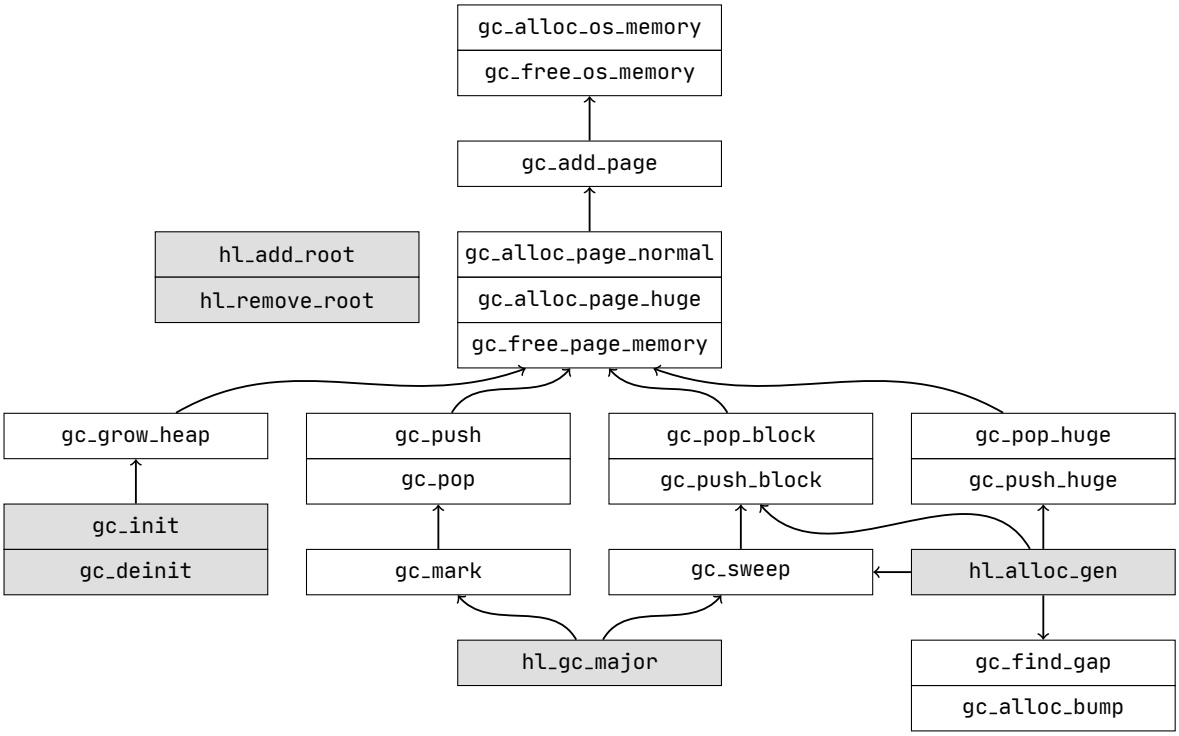

Figure 3.5: Function and state variable dependencies. Shaded blocks represent the public API.

3.4 Optimisations

In this section we explore some of the optimisations of the implementation, regardless of whether they were implemented or only considered.

3.4.1 Hot paths

In realistic garbage-collected applications, allocation is an extremely common operation. Ideally, the allocation routine should be heavily optimised towards allocating small objects as fast as possible. Immix is particularly suited for this, as the most common case when allocating is bump allocation (see Section 3.3.8).

This algorithm is mirrored in the C implementation, where the size check is the first (and potentially only) thing that needs to happen during allocation. Because the majority of allocated objects are small, the branch for when an object fits is additionally tagged with a branch hint, which may cause compilers to emit a branch hint instruction for processor architectures that support it.

3.4.2 Branches in marking loop

Another possible source of latency comes from the marking phase, in which the object graph needs to be traversed to find which objects are reachable and which are not. Ideally, the marking loop could treat objects homogeneously, avoiding the need for branches inside the loop. The branch predictor can remedy some of the negative effects of branching in the loop, but certain allocation patterns might still result in a marking order in which the kinds of marked objects are difficult for the CPU to predict mechanically.

Raw allocation branching In Section 3.3.7, there is a difference in object layout depending on whether the `raw` bit is set. For "raw" objects, the first word of the allocated data is not a pointer to a HashLink type structure. The type in turn contains a `mark_bits` field, which identifies which words of the allocated objects are pointers and therefore should be scanned.

The purpose of raw allocations is to be able to pass GC-allocated structures directly to external libraries. The fields are allocated in an order that aligns with the C definition of the struct expected by the library. Adding an extra type field at the beginning would disrupt this process. Although it is possible to pass a pointer to the middle of the allocated data (offset forward by the size of the type pointer), this complicates the correct identification of allocated data, and would therefore require branching in a different part of the marking loop instead. As such, this particular branch was kept in the loop for simplicity.

Size branching As per the original Immix publication, the lines to which an object belongs are marked when the object is being scanned. However, the line marking code needs to distinguish between small, medium, and huge objects, since the lines are marked differently. Additionally, objects that are not small have extra metadata indicating their full size that needs to be read conditionally (see Figure 3.4). This branch was also kept in the loop, since the trade-off would be a large increase in memory consumption for object metadata, requiring a full 32-bit integer for any object rather than 8 bits for most objects.

3.4.3 Allocation cursor in global register

One final optimisation we considered, but did not implement due to limited compiler support and interoperability with C libraries, was the placement of the allocation cursor and limit into global registers. This would allow any allocation to be even faster, since allocations would not need to load the allocation cursor and limit from memory. Register storage would be even more efficient than CPU cache which most likely contains the allocation cursor and limit in the current implementation.

3.5 Differences from Immix

The implementation of the garbage collector in HashLink is primarily based on the original Immix publication by Blackburn and McKinley, but there are some differences. Several of these are based on the fact that no implementation code was copied, that the publication does not specify all parts of the collector in detail, and that the publication assumes a JVM environment.

However, some differences were necessitated by the requirements of HashLink. We describe these in the following sections.

3.5.1 No compaction

Immix compacts blocks in the heap that contain few objects to reduce fragmentation. This requires setting up forwarding pointers and rewriting the original references to the moved objects. However, because HashLink/C compiles via an intermediate C stage, identification of pointers in thread stacks and registers is not always reliable. In rare cases, code produced by the C compiler will keep references to the interior of allocated objects (i.e. “interior pointers”), rather than their beginning, even if the original C code does not. Additionally, moving objects on the heap would require modifying the GC interface, which would not interoperate very well with external C libraries.

During benchmarking we found that even without compaction, our collector performs well. Nevertheless, more investigation is needed to see if it is possible

to make C compilers more cooperative. It would also be necessary to implement object pinning[7] so that objects can be passed on to external libraries.

3.5.2 Metadata storage

The Immix publication describes a very small overhead for object metadata. In JVM implementations, all allocated objects have headers which can fit a number of bits for the exclusive use of the collector. This is not the case in HashLink, and keeping the object layout intact was a design constraint for this project. Instead, object metadata is stored in sections of the block header. The result is that the block header overhead is larger, but objects are slightly smaller in general.

3.5.3 Huge objects in blocks

In the original publication, there is a limit for "medium-sized" allocations. Objects larger than roughly a quarter of a block are always allocated separately. To avoid branching in the allocation routine, and because the metadata layout conveniently supports this, our implementation allows allocating huge objects inside blocks if they fit on the "first try" (i.e. during fast path allocation). In this case, they are treated as medium-sized objects, even though their size exceeds the threshold.

3.6 Debugging and development timeline

The development of the collector was started with a prototype entirely independent of HashLink. The collector API was modelled after the functions present in HashLink to ease eventual integration, but otherwise no part of the HashLink runtime was used. We checked the correctness of the prototype using a test suite with cases designed to trigger different behaviours of the collector, e.g. allocating many small objects, allocating many medium objects, or creating pointer cycles across blocks.

The final test added to the prototype test suite was a C port of the `mandelbrot` benchmark (used in GC benchmarks, see Chapter 4). This test uses a mixture of small and large objects and performs a large number of allocations. When the prototype correctly executed the test, i.e. with the same result as the old collector, we felt confident about integrating the prototype into the actual HashLink codebase. The prototype tests were adapted to work with the HashLink runtime as well, in order to verify that the integration did not cause any regressions in functionality.

Although HashLink's original GC was designed with some amount of extensibility in mind, we chose to not tie the new GC to this framework too closely, as we found it easier to make rapid changes with a "blank slate" in this area. Re-aligning

[7] A "pinned" object is never moved by the collector.

the new GC with the original framework will be the subject of future work (see Chapter 7).

During development, crashes due to memory violations were extremely common, as is probably the case during any GC-related development. We made extensive use of the interactive debugger IDA Pro [39] to verify memory contents, inspect the assembled code (and its automatic disassembly and C-like decompilation), and step through various programs.

We also added rudimentary support for event logging (inspired by Elephant Tracks [27]) with a custom binary format readable by a small utility written in Haxe. Although this has not been developed further, it helped us identify cases of heap thrashing in some benchmarks.

In the final months of the project, we started formal verification (see Chapters 5 and 6) of the collector implementation. Due to the relative impracticality of the verification method we chose, starting the verification process before reaching a relatively stable implementation likely would have been counter-productive.

Chapter 4

Evaluation

4.1 Introduction

In this chapter, we discuss the improvements obtained with the new GC implementation. In Section 4.2, we briefly discuss our new benchmarking framework. In Section 4.3, we enumerate all the benchmark cases. In Section 4.4, we describe the setup used to obtain our data. In Section 4.5, we show the obtained data comparing the two GC implementations. Finally, in Section 4.6, we summarise our findings.

4.2 Benchmarking framework

At the outset of the project, there were only a small number of benchmark cases available for Haxe, most of which were small. As part of this project, we contributed a new benchmark framework[1], allowing easier creation of new cases.

The original benchmarking setup required a separate GitHub repository per benchmark case, even though most files in the repository would be identical from case to case[2]. These files were primarily scripts that would compile and run the code for a particular target, using a particular Haxe compiler version.

In the new benchmarking framework, we implemented the functionality of the scripts directly in Haxe, localising the implementation to a single library that is easier to maintain and update. New benchmark cases consist of sub-directories in the `benchmark-runner` repository and at minimum only contain a declaration of the benchmark name and the code to run[3].

[1] Available at `https://github.com/HaxeBenchmarks/benchmark-runner`, updated by other members of the community since its conception.

[2] For example, the original implementation of the `json` and `mandelbrot` benchmarks: `https://github.com/HaxeBenchmarks/json-benchmark`, `https://github.com/HaxeBenchmarks/mandelbrot-benchmark`.

[3] The new `json` benchmark: `https://github.com/HaxeBenchmarks/benchmark-runner/tree/master/cases/json`.

Benchmark name	Type	Approximate LoC
`alloc`	micro	50
`binarytrees`	micro	50
`json`	micro	20
`mandelbrot`	micro	100
`mandelbrot-anon`	micro	100
`nbody`	micro	150
`bcrypt`	crypto	400
`sha256`	crypto	400
`sha512`	crypto	400
`formatter`	app	10,000
`formatter-noio`	app	10,000

Table 4.1: List of benchmark cases.
"micro", "crypto", and "app" refer to "microbenchmarks", "cryptography benchmarks", and "application benchmarks", respectively.
The lines of code (LoC) include approximate sizes of library dependencies, but not the standard library (which may or may not be implemented in Haxe, depending on the target).

4.3 Benchmark cases

The full list of benchmark cases used for measuring the performance of the new collector is shown in Table 4.1.

Many of these benchmarks are not "realistic" in terms of application load, in that they focus heavily on allocations and not enough on program logic, I/O operations, user interactions, etc. Nevertheless, in evaluating a garbage collector, even allocation-based benchmarks seem relevant.

The `formatter` and `formatter_noio` are more large-scale applications that have a performance load similar to a compiler. `formatter` reads a number of large Haxe codebases, parsing them, then producing the re-formatted code. `formatter_noio` is the same, but disk I/O is not included in the measurement because files are pre-loaded.

4.4 Measurement setup

The results in the following sections were obtained on a Macbook Pro, "late 2011" model, with the following parameters:

- **OS**: Mac OS X 10.9.5
- **CPU**: 2.4 GHz Intel Core i5
- **RAM**: 16 GiB 1600 MHz DDR3

Benchmarks were also executed on a Ubuntu 18.04 server with a 3.8 GHz Intel Xeon. While the benchmark times were much shorter for both the old and the new GC, the relative performance was in the same ratio as observed on the Macbook. A continuously updated report of the benchmark performances is available online at the Haxe benchmarking server[4], which internally uses our new benchmarking framework.

All benchmarks were executed 20 times, in order to lower the effects of variance.

4.5 Results

This section compares the performance of the new GC with the performance of the original HashLink GC on all the benchmark cases listed in Table 4.1.

In Figures 4.1 to 4.11, the box plots show the non-outlier maxima and minima with "whiskers", the box extents show the first and third quartiles, and the line inside the box shows the median. Any outliers are identified as samples outside the 1.5 IQR range from the first or third quartiles and are plotted as individual points. The top two results in each graph represent the performance of the old GC implementation, whereas the bottom two results, shown in red, represent the performance of our implementation. In all cases, a lower time is better.

The full tabulated dataset of samples for the following graphs is avaiable in Appendix B.

4.5.1 Microbenchmarks

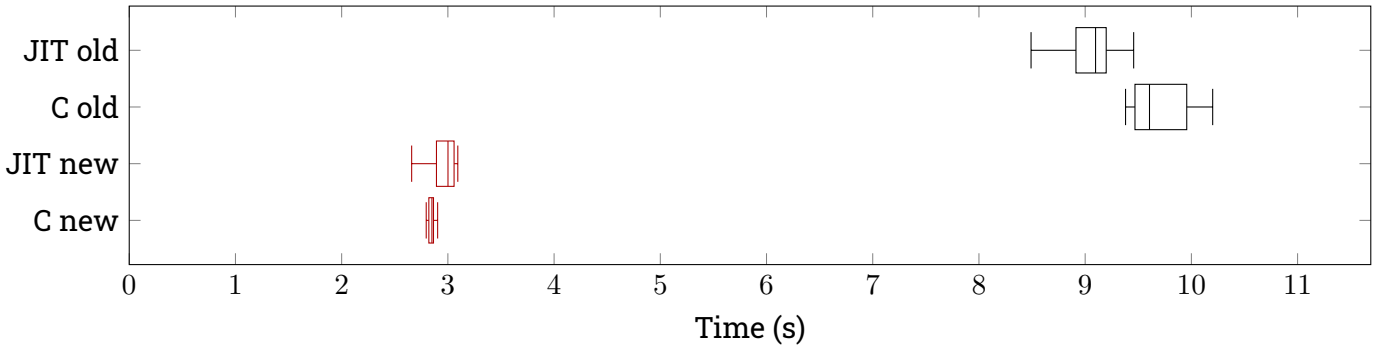

Figure 4.1: `alloc` results.

[4] `https://benchs.haxe.org/`

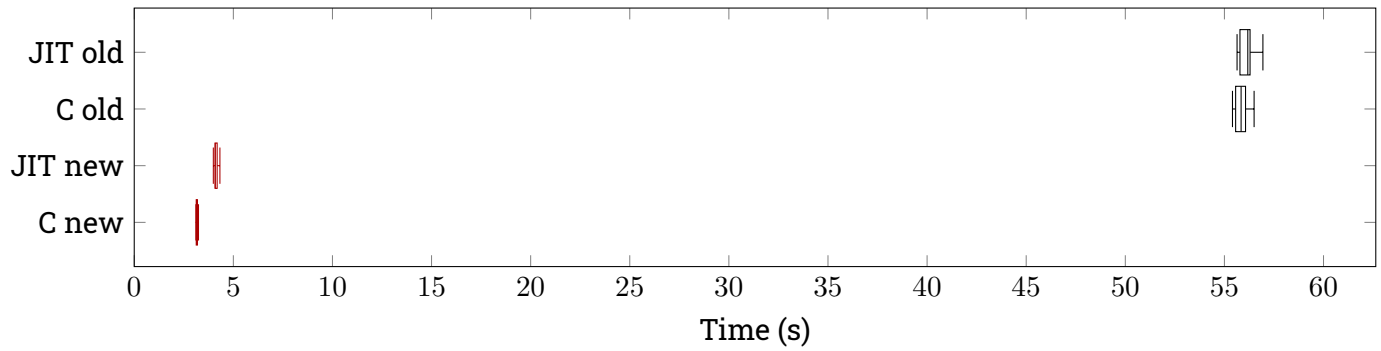

Figure 4.2: `binarytrees` results.

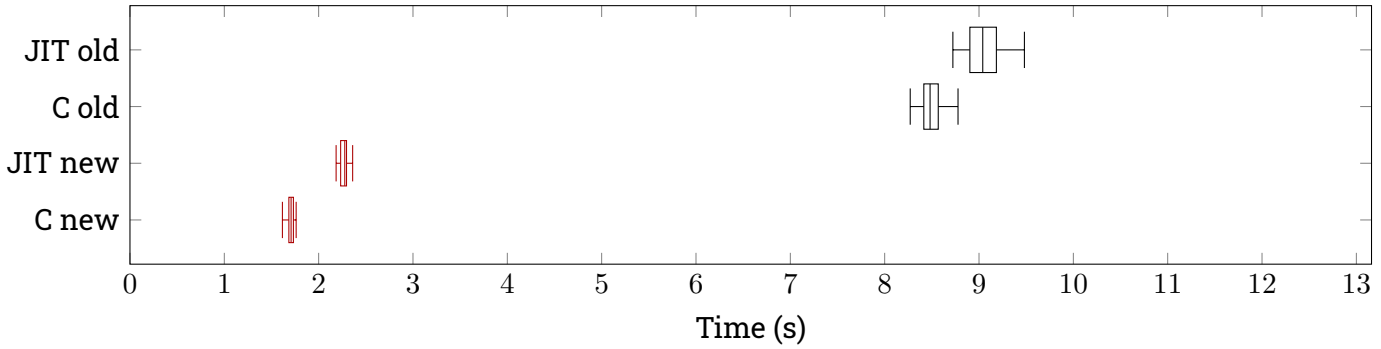

Figure 4.3: `json` results.

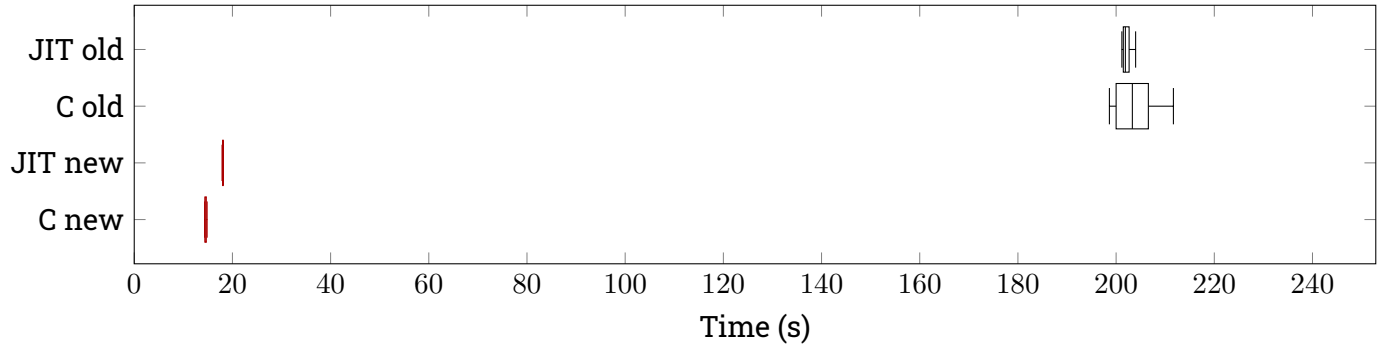

Figure 4.4: `mandelbrot` results.

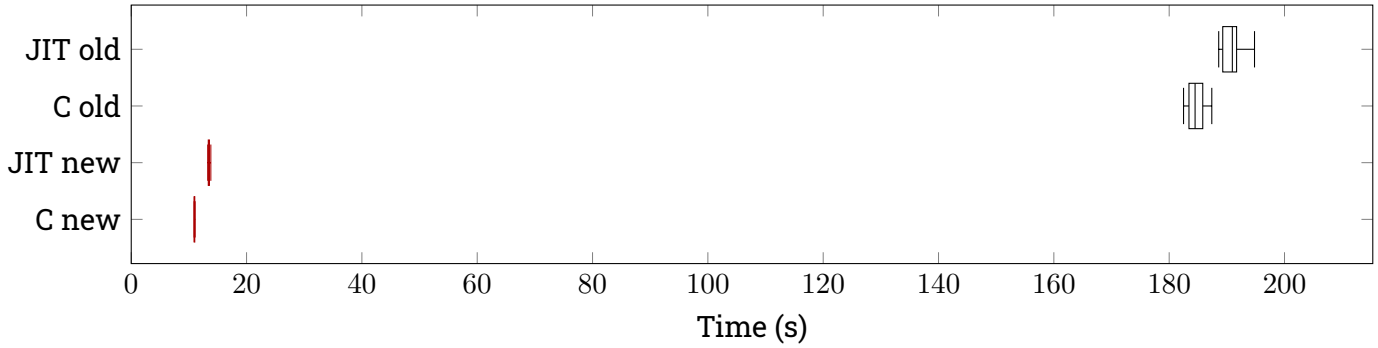

Figure 4.5: `mandelbrot-anon` results.

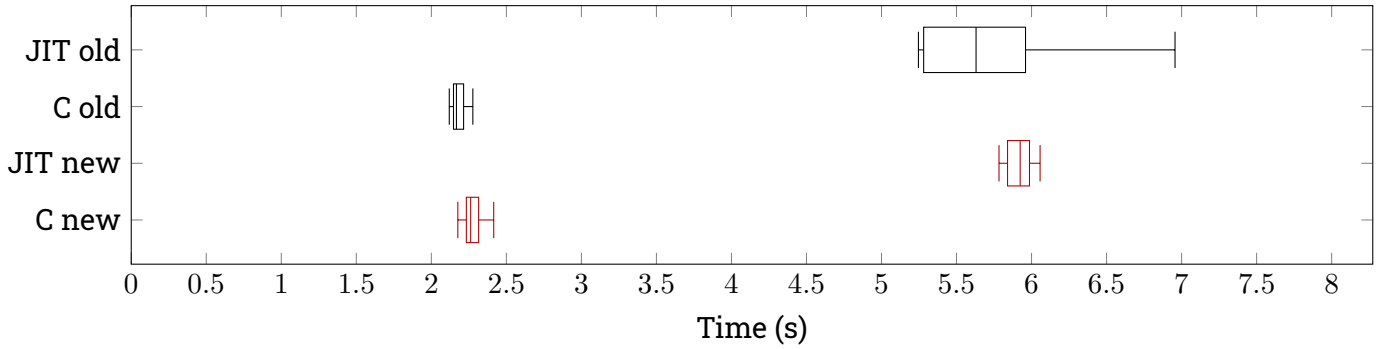

Figure 4.6: `nbody` results.

4.5.2 Cryptography benchmarks

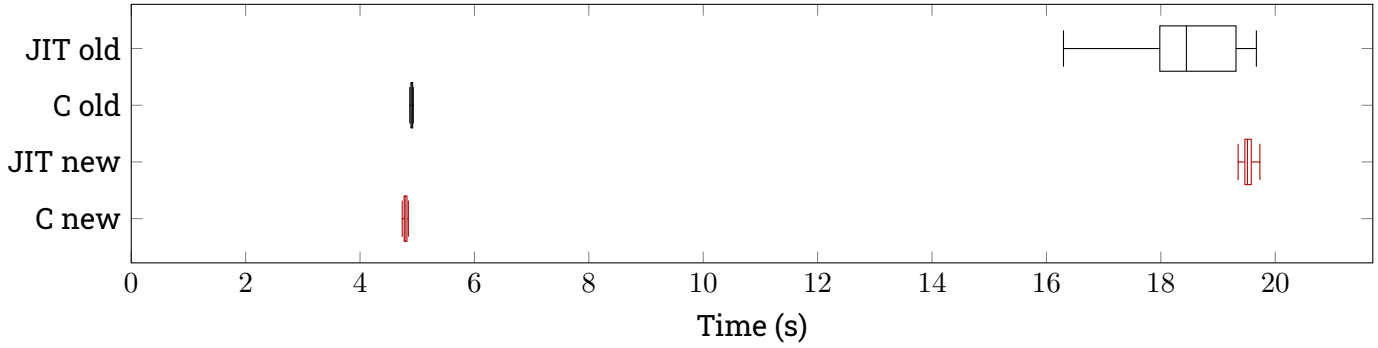

Figure 4.7: `bcrypt` results.

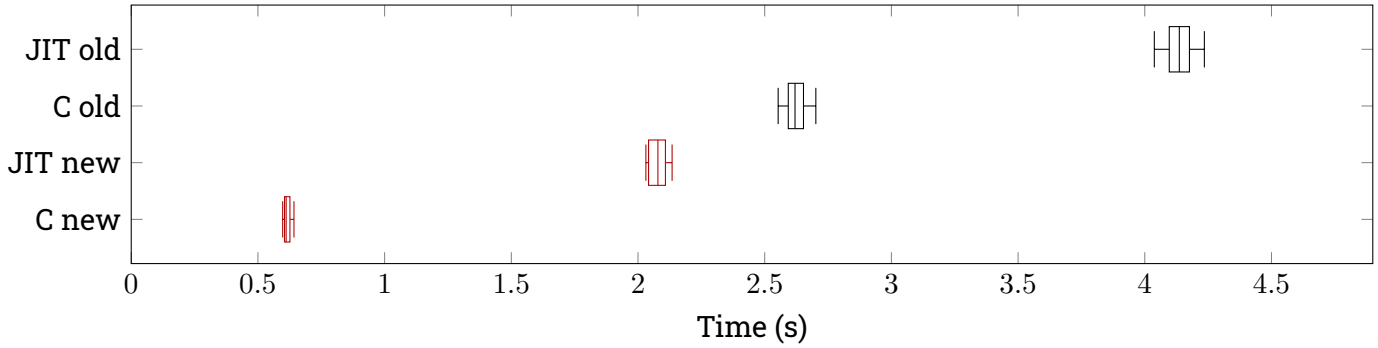

Figure 4.8: `sha256` results.

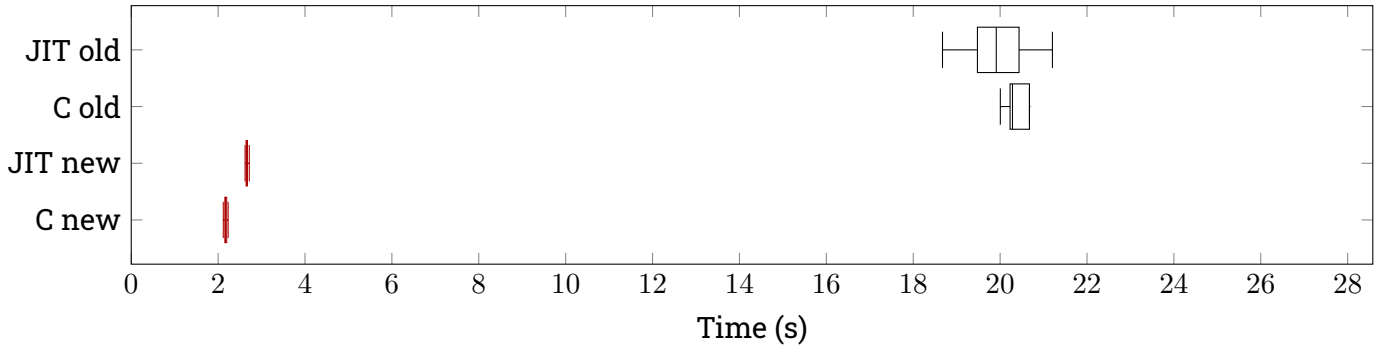

Figure 4.9: `sha512` results.

4.5.3 Application benchmarks

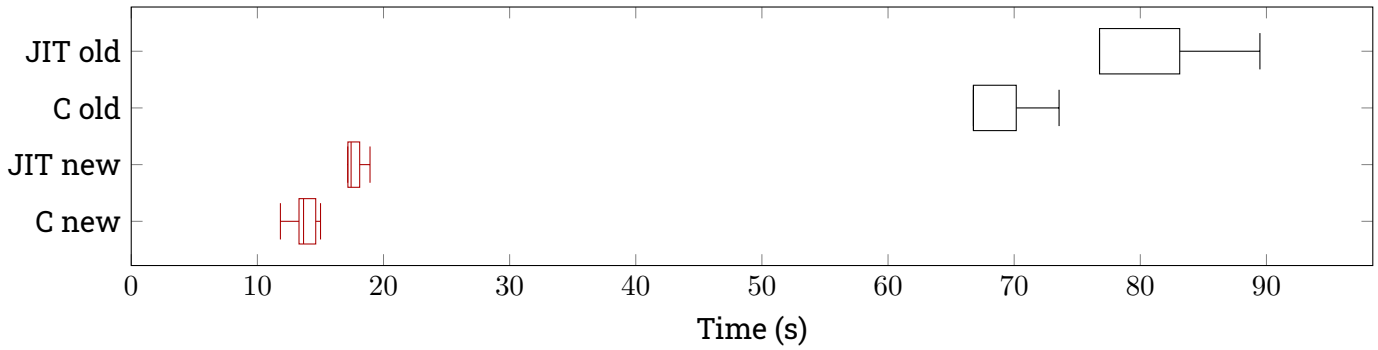

Figure 4.10: `formatter` results.

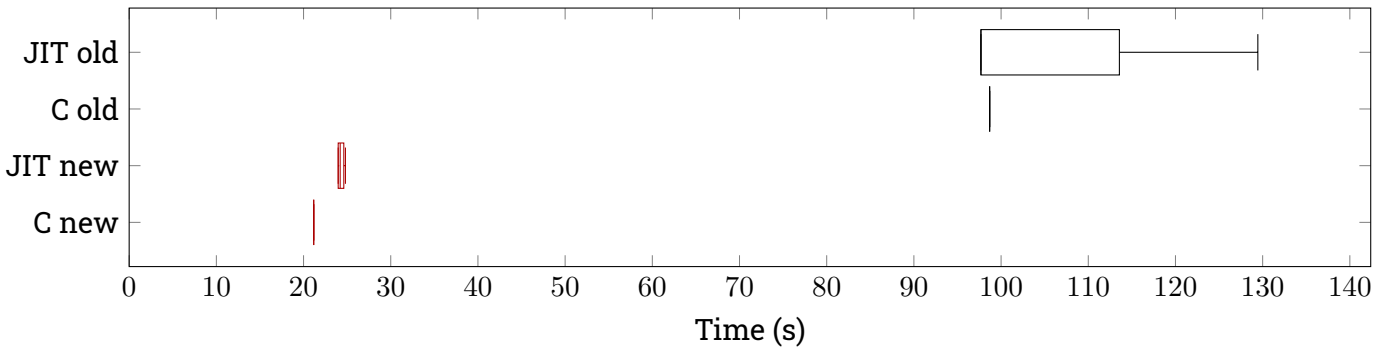

Figure 4.11: `formatter-noio` results.

4.6 Summary

Figures 4.12 and 4.13 show the median performance improvement on all benchmarks. Certain benchmarks, such as `nbody` or `bcrypt` show virtually identical results with the new collector, and instead their time depends only on whether the runtime is HL/JIT or HL/C. These benchmarks allocate very little so this is to be expected. On the other hand, heavily allocating benchmarks, such as `binarytrees` or `mandelbrot` show the best improvements, on the order of 12× faster performance.

As noted in Section 2.4, large, application-based benchmarks are more indicative of the performance of a particular GC than microbenchmarks. As such, the results obtained from the `formatter` and `formatter-noio` are the most important, showing a substantial improvement of 5× faster performance.

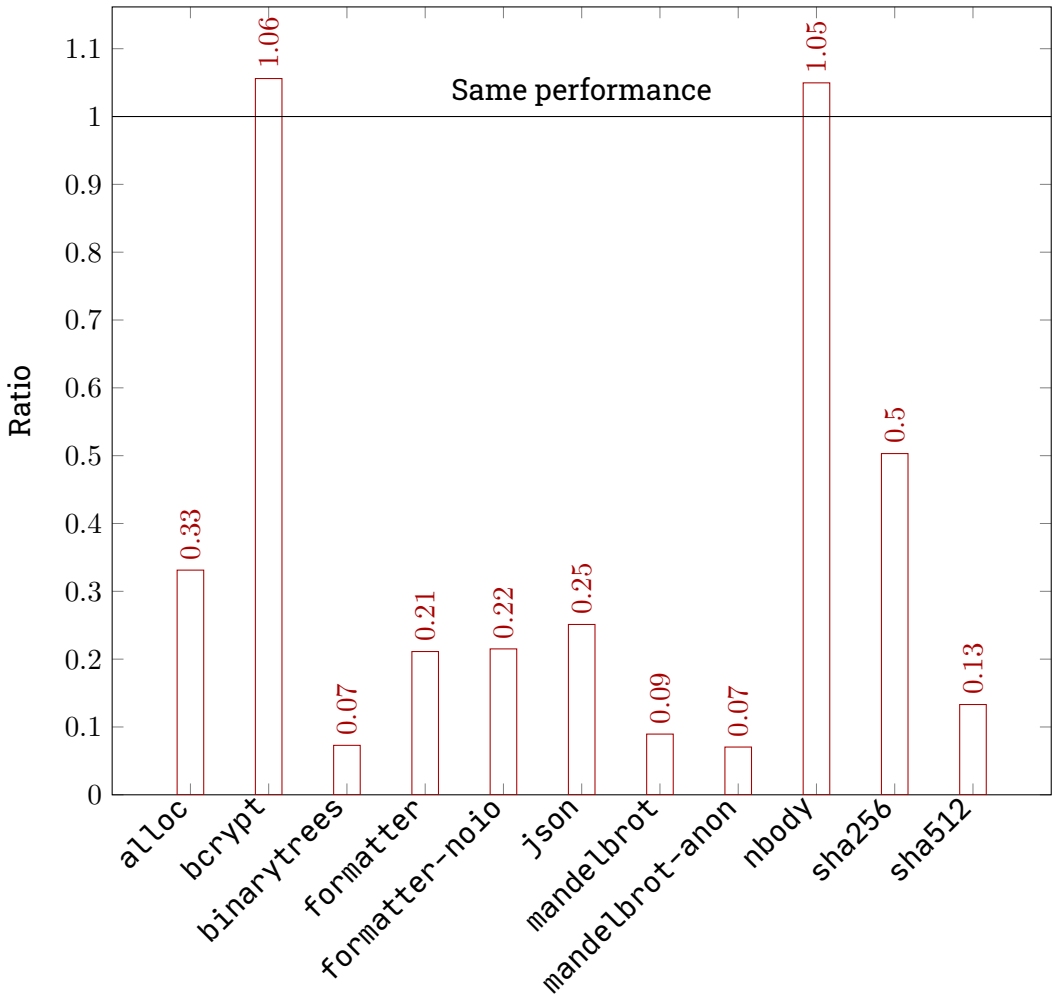

Figure 4.12: Summary of benchmark results in HL/JIT mode. The Y axis shows the ratio of the median time reached with the new GC to the median time reached with the old GC on all benchmarks.

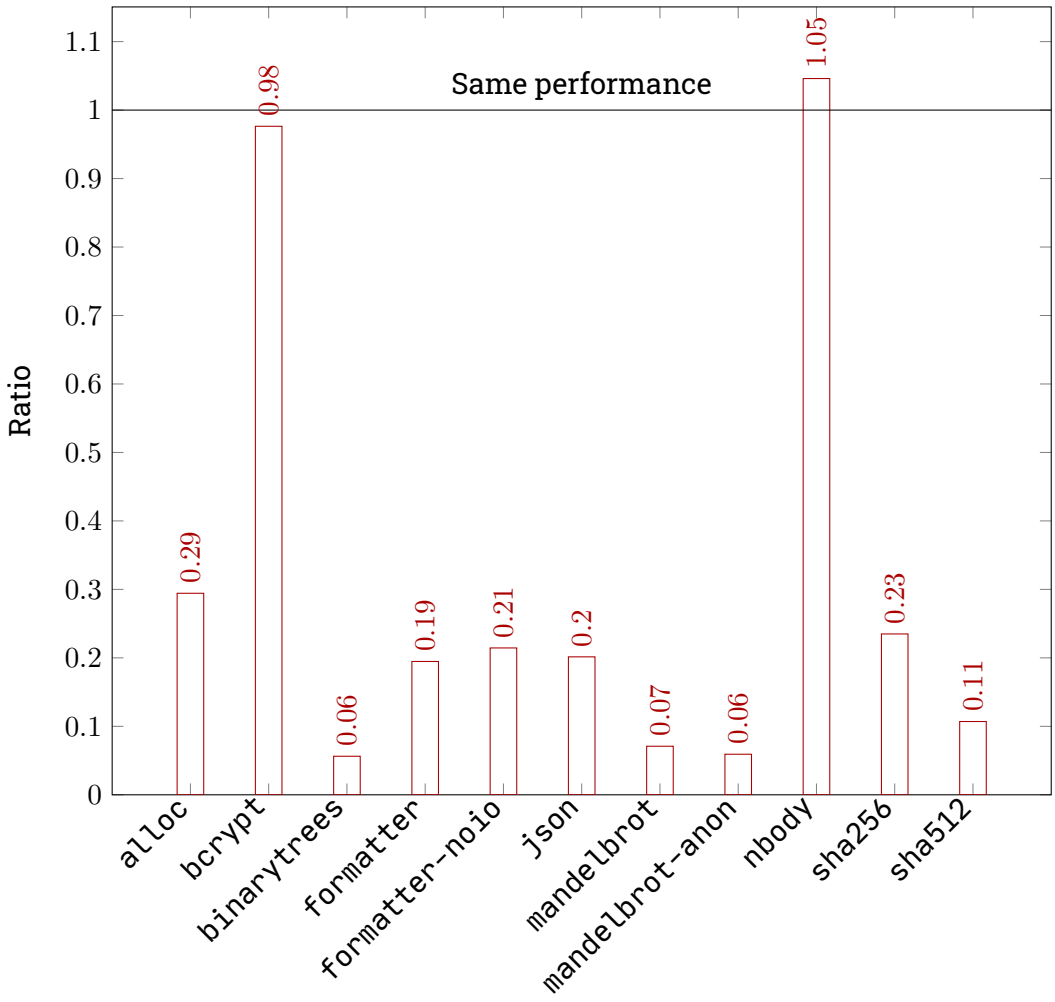

Figure 4.13: Summary of benchmark results in HL/C mode. The Y axis shows the ratio of the median time reached with the new GC to the median time reached with the old GC on all benchmarks.

Chapter 5

Abstract model

5.1 Introduction

In this chapter, we describe an abstract model of our Immix-based collector. In Section 5.2, we describe the motivation behind defining an abstract model. In Section 5.3, we discuss the data model of our abstraction. In Section 5.4, we describe the functional specifications in our model.

5.2 Motivation

The primary goal of this project, and indeed many garbage collectors, was to obtain very good performance in benchmarks while remaining functionally correct. To this end, the implementation is written in C code optimised for speed, not readability or ease of reasoning. Additionally, C is a low-level programming language and so code written in it must make important choices about specific data structures to use, layout of structures in memory, and so forth.

For these reasons, we found it natural to define a high-level model of the collector. This abstract model performs the same general Immix-based collection algorithm, but is as independent of specific memory layout as possible. For instance, where the implementation may use a hash map, a singly-linked list, or a doubly-linked list, the abstract data model uses Coq's built-in lists.

5.3 Data model

To define the state of the abstract model, we define a set of Coq types. These types consist of nested `Record` types and Coq primitive types (natural numbers, lists, Booleans). The state should be minimal and we avoid defining a specific representation and layout of data in memory.

Pages, blocks, lines, and objects are crucial to the collector operation. Although object arrangement in lines is somewhat related to memory layout, we cannot define Immix-based line marking and collection without this facet modelled.

The final type defined in the data model is `imx_state`, which bundles together all the runtime state of the collector, along with the set of allocated pages, their blocks, and objects, represented by the `imx_page`, `imx_block`, and `imx_object` types, respectively.

```
Record imx_state := {
  os_allocs    : list (nat * Z);
  os_alloc_id  : nat;
  pages        : list ImxPage.imx_page;
  roots        : list (nat * nat * Z);
  alloc_cursor : (nat * nat * Z);
  alloc_limit  : Z;
  pool         : list (nat * nat);
  recyclable   : list (nat * nat);
  object_count : nat;
  mark_stack   : list (nat * nat * Z);
}.
```

We define the lemma `imx_state_valid` to mean that the given instance of `imx_state` is valid and self-consistent. For example, each block in the `pool` list must be contained in one of the currently allocated pages. `imx_state_valid` is an invariant that is maintained by all functions of the functional specification (see Section 5.4).

5.3.1 Preprocessing step

The data model additionally defines a number of Coq functions to more conveniently update values in the state. In particular, even though Coq has a record type syntax, it does not have syntax to update a specific field of a record. We automatically generate field updaters in a preprocessing step to avoid additional Coq library dependencies[1].

Our preprocessor is triggered by special comment syntax. As an example, consider a record type `sometype` with fields `foo` and `bar`:

```
Record sometype := {
  foo : nat;
  bar : list nat;
}.
(*@gen_setters sometype foo bar*)
```

After running the preprocessor, the comment is transformed into:

[1] Tej Chajed's `coq-record-update` is one such library, available at `https://github.com/tchajed/coq-record-update`.

```
Definition upd_foo value (original : sometype) : sometype := {|
  foo := value;
  bar := (bar original);
|}.
Definition upd_bar value (original : sometype) : sometype := {|
  foo := (foo original);
  bar := value;
|}.
```

We avoid name collisions by packing record types and their associated functions into individual modules, e.g. `ImxState` contains `imx_state`. This means types need to be qualified with their module path, but this is only a minor nuisance.

5.3.2 State storage

Records in Coq are immutable. Any update operation we might wish to apply to a record can only be modelled as the creation of a new record instance with some values updated. This could cause problems if we had multiple record instances that should represent the current state of the same object in memory, because updating one record would not update the other. This is the case with `imx_state`—it has a list of `imx_page` instances which then have lists of blocks, then objects—but sometimes the abstract model functions hold a local reference to a page.

We chose to resolve these problematic situations by adding identifier fields to each record type, allowing them to be looked up and updated uniquely from the `imx_state` instance.

5.4 Functional specification

5.4.1 Type signature

For each function of the collector implementation, we define an abstract version that operates on the data model. The signature of these functions is one of:

```
imx_state -> imx_result Out
In -> imx_state -> imx_result Out
```

Where `In` is an optional argument type and `Out` is the result type. `Out` may be `unit`, if there is no return value. `imx_result` is a wrapper type that encodes successful results and possible errors:

```
Inductive imx_result (Out : Type) : Type :=
  | imx_ok (res : Out) (state : imx_state)
  | imx_error (msg : string).
```

An `imx_error` is an irrecoverable error that is propagated up the call chain. There is no exception handling and a failed operation is later mapped to an unsatisfiable heap predicate. See Chapter 6 for more detail.

5.4.2 Monadic notation

To avoid having to pattern match on `imx_result` after every call, we define a monadic notation[2] for the model states and results. We find the resulting function definitions a lot more readable and easier to reason about. For a comparison of the notation applied to a trivial example see Figure 5.1. Table 5.1 describes all of the added syntax. The syntax is composable, and may be mixed with regular Coq syntax, such as `let x := ... in ...`.

```
Fixpoint imx_gc_grow_heap
    (count : nat) : imx_func unit :=
  match count with
  | S n =>
    imx_gc_alloc_page_normal !;
    imx_gc_grow_heap n
  | _ => imx_return tt
  end.
```

```
Fixpoint imx_gc_grow_heap
    (count : nat)
    (state : ImxState.imx_state)
    : imx_result unit :=
  match count with
  | S n =>
    match imx_gc_alloc_page_normal state with
    | imx_ok _ state =>
      imx_gc_grow_heap n state
    | imx_err msg => imx_err msg
    end
  | _ => imx_ok tt state
  end.
```

Figure 5.1: Example of monadic notation. The two versions are functionally equivalent, but the left one uses our notation. Note that we defined `!;` as the monadic bind operation.

Syntax	Type of `f`	Description
`x <-^ f ; ...`	`imx_state → X`	Read from the state.
`x <- f ; ...`	`imx_state → imx_result X`	Function call, assigning result to `x`.
`f !; ...`	`imx_state → imx_result X`	Function call, discarding result.
`f ^; ...`	`imx_state → imx_state`	Write to the state.

Table 5.1: Monadic notation. `x` is a variable identifier, available in the scope after the call. `f` is one of the functions of the functional specification. Function calls may fail, in which case the failure is propagated as the result of the entire monad chain.

5.4.3 Functions

Table 5.2 shows a full list of the functions of the abstract model. Each function mirrors the functionality of the C implementation, but is applied to the abstract data

[2] Coq's parser allows introducing custom extensions to the syntax without having to modify the entire compiler.

Function name	Input type	Output type
gc_alloc_os_memory	Z	nat
gc_free_os_memory	nat	–
gc_add_page	imx_page	–
gc_alloc_page_normal	–	imx_page
gc_alloc_page_huge	Z	imx_page
gc_free_page_memory	Z	–
hl_add_root	(nat * nat * Z)	–
hl_remove_root	(nat * nat * Z)	–
gc_push_block	imx_block	–
gc_push_huge	imx_object	–
gc_push	(nat * nat * Z)	–
gc_pop	–	imx_object
gc_mark	–	–
gc_sweep	–	–
hl_gc_major	–	–
gc_alloc_bump	Z	imx_object
gc_find_gap	–	bool
gc_pop_block	–	imx_block
gc_pop_huge	Z	imx_block
hl_gc_alloc_gen	Z	imx_object

Table 5.2: List of functions in the functional specification of the abstract model. All function names are prefixed with `imx_func_` in the Coq code. "–" as an input type indicates no input argument. "–" as an output type indicates the result is `unit` in Coq.

model. The abstract model is approximately 600 standard[3] LoC in size, compared to the 1200 standard LoC of the C implementation. The functional specification is also less complex due to the simpler syntax of Coq.

For functions that take some arguments in the C implementation, we map these as follows:

- **Object sizes**: mapped as a single `Z`-typed argument. `Z` in Coq efficiently represents any integer, including negative numbers, but this mapping allows us easier transition from the VST representation, discussed later.
- **Arbitrary pointers**: mapped as a tuple `(nat * nat * Z)`. The first component represents the ID of the page, the second is the block number, and the third is the offset inside the block.
- **Typed pointers**: in some situations, such as in `gc_push_block`, it is more convenient to accept as an argument the record representing a model object,

[3]Excluding comments and empty lines.

rather than its identifier (see Section 5.3.2). This is usually used for arguments that are not modified.

5.5 Correctness

Although the abstract model is much shorter and more readable than the C implementation, a curious reader might still wonder whether the model itself is correctly defined. In our project, we chose to address this issue with a test suite evaluating the behaviour of the model in a variety of situations. We believe this is sufficient, because it would require a consistent, systematic error to produce the same issue in both the C implementation, the Coq model, the abstract model test suite, and the VST proofs (defined in Chapter 6) linking the abstract model to the implementation. Alternative approaches exist, such as Hawblitzel's and Petrank's [32], which establishes the correctness of the collector based on invariants that must persist across execution of its functions.

Chapter 6

Formal verification

6.1 Introduction

In this chapter, we describe the verification process used for the implementation. In Section 6.2, we give the high-level overview of the proof method. In Section 6.3, we describe a mapping from the abstract model to VST predicates. In Section 6.4, we describe the process of generating VST specifications. Finally, in Section 6.5, we discuss the specifics of the proof process.

6.2 Proof overview

Because the collector is designed to integrate into the existing HashLink project which consists of a codebase of considerable size[1] and many third-party library dependencies, the proof scope is limited to the operation of the GC only. We relate the abstract model defined in Chapter 5 to the C implementation using VST-Floyd[2] and CompCert. This process is similar to the approach taken by Koh et al. [31]

We complete the proof in the following steps, each of which is described in more detail in the subsequent sections:

1. **Model-to-VST mapping**: A mapping from model states to VST heap predicates is defined.
2. **VST specifications**: The pre- and postconditions of the C implementation functions are created from the model and functional specifications.
3. **VST proofs**: VST is used to prove that the postconditions are ensured given the pre-conditions and the C code.

[1] The HashLink project spans approximately 30,000 lines of C code, excluding library dependencies.

[2] VST is the Verifiable Software Toolchain, which includes the verified C compiler CompCert; the Verifiable C separation logic; and VST-Floyd, a proof automation library for Verifiable C proofs. We will usually refer to the latter as "VST" for brevity.

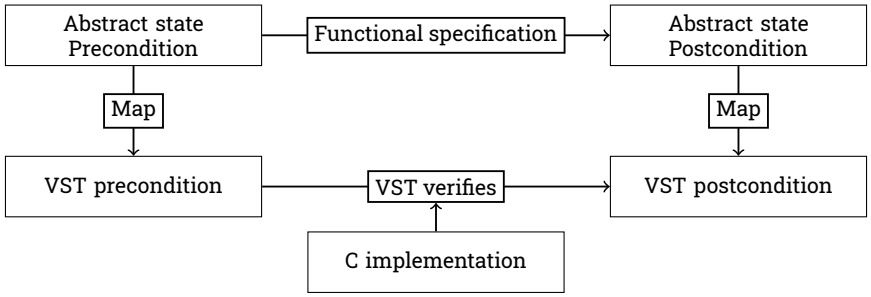

Figure 6.1: Proof overview.

A visual representation of the proof steps is shown in Figure 6.1. Note that the proof could have been completed without defining an abstract model. Instead, all of the collector functions could have been proven individually using standard VST specifications. However, this approach would have resulted in a much less useful proof since the pre- and postconditions would be very long and not obviously correct or connected to the operation of an Immix-based garbage collector. By basing our proof in a simple, relatively easy to understand model, we can be more confident in the validity of the proof itself. See also Section 5.5.

6.3 Model-to-VST mapping

To be able to use the abstract model in our VST proofs, we needed some method of connecting the states of the model with VST. In VST, function pre- and postconditions are "lifted predicates", which consist of pure logic propositions, separation logic conjuncts, local variables in the scope, and, optionally, existential qualifiers. Such a predicate can be thought of as a subset of all possible heaps. To connect the model states with VST, we considered two possible methods: converting VST heap predicates to instances of the model state, and the opposite, converting model state to VST heap predicates.

In our work, we concluded it to be too impractical to match on parts of a VST predicate in an attempt to map them to an abstract state. Due to the shadow state (explained below), we believe it is actually impossible to perform this mapping without losing some information.

Instead, we define a mapping from abstract states to VST heap predicates. In Coq types:

```
Definition imx_to_vst
    (gv     : globals)
    (shadow : vst_shadow)
    (state  : imx_state)
    : mpred := ...
```

This mapping function generates VST heap predicates corresponding to a particular abstract state. `globals` is a mapping of identifiers to symbols, which al-

lows us to refer to static variables defined in the C code along with the implementation functions.

The `vst_shadow` type encompasses all of the implementation-specific parts of the heap state, such as the specific arrangement of pages in the collector's hash map. We define the supplementary lemma `imx_shadow_valid` to mean that the given `shadow` is valid for the given `state`; for example, the number of specific addresses of OS allocations in `shadow` must match the number of abstract OS allocations in `state`. We require `vst_shadow` to exist because we made the choice to omit the specifics of memory layout from our abstract model. Without this additional information, we could only generate VST predicates that would be too generic, i.e. some heaps that would not satisfy the precondition would be considered valid, and likewise for the postcondition.

To the best of our knowledge, generating VST heap predicates from an abstract state (with or without the supplementary concrete state) is a novel method to reason about complex programs with a large amount of state variables.

6.3.1 Mapping components

The definition of the `imx_to_vst` function consists of a separating conjunction of several helper functions, one for each state variable. For example, the assertion about the state of the `gc_pool` variable is generated using the `gen_gc_pool` function. With this approach, we can treat calls to the generator functions as abstract predicates during our VST proofs, unless we explicitly require them, in which case they can be `unfold`ed.

Functions such as `gen_pages` are tail-recursive on the list of pages in the abstract state. This is useful when proving certain implementation functions, because it allows us to not have to reason by induction over the entire list of pages.

6.3.2 Erroneous operation

As mentioned in Section 5.4.1, functions of the abstract model may execute successfully (`imx_ok`), or they may fail (`imx_error`). We define the model such that the latter case would only happen if a function was called at the wrong time or with incorrect arguments. In other words, if the precondition is satisfied, the function will always execute successfully. To avoid the need to define the pre- and postconditions in the model as well, we map the `imx_error` result to an unsatisfiable VST heap predicate, namely `FF`. This effectively re-uses the VST precondition in the abstract model functions.

6.4 VST specifications

For each function of the implementation, we generate a VST specification using the state mapping function and the abstract functions.

```
Definition vst_spec_gc_free_os_memory : ident * funspec :=
  DECLARE I._gc_free_os_memory
  WITH
    shadow : ImxShadow.imx_shadow,
    state : ImxState.imx_state,
    gv : globals,
    id : nat
  PRE [tptr tvoid]
    PROP    (Map.imx_shadow_valid shadow state)
    PARAMS  (Map.vst_page id)
    GLOBALS (gv)
    SEP     (Map.imx_to_vst gv shadow state)
  POST [tvoid]
    Map.imx_to_vst_post
      gv
      shadow
      state
      (imx_func_gc_free_os_memory id).
```

Note that the spatial assertion part of the precondition (`SEP(...)` in VST syntax) consists only of the mapped abstract state. Likewise, the entire postcondition is generated using the helper function `imx_to_vst_post`.

We also define more straight-forward mapping functions to map arguments from one representation to another, such as the `vst_page` used above.

6.4.1 Axioms

Because the collector is a very low-level component of HashLink, it is possible to separate it from the codebase without too much work. On the other hand, the GC also relies on the operating system for page allocation. To avoid proving the internal implementation of any particular OS, the `mmap` and `munmap` functions are axiomatised. In particular, the page allocation assumes on the fact that `mmap` will produce pages with the proper memory alignment. This is not actually the case in practice, so the full GC implementation includes code to re-align as needed. In the proofs, the `mmap` axiom states that the memory is aligned and the re-alignment code is removed to simplify the proofs.

Similarly, calls to `malloc`, `calloc`, and `free` were axiomatised, although these axioms are already provided by VST.

6.5 VST proofs

Finally, we prove the VST functional specifications using VST forward separation logic. The proof scripts are available in the project archive. Each proof is rather long due to the nature of Coq-based proofs and idiomatic C code being generally difficult to reason about.

6.5.1 Verification setup

For proving the correctness of the implementation, we used VST-Floyd (commit 9f136a9a), CompCert (commit 35473ecc), Coq (8.11.1), and OCaml (4.05.0).

To be able to refer to the collector sources in our proofs, we invoke Clightgen, a tool that is bundled with CompCert. The output is an abstract syntax tree (AST) that represents the code in all of the defined functions, in addition to "identifiers", arbitrary numbers that uniquely identify each function or variable symbol. Note that Coq/VST compilation is not particularly fast, and including the AST module results in a small but visible slowdown.

The VST manual recommends structuring large program proofs such that each function is proven in an individual file that imports a base file that contains the VST specifications themselves. The base file then depends on the VST library, as well as the extracted AST.

After introducing the levels of abstraction listed in the previous sections, we decided on further separating our modules. We define the Immix abstract data model and functional specification in `Immix`. To avoid depending on the specific AST symbols, we define a module type `VSTIdents`[3] that lists all the required symbols as separate parameters. We define the model-to-VST mapping in `VSTMap`, and instantiate the VST specifications in `VSTSpecs`. In `VSTProofs` we finally provide the module with concrete symbol identifiers as defined in the extracted AST module `gc`. Each function is then proved individually in the `Proof*` modules. With this approach, we were able to reduce compilation times for modules that did not absolutely require the extracted AST. See Figure 6.2 for a visual representation of the module dependencies.

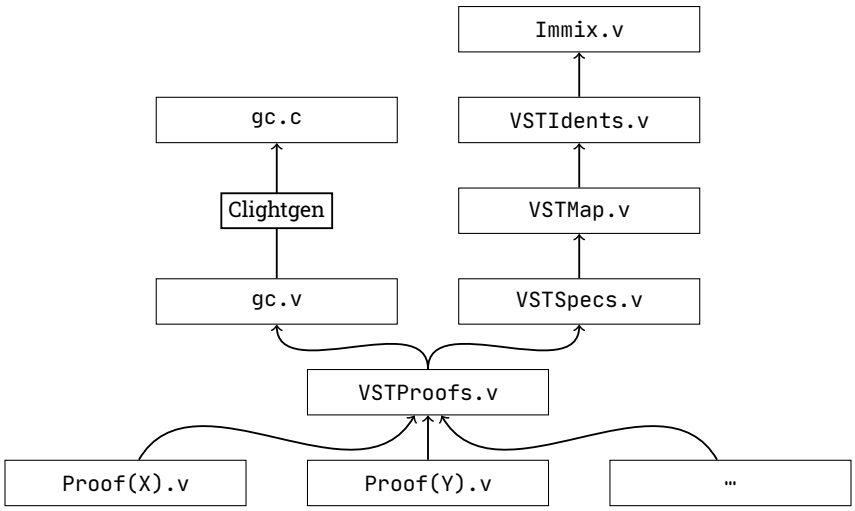

Figure 6.2: Dependencies between proof files/modules.

[3] A module type corresponds to an "interface" in standard OOP terminology. A parameter is a variable that can then be concretely defined in a separate module.

Chapter 7

Conclusions and future work

7.1 Contributions

To summarise the contributions of our project:

- We implemented an Immix-based garbage collector and demonstrated significant performance improvements of the HashLink virtual machine and runtime with its integration.
- We created a benchmarking framework for Haxe, allowing easy creation and contribution of new cases.
- We introduced a formal model of the collector written in Coq and proved that the code implements it correctly using VST-Floyd.
- We introduced a novel method to generate VST assertions for complex programs with a large state.

Since the conception of the project, improving HashLink (and by extension the Haxe ecosystem) in practice was an important goal. As such, after the conclusion of the academic portion of this project, the natural next step is to ensure the collector code is contributed back into the master branch of the HashLink repository, allowing users to test it. The pull request implementing the new collector is accessible online in the HashLink repository[1].

7.2 Performance improvements

Some optimisations that were explored, but not ultimately implemented, were discussed in Section 3.4. Further performance improvements may be possible by introducing a generational collection system, making the marking and sweeping

[1] `https://github.com/HaxeFoundation/hashlink/pull/372`

multi-threaded, and optimising the codebase for generating better assembly code with various compilers.

A fully concurrent collector could be very beneficial for the performance as well, although this would require changes to the GC interface, which was left unmodified for this project by design. Changing the GC interface would also require potentially complex changes to the code produced by HashLink's JIT compiler, e.g. to introduce write barriers.

7.3 Formal verification

It may also be possible to specify the behaviour of the HashLink virtual machine as a whole, to establish the correctness of its implementation, its VM semantics, and to allow its integration in more in-depth research projects. This would be a much larger undertaking, due to its larger codebase, various low-level interactions with the operating system, and its own JIT compiler. Proof of the latter might necessarily be built on top of proofs for the semantics of the x86 architecture. A similar work, towards proving a Java VM, has been conducted by Hanbing [40], and could be a good starting point.

The framework of VST-Floyd itself could also be improved, in terms of its documentation and tooling, since we did not find the development of proofs for the collector to be particularly intuitive or user-friendly. Some possible improvements include a more comprehensive and up-to-date manual, a guide to the internals of the framework, as well as general performance improvements.

List of Figures

List of Tables

Bibliography

[1] Cannasse N, contributors. The Haxe Programming Language; 2006–2020. Accessed: 2020-06-07. Available from: https://www.haxe.org/.

[2] Cannasse N, contributors. HashLink; 2015–2020. Accessed: 2020-06-07. Available from: https://hashlink.haxe.org/.

[3] Blackburn S, McKinley KS. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In: Gupta R, Amarasinghe SP, editors. ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices 43(6). Tucson, AZ: ACM Press; 2008. p. 22–32.

[4] Alpern B, Augart S, Blackburn SM, Butrico M, Cocchi A, Cheng P, et al. The Jikes Research Virtual Machine project: Building an open-source research community. IBM Systems Journal. 2005;44(2):399–417.

[5] Coq development team. The Coq proof assistant reference manual; 2020. Accessed: 2020-06-07. Available from: https://coq.inria.fr/.

[6] Cao Q, Beringer L, Gruetter S, Dodds J, Appel AW. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. Journal of Automated Reasoning. 2018;.

[7] Leroy X, Blazy S, Kästner D, Schommer B, Pister M, Ferdinand C. CompCert - A Formally Verified Optimizing Compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. Toulouse, France: SEE; 2016. Available from: https://hal.inria.fr/hal-01238879.

[8] The GCC Team. The GNU Compiler Collection;. Accessed: 2020-06-10. Available from: https://gcc.gnu.org/.

[9] The LLVM Project. Clang C language family frontend for LLVM;. Accessed: 2020-06-10. Available from: https://clang.llvm.org/.

[10] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer Manuals; 2019. Accessed: 2020-06-07. Available from: https://software.intel.com/en-us/articles/intel-sdm.

[11] Butters AM. Total Cost of Ownership: A Comparison of C/C++ and Java. Evans Data Corporation; 2007. Available from: http://docplayer.net/24861428-Total-cost-of-ownership-a-comparison-of-c-c-and-java.html.

[12] Jones R, Hosking A, Moss E. The Garbage Collection Handbook: The Art of Automatic Memory Management. CRC Applied Algorithms and Data Structures. Chapman & Hall; 2012.

[13] Shahriyar R, Blackburn SM, Yang X, McKinley KS. Taking off the Gloves with Reference Counting Immix. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. Indianapolis, IN: ACM Press; 2013. p. 93–110.

[14] McCarthy J. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. Communications of the ACM. 1960 Apr;3(4):184–195.

[15] Saunders RA. The LISP System for the Q–32 Computer. In: Berkeley EC, Bobrow DG, editors. The Programming Language LISP: Its Operation and Applications. Cambridge, MA: Information International, Inc.; 1974. p. 220–231. Available from: http://www.softwarepreservation.org/projects/LISP/book/III_LispBook_Apr66.pdf.

[16] Jonkers HBM. A Fast Garbage Compaction Algorithm. Information Processing Letters. 1979 Jul;9(1):26–30.

[17] Fenichel RR, Yochelson JC. A Lisp Garbage Collector for Virtual Memory Computer Systems. Communications of the ACM. 1969 Nov;12(11):611–612.

[18] Jones R, Ryder C. A Study of Java Object Demographics. In: Jones R, Blackburn S, editors. 7th ACM SIGPLAN International Symposium on Memory Management. Tucson, AZ: ACM Press; 2008. p. 121–130.

[19] Lieberman H, Hewitt CE. A Real-Time Garbage Collector Based on the Lifetimes of Objects. Communications of the ACM. 1983 Jun;26(6):419–429. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981. Available from: http://lieber.www.media.mit.edu/people/lieber/Lieberary/GC/Realtime/Realtime.html.

[20] Flood CH, Kennke R, Dinn A, Haley A, Westrelin R. Shenandoah: An Open-source Concurrent Compacting Garbage Collector for OpenJDK. In: ACM International Symposium on Principles and Practice of Programming in Java. Lugano, Switzerland: ACM; 2016. p. 13:1–13:9.

[21] Shahriyar R, Blackburn SM, McKinley KS. Fast Conservative Garbage Collection. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. Portland, OR: ACM Press; 2014. p. 121–139.

[22] Ueno K, Ohori A, Otomo T. An Efficient Non-Moving Garbage Collector for Functional Languages. In: 16th ACM SIGPLAN International Conference on Functional Programming. Tokyo, Japan: ACM Press; 2011. p. 196–208.

[23] Blackburn SM, Garner R, Hoffman C, Khan AM, McKinley KS, Bentzur R, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM SIGPLAN Notices 41(10). Portland, OR: ACM Press; 2006. p. 169–190.

[24] Standard Performance Evaluation Corporation. SPEC JVM2008 benchmarks; 2008. Accessed: 2020-06-07. Available from: http://www.spec.org/jvm2008.

[25] Zorn BG. Comparative Performance Evaluation of Garbage Collection Algorithms. University of California, Berkeley; 1989. Technical Report UCB/CSD 89/544. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/CSD-89-544.pdf.

[26] Wilson PR, Johnstone MS, Neely M, Boles D. Memory Allocation Policies Reconsidered; 1995. Unpublished manuscript. Accessed: 2020-06-12. Available from: ftp://ftp.cs.utexas.edu/pub/garbage/submit/PUT_IT_HERE/frag.ps.

[27] Ricci NP, Guyer SZ, Moss JEB. Elephant Tracks: Portable Production of Complete and Precise GC Traces. In: Petrank E, Cheng P, editors. 12th ACM SIGPLAN International Symposium on Memory Management. Seattle, WA: ACM Press; 2013. .

[28] Lengauer P, Mössenböck H. 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 (ICPE14). Dublin, Ireland: ACM Press; 2014. p. 111–122.

[29] Jacek N, Chiu MC, Marlin B, Moss E. Optimal Choice of When to Garbage Collect. ACM Transactions on Programming Languages and Systems. 2019 Jan;41(1):3:1–3:35.

[30] Paulson LC. Natural Deduction as Higher-Order Resolution. Computing Research Repository. 1993;cs.LO/9301104. Available from: https://arxiv.org/abs/cs/9301104.

[31] Koh N, Li Y, Li Y, yao Xia L, Beringer L, Honore W, et al. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. ACM SIGPLAN International Conference on Certified Programs and Proofs. 2019 Jan;Available from: https://www.cis.upenn.edu/~bcpierce/papers/deepweb-overview.pdf.

[32] Hawblitzel C, Petrank E. Automated Verification of Practical Garbage Collectors. In: 36th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. Savannah, GA: ACM Press; 2009. p. 441–453.

[33] Barnett M, Chang BYE, DeLine R, Jacobs B, Leino KRM. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer FS, Bonsangue MM, Graf S, de Roever WP, editors. Formal Methods for Components and Objects. Berlin, Heidelberg: Springer Berlin Heidelberg; 2006. p. 364–387.

[34] de Moura L, Bjørner N. Z3: An Efficient SMT Solver. In: Ramakrishnan CR, Rehof J, editors. Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer Berlin Heidelberg; 2008. p. 337–340.

[35] Ericsson AS, Myreen MO, Pohjola JÅ. A verified generational garbage collector for CakeML. Journal of Automated Reasoning. 2019;63(2):463–488.

[36] Ungar DM, Jackson F. Tenuring Policies for Generation-Based Storage Reclamation. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM SIGPLAN Notices 23(11). San Diego, CA: ACM Press; 1988. p. 1–17.

[37] Boehm HJ, Weiser M. Garbage Collection in an Uncooperative Environment. Software: Practice and Experience. 1988;18(9):807–820. Available from: `http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html`.

[38] Hosking AL, Moss JEB, Stefanović D. A Comparative Performance Evaluation of Write Barrier Implementations. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM SIGPLAN Notices 27(10). Vancouver, Canada: ACM Press; 1992. p. 92–109. Available from: `ftp://ftp.cs.umass.edu/pub/osl/papers/oopsla92.ps.Z`.

[39] Hex-Rays SA. IDA Pro;. Accessed: 2020-06-10. Available from: `https://www.hex-rays.com/products/ida/`.

[40] Hanbing L. Formal specification and verification of a JVM and its bytecode verifier. University of Texas at Austin; 2006. Available from: `https://repositories.lib.utexas.edu/handle/2152/2763`.

Appendix A

Installation guide

At the time of writing, the new collector is only usable with 64-bit architectures, and was tested on Mac OS X 10.9.5 and Ubuntu 18.04 only. Compilation fails entirely on Windows, since the underlying OS calls assume a POSIX environment. As noted in Section 3.6 and Chapter 7, re-aligning the codebase with the framework of the original HashLink project is subject to upcoming future work.

To install HashLink with the new collector, check out the `feature/gc` fork branch of the `Aurel300` fork of HashLink[1]. In a standard `git`-equipped terminal:

```
$ git clone --single-branch --branch feature/gc
    git@github.com:Aurel300/hashlink.git
```

Alternatively, if SSH login for `git` is not set up:

```
$ git clone --single-branch --branch feature/gc
    https://github.com/Aurel300/hashlink.git
```

Then follow the installation guide found in the README file[2]. The new collector does not add any additional dependencies, so `apt-get` on Linux and `brew` on Mac OS X should suffice before invoking `make`.

[1] `https://github.com/Aurel300/hashlink/tree/feature/gc`
[2] `https://github.com/HaxeFoundation/hashlink/blob/master/README.md`

Appendix B

Tabulated benchmark results

This appendix lists the specific benchmark measurements used to produce the graphs in Chapter 4.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	8.491	9.395	3.075	2.820
2	8.467	9.380	3.068	2.904
3	9.180	9.391	3.086	2.801
4	10.626	9.511	3.094	2.852
5	9.392	9.761	3.027	2.796
6	8.858	9.469	3.057	2.898
7	9.009	9.510	2.659	2.841
8	8.885	9.433	3.050	2.850
9	8.913	9.606	3.058	2.822
10	9.086	9.556	3.087	2.862
11	9.098	9.592	2.938	2.821
12	9.209	9.874	3.038	2.843
13	9.018	9.960	1.457	2.865
14	9.099	10.038	2.918	2.850
15	9.143	9.875	2.982	2.808
16	9.045	10.117	3.001	2.870
17	9.197	9.902	2.995	2.865
18	9.198	10.201	1.376	2.883
19	9.365	9.956	2.892	2.858
20	9.457	10.005	2.824	2.838

`alloc` benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	16.298	4.930	19.497	4.845
2	17.355	4.883	19.351	4.783
3	17.795	4.901	19.492	4.798
4	17.959	4.880	19.732	4.743
5	17.982	4.894	19.541	4.771
6	18.263	5.100	19.471	4.981
7	18.292	4.895	19.500	4.813
8	18.291	4.889	19.444	4.737
9	18.371	4.909	19.440	4.823
10	18.514	4.906	19.454	4.752
11	18.447	4.869	19.517	4.793
12	18.656	4.907	19.470	4.765
13	18.675	5.146	19.513	5.016
14	18.722	4.921	19.581	4.775
15	19.314	4.887	19.693	4.820
16	19.505	4.894	19.629	4.776
17	19.670	4.906	19.630	4.782
18	19.620	4.982	19.638	4.788
19	19.550	4.908	19.531	4.795
20	19.563	5.142	19.535	5.022

`bcrypt` benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	46.492	55.416	4.288	3.110
2	50.826	55.457	4.176	3.180
3	52.080	55.407	4.155	3.401
4	56.942	55.585	4.175	3.191
5	56.543	55.430	4.192	3.141
6	55.638	55.658	4.184	3.119
7	55.841	56.155	4.323	3.145
8	55.784	55.917	4.217	3.137
9	56.180	56.405	4.078	3.129
10	55.936	55.937	4.065	3.158
11	55.989	56.468	4.089	3.189
12	56.194	55.640	4.080	3.144
13	56.288	55.634	4.105	3.406
14	56.242	55.842	4.079	3.242
15	56.294	55.945	4.241	3.113
16	56.416	56.066	4.079	3.127
17	56.180	55.566	4.015	3.146
18	56.437	56.433	4.021	3.174
19	56.446	55.879	4.087	3.110
20	56.203	56.498	3.990	3.196

`binarytrees` benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	76.767	73.560	14.913	11.826
2	89.483	66.769	17.273	13.661
3	n/a	n/a	18.128	18.128
4	n/a	n/a	17.192	13.665
5	n/a	n/a	18.816	14.246
6	n/a	n/a	17.162	15.007
7	n/a	n/a	17.685	13.225
8	n/a	n/a	18.931	13.497
9	n/a	n/a	18.095	13.377
10	n/a	n/a	17.431	22.140
11	n/a	n/a	n/a	n/a
12	n/a	n/a	n/a	n/a
13	n/a	n/a	n/a	n/a
14	n/a	n/a	n/a	n/a
15	n/a	n/a	n/a	n/a
16	n/a	n/a	n/a	n/a
17	n/a	n/a	n/a	n/a
18	n/a	n/a	n/a	n/a
19	n/a	n/a	n/a	n/a
20	n/a	n/a	n/a	n/a

`formatter` benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	129.448	98.689	24.804	21.167
2	97.689	n/a	23.973	n/a
3	n/a	n/a	24.430	n/a
4	n/a	n/a	n/a	n/a
5	n/a	n/a	n/a	n/a
6	n/a	n/a	n/a	n/a
7	n/a	n/a	n/a	n/a
8	n/a	n/a	n/a	n/a
9	n/a	n/a	n/a	n/a
10	n/a	n/a	n/a	n/a
11	n/a	n/a	n/a	n/a
12	n/a	n/a	n/a	n/a
13	n/a	n/a	n/a	n/a
14	n/a	n/a	n/a	n/a
15	n/a	n/a	n/a	n/a
16	n/a	n/a	n/a	n/a
17	n/a	n/a	n/a	n/a
18	n/a	n/a	n/a	n/a
19	n/a	n/a	n/a	n/a
20	n/a	n/a	n/a	n/a

`formatter-noio` benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	9.411	8.271	2.185	1.616
2	9.656	8.568	2.212	1.684
3	11.964	8.323	2.287	1.616
4	9.481	9.099	2.233	1.679
5	9.133	8.452	2.233	1.689
6	8.872	8.529	2.233	1.649
7	8.840	8.491	2.246	1.973
8	9.149	8.688	1.219	1.762
9	8.723	8.382	2.280	1.710
10	8.910	8.436	2.295	1.736
11	8.903	8.385	2.290	1.701
12	9.033	8.677	2.325	1.708
13	8.922	8.415	2.259	1.708
14	8.891	8.485	2.549	1.741
15	9.040	8.429	2.314	1.727
16	9.170	8.778	2.249	1.713
17	9.039	8.515	2.310	1.732
18	9.107	8.481	2.277	1.694
19	9.184	8.426	2.293	1.756
20	9.205	8.853	2.360	1.717

`json` benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	203.976	211.670	18.025	14.416
2	202.291	229.902	17.983	14.668
3	202.201	206.827	18.077	14.368
4	202.632	203.729	17.904	14.506
5	202.547	202.633	18.072	14.685
6	201.471	203.670	17.884	14.421
7	201.174	201.836	18.125	14.339
8	201.325	200.001	18.106	14.740
9	201.884	199.342	18.027	14.347
10	201.465	199.777	18.131	14.388
11	201.698	198.790	18.095	14.785
12	201.455	198.632	18.090	14.435
13	201.639	200.329	17.944	14.431
14	201.233	202.149	18.082	14.852
15	201.740	203.308	18.120	14.414
16	202.648	204.262	18.131	14.455
17	203.789	205.457	18.029	14.736
18	203.819	206.568	18.122	14.434
19	204.724	207.202	18.136	14.430
20	205.643	208.473	18.132	14.839

`mandelbrot` benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	162.893	185.824	13.805	11.052
2	175.236	186.193	13.695	10.923
3	188.679	184.493	13.661	10.938
4	188.589	185.763	13.404	11.069
5	189.289	183.744	13.366	10.964
6	189.670	184.822	13.359	10.922
7	190.506	183.431	13.275	11.019
8	191.198	184.201	13.269	10.937
9	195.735	182.706	13.391	10.931
10	190.895	184.971	13.355	11.065
11	191.460	186.470	13.360	10.943
12	192.352	186.229	13.546	10.913
13	192.007	182.728	13.452	11.034
14	192.584	186.460	13.499	10.899
15	191.692	182.493	13.590	10.999
16	194.799	185.401	13.404	11.011
17	191.438	184.253	13.441	10.904
18	190.953	187.391	13.669	10.905
19	191.028	184.062	13.470	11.080
20	190.435	183.242	13.564	10.903

`mandelbrot-anon` benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	5.253	2.277	5.985	2.233
2	5.266	2.160	5.824	2.256
3	5.267	2.180	6.007	2.234
4	5.246	2.163	6.012	2.210
5	5.391	2.191	5.840	2.241
6	5.287	2.135	5.816	2.204
7	5.281	2.161	5.986	2.262
8	5.669	2.244	5.880	2.212
9	6.956	2.133	5.937	2.177
10	7.236	2.199	5.783	2.416
11	7.522	2.215	5.832	2.291
12	6.335	2.148	5.924	2.285
13	6.186	2.119	6.039	2.249
14	5.826	2.148	5.943	2.315
15	5.584	2.132	5.915	2.301
16	5.613	2.254	5.923	2.319
17	5.670	2.222	5.946	2.296
18	5.960	2.222	6.057	2.341
19	5.630	2.167	6.232	2.318
20	5.703	2.184	5.895	2.356

nbody benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	4.235	2.663	2.228	0.642
2	4.454	2.593	2.067	0.631
3	4.101	2.645	2.069	0.627
4	4.112	2.553	2.113	0.602
5	4.148	2.650	2.033	0.598
6	4.122	2.617	2.060	0.609
7	4.144	2.622	2.111	0.605
8	4.138	2.962	2.031	0.620
9	4.193	2.595	2.092	0.626
10	4.437	2.590	2.035	0.633
11	4.136	2.653	2.099	0.600
12	4.152	2.605	2.098	0.610
13	4.115	2.647	2.084	0.607
14	4.176	2.620	2.108	0.606
15	4.092	2.603	2.328	0.621
16	4.085	2.580	2.079	0.624
17	4.097	2.702	2.042	0.624
18	4.339	2.578	2.135	0.627
19	4.038	2.668	2.050	0.611
20	4.074	2.818	2.037	0.601

sha256 benchmark results.

Sample	JIT old (s)	C old (s)	JIT new (s)	C new (s)
1	18.814	20.309	2.736	2.182
2	18.674	20.005	2.664	2.270
3	19.171	20.285	2.676	2.090
4	19.357	20.675	2.676	2.162
5	19.478	20.256	2.648	2.155
6	20.824	20.443	2.646	2.147
7	19.611	22.681	2.656	2.164
8	19.764	21.624	2.642	2.198
9	19.797	25.316	2.673	2.163
10	19.808	22.873	2.672	2.119
11	20.088	20.259	2.783	2.175
12	19.911	20.399	2.667	2.170
13	20.061	20.369	2.717	2.144
14	20.276	20.254	2.631	2.417
15	20.334	20.229	2.644	2.231
16	20.436	20.167	2.620	2.366
17	20.753	20.140	2.681	2.169
18	20.475	20.252	2.629	2.199
19	21.204	20.123	2.654	2.185
20	20.570	25.980	2.646	2.182

sha512 benchmark results.