
# Feasibility of an Integrated Hardware Garbage Collector

**Ed Nutting**

**May 2017**

**Project thesis submitted in support of the Degree of Bachelor of Engineering in Computer Science and Electronics**

**Department of Electrical & Electronic Engineering**
**University of Bristol**

## DECLARATION AND DISCLAIMER

Unless otherwise acknowledged, the content of this thesis is the original work of the author. None of the work in this thesis has been submitted by the author in support of an application for another degree or qualification at this or any other university or institute of learning.

The views in this document are those of the author and do not in any way represent those of the University.

The author confirms that the printed copy and electronic version of this thesis are identical.

Signed:

Dated:

# *Abstract*

The majority of modern programming languages are dependent on Garbage Collection (GC) for memory management. Traditionally, GC has been implemented in software but it comes with a significant number of drawbacks. Even recent advances in generational, concurrent and hardware-assisted GC have failed to create a viable system for embedded and real-time devices. Furthermore, GC provides a trustworthy execution environment by preventing memory errors such as buffer overflows, *nil* pointer access, accessing freed memory, failing to free memory, repeat freeing memory, etc. Enforcing these rules in software without hardware support is difficult and costly, in performance, code size and development time. A new design for GC that simplifies, reduces cost and provides real-time performance is required.

This thesis presents a new design for hardware GC that is directly integrated with the CPU (the IHGC), effectively replacing a traditional MMU. The commercial feasibility of the design is tested against 51 criteria and is shown to be viable. This thesis demonstrates that the IHGC is both realisable with current hardware technologies and only requires minimal, achievable, software changes. Further to this, the IHGC's performance is shown to be sufficient for real-time applications and the synthesised design is small enough for embedded devices. The IHGC is faster than the simplest, non-garbage-collecting equivalent *malloc/free* routines written in C. This thesis concludes that the IHGC is a significant step forward for GC design and has wide scope for promising future work.

To

Barry Skeggs

Loving Grandfather

# *Acknowledgements*

This thesis would not have even begun without the innovative design created by Professor David May. I am most grateful to him for sharing his ideas and wealth of experience over the two years leading to the start of this thesis. I look forward to continuing our work together over the next few years.

I am also grateful to my parents, who have provided so much support and unwavering confidence in me. To my father especially, I am thankful for having guided and taught me. In particular the principles of accurate, detailed work and the motivation to stay dedicated to challenging, at times frustrating, projects, are two of the most important lessons any son could have hoped to learn.

Lastly, I would like to thank Caroline Higgins, Naim Dahnoun and Richard Grafton, who have supported me and who have added so much to my time at university.

# *Table of Contents*

# *List of Tables*

# *List of Figures*

**Figure** **Page**

Chapter One

# *Introduction*

Garbage collection (GC) provides safely managed memory for a system, which solves otherwise insoluble problems. Automatically managed memory reduces code clutter, complexity and number of bugs and so reduces development and maintenance time, cost and effort. GC is the best known method of memory management and so dramatically improves security and reliability of software. A GC-based system achieves this by ensuring memory is automatically freed, pointers are never out of bounds, and that pointers cannot form without allocating memory or offsetting an existing pointer. This safe environment protects the programmer from common, often hard-to-find issues such as memory leaks and buffer overflows. Consequently, GC has become an integral part of widely used "managed" programming languages.

The majority of systems implement GC in software, which comes at a cost and cannot enforce the memory model across the entire system. Software GC leads to unavoidable barrier and stop-the-world conditions which prevent threads from progressing. It also requires a significant amount of processing time. Due to these limitations, software GC and thus managed languages are not widely used in areas such as high performance computing (HPC), embedded, or real-time systems.

Unlike with software garbage collectors, it is possible to design GC in hardware which has complete knowledge of the system at all times. This tackles both the traditional and the previously insoluble problems of software GC design. For example, hardware controls all memory operations so can avoid stop-the-world and barrier conditions, and check bounds on all uses of pointers. The overhead of allocation instructions is also reduced from hundreds to one.

In April 2014, the Heartbleed OpenSSL security flaw caused international panic and is estimated to have cost hundreds of millions of dollars. Attackers obtained data from any system that enabled OpenSSL Heartbeat Extension Packets. The packet handling code failed to check buffer boundaries or to zero-out buffers. Similarly, in February 2017, the Cloudbleed buffer overrun bug allowed attackers to obtain sensitive data. In both cases a managed memory system would have caught the

bugs.

In today's increasingly security conscious world, hardware GC could bring managed memory to a wide range of devices thereby decreasing software development time, saving power, increasing performance and preventing many security flaws and bugs.

## 1.1 Research Aims

The research presented in this thesis aims to demonstrate the feasibility of an integrated hardware garbage collector (IHGC) by answering two overarching questions: Can an IHGC system be created in a reasonable quantity of hardware? Will it have sufficient performance to run real programs? These are answered by evaluating the system against chosen criteria.

The criteria presented fit current commercial-grade expectations for performance, complexity and physical size of micro-controller processors. Application-scale processors present additional challenges, such as pipelining, which will require further investigative work. The criteria cover: the technical challenge of creating the hardware, integration with an existing processor, quantity of required hardware, type and scale of system-level optimisations, performance characteristics of the GC, and ability to execute real programs.

This thesis develops a processor in Verilog synthesised to an FPGA based on the increasingly popular, open-source RISC-V Instruction Set Architecture (ISA). It is then demonstrated that the design meets the feasibility criteria through a series of test programs written in assembly and C. It also suggests that the performance shown combined with future work would enable use in HPC systems. With the increasing use of Python, Java and other managed programming languages for embedded, real-time, low-power and HPC software, the system developed here could make a significant contribution to a growing area of academic and commercial, research and development.

This research does not aim to verify the IHGC or the processor. However, tight upper and lower bounds at the mico-architectural level of the GC operations are presented. Verification and precise characterisation of an IHGC (by using other main memory and directory sizes) will form the basis of future research.

## 1.2 Thesis Outline

Chapter 2 provides background to the large topic of garbage collection and Chapter 3 reviews prior work in hardware implementations. It is common to find the concept of hardware GC dismissed as too complex or costly. Chapter 4 dispels these myths by presenting the Integrated Hardware Garbage Collector design used in this research.

The feasibility criteria for a commercially-viable, embedded-scale IHGC implementation are prsented in Chapter 5. Chapter 6 explains the approach to implementation of the IHGC and discusses various problems encountered. Surprisingly, no major obstructions were encountered, given the IHGC design has not been verified. Some system level optimisations were required (Chapter 7) but they are routine; cutting edge hardware design has used such optimisations for over two decades.

With kind permission from Professor David May (University of Bristol), his design for the IHGC is presented in Appendix A. It has not yet been submitted for publication elsewhere and so cannot otherwise be referenced. The Verilog implementation, provided with this thesis, and the explanation of both the design (Chapter 4) and the implementation (Chapter 6), are the original work of the author of this thesis.

Chapters 8 through 10 evaluate the implementation by comparing test results to both the feasibility criteria and results of best-case equivalent programs. A secondary outcome is the demonstration that realistic programs can compile to the new architecture with minimal change. Examples of adapted programs and their performance measurements are provided in Chapter 10.

This thesis concludes in Chapter 11 that an IHGC is feasible and has a high chance of significantly improving performance, energy efficiency and trustworthiness of modern processors. Chapter 12 presents ideas for future work, including formal verification of the design, optimisations, extended features, and proper error handling mechanisms within the CPU.

Chapter Two

# *Background*

Garbage collection is the subject of thousands of papers, books and codebases, so it is difficult to gain a comprehensive perspective of the topic. As an introduction, the 2016 edition of the book by Richard Jones et al. is highly recommended.[16]

When considering any GC system, the important aspects are: the algorithm, the application, hardware and software environments, and any proven advantages or disadvantages. The question of hardware versus software is important since software problems exist that may not exist in hardware.

## 2.1 Garbage collection algorithms

There are three classical GC algorithms: Reference-counting, Mark-Sweep, and Copying (largely unused nowadays for space efficiency reasons). Mark-Compact is slightly more recent, is used by the IHGC (Chapter 4) and neatly solves two key problems that other algorithms do not: fragmentation and reference cycles.

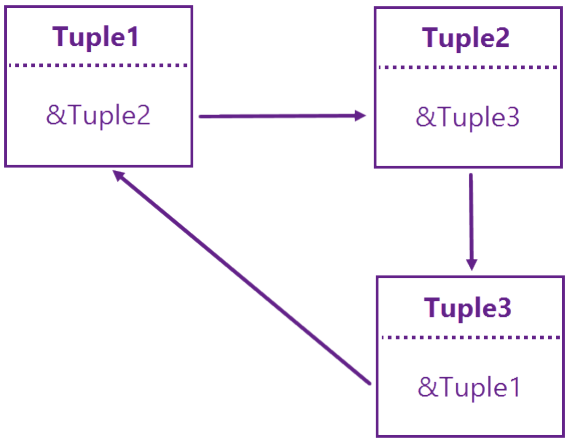



Figure 2.1: Reference cycles between tuples

Fragmentation is small gaps between "live" memory (memory which is still in use) created when memory is swept ("freed"). Compaction copies live memory to a contiguous region thus eliminating fragmentation. Reference counting cannot detect reference cycles (Figure 2.1), consequently "marking" GC is necessary in many applications.

Some Mark-Compact implementations are cache-aware and/or generational. Generational GC reduces cycle counts and times by minimising the traced heap area. Older objects are assumed to be live and so are marked/compacted less often. Older objects are typically larger too so generational GC can reduce the size of memory processed in each cycle.

Lastly, some GC algorithms are distributable meaning the mark, sweep or compact stages can execute simultaneously on multiple logical processors. In practice, these have been used for "cloud" servers with large amounts of shared memory.

## 2.2 Problems of software GC

Software Garbage Collection has five significant problems, Stop the world (STW) conditions, necessity of read/write barriers and overheads of memory, processing time and cache invalidation.

STW occurs when the GC has to pause the mutator (the program using the GC) to ensure consistency. Extensive research has developed algorithms to avoid large pause times and some fully concurrent GC algorithms have been created which avoid all STW conditions, but these require hardware R/W Barriers (see Chapter 3).

Hardware barriers delay access to memory which is being manipulated, causing short pauses to the mutator. Real-time performance of hardware-assisted GC has been achieved by tightly bounding such delays.

Software GC consumes CPU time to execute the collection cycles and requires instructions in the program to interface with GC software. Furthermore, GC threads interrupting the program cause cache eviction and loss of out-of-order execution state. The result is bloated program size and significantly reduced performance, especially on highly pipelined processors.

Money and power are the solutions thrown at applications-scale processors but neither are acceptable for micro-controller or embedded scales. Any GC implementation also requires memory for "dead" tuples awaiting collection and tracking object states.

## 2.3 Usage and popularity of GC

GC is widely used as it provides a highly reliable memory system, protecting programmers from common errors. Bugs are also detected at runtime and for some highly modular software, no alternative memory management is available. Managed memory reduces development time and effort while increasing reliability and security. Code readability also usually increases, further reducing development cost. GC underpins so many modern languages that it is almost unavoidable. It is surprising, therefore, that there is so little hardware support.

## 2.4 Capabilities and limitations of hardware

Hardware has capabilities which are impossible for software. For example, hardware always knows the complete state of a system. This is true even for modern C/C++ programs. The dangers of arbitrarily converting between numbers and pointers have been learned and most modern software - certainly good, reliable, secure software - carefully distinguishes between them. The memory model of modern C/C++ is similar to that of C# or Java, so "conservative" GC is no longer necessary.

Hardware implements algorithms more efficiently and thus faster than software. There is greater scope for clock-cycle-level optimisation and tight integration with the system to maximise throughput and minimise overheads. For example, compact state machines can be synthesised that execute in parallel with the main processor, without adverse software effects (see Chapter 4).

Production hardware is unmodifiable (and FPGAs have write-once techniques) but can be strongly or formally verified. This means a reliable, secure, trustworthy memory model can be enforced across the system. This is far more trustworthy than any likely software environment.

However, more hardware means higher power consumption, greater cost and potential clock speed limitations. In the embedded processor market, size, cost and power consumption are the deciding factors between processors. Implementing and verifying complex hardware is difficult and mistakes cannot be corrected after final tape-out. Furthermore, hardware operating in a fixed mode may not be flexible enough so hardware algorithms must be carefully and compactly designed and implemented. These reasons may suggest why hardware GC is uncommon.

Lastly, it is reasonable to expect that hardware changes will necessitate software redesign. This thesis postulates three kinds of changes:

**Re-write** Large parts of software are re-written to fit the new hardware operational model.

**Re-compilation** Only the compiler needs modification, with minor changes to standard libraries. Most software only needs tweaking and re-compilation.

**Cut-down** Large parts of software need modifying to delete superfluous code and to take advantage of the new hardware. Re-compilation is necessary.

"Cut-down" is considered distinct from "re-write" as less development effort is required. If the software and hardware operational models match, code can be reduced to hardware calls (instructions). This is simpler than rewriting to update the entire software design to match the model used by the new hardware. Chapter 10 shows that only software cut-down is required as the IHGC uses the same operational model as existing C/C++ software.

## 2.5 Evaluating GC Implementations

Evaluating a GC implementation is problematic as isolating any variable of performance is impossible. However, appropriate metrics are useful assessing system and GC performance. The criteria for feasibility, Chapter 5, utilise some of the common metrics listed below.

**Collection time** Average and upper/lower bounds of collection cycle time.

**Allocation time** Average and upper/lower bounds of time for memory allocations. (Careful comparison of allocation times is required as some implementations only allow fixed-sized allocations. The IHGC allows variable sized allocations, including zero size.)

**Access time** Average and upper/lower bounds of time to access valid memory. It is sometimes acceptable to ignore invalid accesses.

**Total execution time** Average time required for a given program to execute. There are only two GC benchmark suites (Jikes RVM and Dacapo, no longer maintained) both for Java.[1, 4]

**Heap size** Maximum size of heap the GC can manage.

**Handle count** Maximum number of tuples the GC can manage.

**Dead memory size** Average and maximum amounts of dead memory overhead.

**GC memory size** Size of memory the GC requires, e.g. for physical addresses of tuples.

**Algorithmic complexity** No single measure exists; this is not a good performance indicator.

**Implementation complexity** Similar to Algorithmic complexity but can indicate development cost.

**System-wide effects** e.g. on caches, out-of-order execution state, memory bus activity, other programs, physical memory degradation over time, ...

**Capability** General or special purpose capability e.g. only usable for JVM or usable for any managed memory system, e.g. Reference counting vs full Mark-Sweep

Chapter Three

# *Prior Work*

The IHGC design is so new that there are only two comparable papers (of which one claims to be the first on this topic), which are discussed at the end of this chapter. The rest of this chapter provides a comprehensive summary of all relevant publications from the last few decades. There is incredibly little research into the topic of hardware GC, which is probably because most hardware engineers have no idea what GC is, and most software engineers invariably prefer software solutions. It takes a rare breed of engineer to even consider bridging the divide. Nevertheless, hardware GC has featured occasionally over the last 30 years and a recent increase in papers suggests a chance that hardware GC might take off.

Past research emphasised three things: Java, Real-time and Embedded (e.g. Ive 2003).[14] This thesis is no different except that C, C# and Haskell (and other languages) are considered equally important and worthy of support. Software GC is not perfect but is good enough for applications-scale use, but in real-time and embedded areas it has never yet proven adequate. Another increasingly important consideration is energy efficiency, where some research suggests adding more hardware, to replace complex or frequently used software, can reduce overall energy consumption (e.g. Cao et al. 2012).[5]

## 3.1 Theory

There is a lot of theory surrounding GC mostly dealing with two areas: algorithms and software concurrency. The world has settled on the algorithms mentioned in Chapter 2 but research continues to find minor improvements. More recently research has focused on concurrent operation with the mutator with some success. No references are provided here as concurrent software GC differs significantly from concurrent (true parallel) hardware GC.

One paper from 1995, however, considered whether hardware was required for object-oriented programming.[13] Interestingly, it concluded that hardware support had little benefit over contem-

porary heavily optimising compilers. Furthermore, they found that cache hierarchies had a bigger impact on performance. However, a lot has changed in 20 years. Object oriented languages are far more prevalent, more complex with higher performance demands and, crucially, we have reached the limits of current cache memory structures.

## 3.2 Commercial Solutions

There is currently one commercial solution for hardware assisted GC. The C4: Concurrent Garbage Collector created by Azul Systems is marketed as the only fully concurrent GC that doesn't stop the world. It is a software, continuously-concurrent GC with hardware assistance from a "Loaded Value Barrier", targeted at the JVM. It has been integrated with X86 hardware and has a maximum heap size of 670GB. This system is not suitable for embedded or real-time systems and Azul (citing "practical engineering complexity reasons") have not yet been able to implement the complete algorithm and so their implementation does stop-the-world for short periods. This thesis makes practical, full implementation and no STW conditions an explicit target of the IHGC design.

## 3.3 Hardware Assisted GC

Hardware assistance exists in four forms: memory mechanisms, co-processors, ISA extensions, and profiling. Memory mechanisms typically either add useful features for software (e.g. read/write barriers) or implement part of the GC algorithm in hardware. Co-processors are general or special purpose processors connected via the system bus to the main processor to handle some or all GC operations. ISA extensions are suites of new instructions for micro-managing memory, cache and processes. Profiling hardware provides information to facilitate dynamic optimisation of software GC.

### 3.3.1 Memory Assistance

Higuera et al. considered using hardware read/write barriers to handle the copying/compacting stage of a concurrent GC algorith, to ensure that the mutator has a consistent view of memory words while the GC moves them.[12] Although this can be effective for performance, it only addresses the responsiveness problem.

Memory assistance using reference counting memory mechanisms has been attempted but is of limited general purpose use as it cannot clean up loops of objects or similar, common

cases.[8, 15, 26, 31] In 1997, Wise et al. concluded that their hardware reference-counting heap was the first and showed scientific break-even.[31] They concluded that GC must support parallel systems and new technologies, or it will fade in favour of alternatives with lower software development cost, such as C-style memory management. 20 years on, GC has not faded but instead grown in use, popularity and performance. Their system also targeted on-disk management and utilised an expensive, offline, software (hardware supported) mark-sweep collector - a heavyweight design probably not suitable for integration with modern systems of any scale, given the range of storage technology today.

Hardware support for the copying (or sweeping or compacting) stage of GC has been attempted and achieved incremental improvements but with no particular breakthroughs.[24, 29] Wright et al designed an innovative but complex system, using a separate object-only address space.[33] Their design permitted a fully parallel software GC on multi-core processors or in multi-processor systems but with at least one dedicated core during mark-sweep. Although retaining software backwards compatibility, this necessitated significant modification of new programs and the operating system to utilise the split address space - a significant imposition on software developers. Furthermore, the design only supported JVM execution, citing the unsafe subset of C# as a barrier. However, they did show that memory hierarchy modifications can be lightweight and compatible with existing system design. These are important features if hardware GC is to be adopted by the wider community, which is justifiably concerned with software redevelopment and backwards compatibility.

### 3.3.2 Co-processors

A number of co-processor designs have been proposed, including offloading the marking phase to a GPU.[7, 10, 20, 21, 23, 25, 27, 28] However, co-processor design has many of the disadvantages of software GC design, in particular, requiring barriers during copying/compacting, achieved with STW or by hardware support in the main processor (or equivalently, the memory bus). Hard real-time performance was achievable with these schemes but as Nilsen and Schmidt concluded, the cost of an entire extra GC module was not outweighed by the performance gain, particularly for multi-core C++ systems of the time, which could achieve very similar performance levels.

### 3.3.3 ISA Extensions

The main work on ISA extensions comes from papers by Chang et al, the most useful being "DMMX (dynamic memory management extensions): An introduction" in 1999.[6] Unlike any other that this author has seen in the field of hardware GC, this envisaged a combinatorial logic, bitmap memory structure to provide constant-time allocation and sweeping. By considering extending an existing ISA they also hoped to support the Java Virtual Machine in hardware. It is unclear why their work hasn't been followed up, though improvements in cache and processor speeds may have made it seem superfluous. They also amusingly included the prediction "by year 2010, there will be 10 times more embedded system programmers than general-purpose programmers" by Atherton, 1998 in their conclusion. Almost the exact opposite has occurred with the web and JavaScript, with fewer systems programmers than ever. Even the growing IoT market seems unlikely to change this.

### 3.3.4 Profiling

Heil and Smith created a concurrent GC using hardware-assisted profiling that achieved improvements upon the standard Java generational GC of the time.[11] However, this expensive in hardware and offers little benefit compared to effective software optimisation or alternative hardware-assistance techniques.

## 3.4 Full Hardware GC

In 2012, Bacon et al developed the first real-time GC fully in hardware using "miniheaps" that supported fixed-size objects and a stack for the free list.[3] Although a first, as noted by Maas et al in 2016, it is only capable of special purpose applications on FPGAs, utilising properties of block RAMs, and requires the mutator program itself to be synthesised to hardware.[19] The design shows no promise of being expandable to general purpose GC for CPUs or larger scale systems.

Maas et al presented the first, full hardware, concurrent, general-purpose GC targeting servers (presumably mindful of cloud computing applications).[19] They observed that the current climate of hardware and software makes it an ideal time to revisit the idea of hardware GC. They adopted the same principle as the IHGC: that maximum memory bus utilisation will lead to the most efficient system. But favouring full address bus backwards compatibility and following the same Pauseless algorithm as Azul's C4 design, required a read barrier to handle relocated objects. This is a significant

step forwards for hardware GC but their system, compared to the IHGC, is very complex and requires significant compiler (and application) modifications.

By contrast, the IHGC recognises that address bus compatibility is unnecessary given current software design, so much simpler integration with existing CPUs can be achieved.

Chapter Four

# *The Integrated Hardware Garbage Collector*

This thesis presents an adapted version of the original IHGC (Appendix A), designed by Professor David May at the University of Bristol. The design has been integrated with the open-source PicoRV32 implementation of RISC-V.[30, 32] This chapter describes the design's operation and adaptations made to fit the register-based CPU.

## 4.1 Memory Model

Conceptually, the IHGC is a memory management unit sitting between the processor and main memory, imposing a new memory model unlike any traditional model. Traditional models view memory as a randomly-accessible, flat address space. (Paging or segmentation mechanisms may be layered on top to provide virtual memory.) The IHGC memory model views memory as a sparsely-connected, unordered space of *tuples* (objects).

Tuples contain words called *fields* which contain either plain values or pointers. Only fields can be accessed and comparing locations of tuples is invalid. Pointers point to fields and consist of a handle and an offset (split between a word's upper and lower parts). If the offset is beyond the bounds of the tuple to which the handle refers, the pointer is invalid. A special $nil$ handle points to nothing. Two pointers are equal if they point to the same field or to any $nil$ field. New pointers are formed by allocating memory or offsetting an existing pointer. Bounds checks can be performed at formation or at access time.

Address buses contain pointers, and data buses contain pointers or values, indicated by an additional pointer flag bit (called $pflag$ or subscripted as $ptr$).

## 4.2 Programming Model

In practice the memory model presents no challenges to the programming model used in C, C++, C#, Java, Python, Haskell or other languages. As such, the ISA changes (see Chapter 6) are seamless for the majority of software. However, untrustworthy software would probably be blocked but that is a significant motivation for using the design.

The memory model supports general programming but requires that software does not convert between values and pointers. Such conversions have been gradually phased out of C/C++ programming due to their inherent risk. The ideas for future work (see Chapter 12) suggest an approach to eliminating all conversions.

## 4.3 Mark-Compact Algorithm for Garbage Collection

Figure 4.1 and table 4.1 summarise the IHGC Mark-Compact algorithm which has four stages and two possible interruptions.

| | |
|---|---|
| 1. Root-finding | Finding pointers at which to start searching for tuples to mark - taken from CPU registers. |
| 2. Marking | Recursively searching every pointer within every tuple found. Each tuple found is "marked". |
| 3. Sweeping | Marked tuples are compacted (copied) to the bottom of memory. Unmarked (dead) tuples are overwritten by marked (live) tuples or by zeros. |
| 4. Finish | The end of the live memory is found. The M&S process is reset and begins again. |
| A. Allocation | The CPU requests a new tuple, interrupting the M&S cycle - blocked for at most a few clock cycles if enough memory is available. |
| B. Access | The CPU requests a read-from/write-to a tuple, interrupting the M&S cycle - blocked for at most a few clock cycles. |

Table 4.1: Stages of Mark-Compact Algorithm

Root-finding checks for pointers to *deep* tuples (tuples which contain pointers not just values). If a non-$nil$ deep pointer is found, it is marked, the tuple's size is added to livesize and then the tuple is scanned. Otherwise, the next register is selected (split into $mark_{next}$).

$mark_{next}$ loads the next tuple's information and then moves to $mark_{scan}$. Every word of the tuple is then scanned for pointers, which are added to the marking list using $mark_{add}$. By the end of marking, all reachable (live) tuples will have been marked. During marking, the total size of live

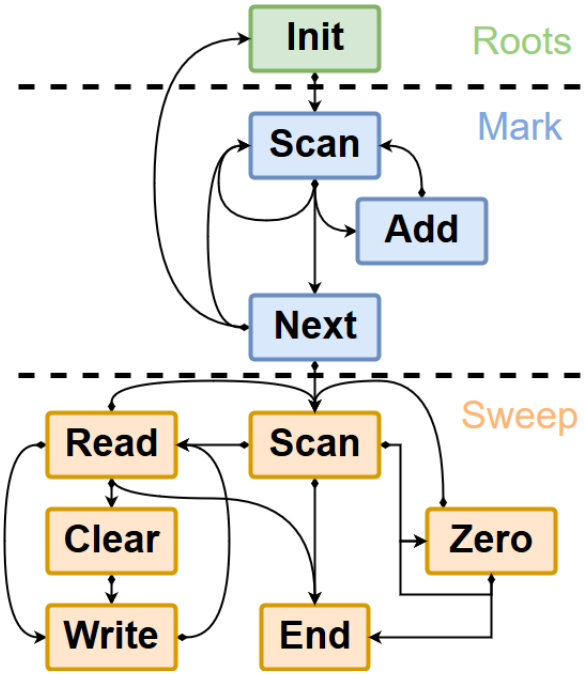



Figure 4.1: The IHGC Mark-Compact Algorithm

tuples ($livesize$) is accumulated for use during sweeping. $mark_{next}$ detects end of marking and moves to $sweep_{scan}$.

$mark_{sweep}$ moves from the bottom upwards through memory. All tuples are guaranteed to appear contiguously from the bottom of memory. $src$ and $dest$ are physical addresses of where to copy from or clear, and where to write to respectively.

A tuple's first word is hidden from the CPU and contains its handle (tuples are one word larger than the CPU requested). Sweeping reads the first word to access the handle and so determine the tuple's size. Sweeping compacts live tuples to the bottom of memory. A marked tuple is either left where it is (if $src = dest$) or copied from $src$ to $dest$ ($sweep_{read/clear/write}$). An unmarked tuple is either within the live size of memory, in which case it is skipped, or it is beyond the live size of memory, in which case it is zeroed out ($sweep_{zero}$). If a tuple is marked and copied to lower memory but the tail of the object is beyond livesize, the extra words are zeroed out ($sweep_{clear}$). $src$ and $dest$ are accumulated in complete tuple sizes. The sweep position within a tuple is tracked by $index$. Sweeping clears the tuple $mark$ flags and appends dead tuples to the $free$ list.

Allocations and accesses can occur during marking and sweeping, handled by the $getmem$, $read$ and $write$ interruption states. Interruptions are blocked until a transition between two M&S states. The design requires only one directory and/or one memory access per main state. This means the

blocking time for access and (if the system is not out of memory) allocation is at most the longest state's duration, which in practice is very short.

New tuples are initially marked so will not be immediately removed by sweeping. Accesses check if the tuple is being swept and reads/writes to/from the appropriate place to maintain consistency.

If an allocation request occurs when the system is out of space or handles (OOS or OOH), the IHGC will block for up to two complete M&S cycles. The second M&S cycle determines if the system is entirely out of memory.

## 4.4 IHGC Memory

Main memory contains the tuple space and each word requires an additional bit to indicate if it contains a pointer or not. In practice approximately 3% extra main memory is required by the GC for the $pflag$ and per-tuple handle word.

The directory memory, as shown in Appendix A, tracks the size, location and state of tuples, including allocated tuples and the free handles list. In a 32 bit system with 4GiB main memory (excluding $pflag$ bits) and 16 bit handles, the directory requires 62 bits per tuple, requiring 496 KiB of high-speed memory (L1/L2 cache). However, with 4,096 handles and maximum object size of 65,536 bytes, only 27 KiB is required (for 256 MiB of main memory). Future work will consider a technique for allowing a few large objects alongside many more smaller objects, by splitting the handle space.

Chapter Five

# *Criteria for Feasibility*

The IHGC is designed for embedded or micro-controller processors and commercial implementation. Key aspects are implementation, size and speed, including Worst Case Execution Times (WCETs).

## 5.1 Size, Complexity and Usage

| # | Focus | Description |
|---|---|---|
| **1.** | **Complexity** | *Difficulty of correct implementation* |
| 1.0 | | Implementable within 2.5 months by one person working full time. |
| 1.1 | | Implementable in current versions of HDLs |
| 1.2 | | Synthesisable using current commercial tools |
| 1.3 | | Only requires hardware elements that are standard to FPGAs |
| 1.4 | | Can be integrated with an existing ISA |
| 1.5 | | Only one additional instruction, for allocating tuples, added to the ISA |
| 1.6 | | No structural changes required to integrate with a typical micro-architecture |
| 1.7 | | Necessary system optimisations are similar to commercial systems |
| **2.** | **Size** | *Quantity of hardware required* |
| 2.0 | | IHGC is smaller than the chosen embedded processor (excluding memory) |
| 2.1 | | IHGC fits onto a low-end FPGA with the CPU core and memory |
| 2.2 | | Equivalent GC/non-GC programs execute with same size of main memory |
| **3.** | **Software** | *Required software changes to execute programs on the new system* |
| 3.0 | | Only *cut-down* modifications to existing C programs |
| 3.1 | | At most *re-compilation* and minor *re-write* of the C standard library. |
| 3.2 | | No changes to C-based software programming or memory model. |
| 3.3 | | No necessary compiler modifications. |
| 3.4 | | No reliance on non-standard or not best-practice software design. |

Table 5.1: Complexity, size and software feasibility criteria

*Note: Energy consumption is ignored since accurate energy estimates require optimisation of the system and an FPGA implementation is not representative of energy consumed by an ASIC implementation.*

## 5.2 Responsiveness

Response times refer to the time taken to process a response once the GC is ready to do so. Response time does not include blocking time, which is covered in "overheads".

| # | Focus | Description |
|---|---|---|
| **4.** | **Allocation** | *Time taken to allocate a new tuple* |
| 4.0 | | GetM instruction is faster than smallest-possible, equivalent C *malloc* routine |
| 4.1 | | GetM is tightly bounded |
| 4.2 | | GetM faster than software GC allocate functions (e.g. in C#) |
| 4.3 | | GetM operates regardless of CPU state (e.g. during interrupts) |
| **5.** | **Access** | *Time taken to access memory of a tuple* |
| 5.0 | | Memory access within 5 times max. speed for the memory type (BRAM/SRAM/DRAM). |
| 5.1 | | R/W time is within 2 orders of magnitude of comparable real systems (Expected improvement from optimisation and FPGA vs. ASIC implementation) |
| 5.2 | | R/W time is tightly bounded |
| 5.3 | | R/W time is within 2 orders of magnitude of cutting-edge systems |

Table 5.2: Responsiveness feasibility criteria

## 5.3 Overheads

Out of Memory (OOM) refers to either being Out of Handles (OOH) or Out of Space (OOS).

| # | Focus | Description |
|---|---|---|
| **6.** | **Allocation** | *Time spent blocking before a GetM request is processed* |
| 6.0 | | GetM only blocked for more than 1 M&S state transition when OOM |
| 6.1 | | GetM blocked for at most 1 M&S state transition when not OOM (Both 6.0 and 6.1 are necessary given possible error cases). |
| **7.** | **Access** | *Time spent blocking before a R/W request is processed* |
| 7.0 | | R/W requests only ever blocked for at most 1 M&S state transition |
| 7.1 | | R/W requests never blocked for longer than max. time taken for a R/W request |
| **8.** | **Space** | *Additional memory required compared to a non-GC system* |
| 8.0 | | Directory memory uses similar space to L1 or L2 cache sizes |
| 8.1 | | Additional main memory per tuple is at most 1 word |
| 8.2 | | Additional main memory per word is at most 1 bit |
| 8.3 | | Live size for an IHGC-based program is never greater than its best C equivalent (excluding extra word per tuple) |
| 8.4 | | Heap point is approximately 10% larger than live size under worst-case conditions. |

Table 5.3: Overheads feasibility criteria

## 5.4 Scalability

Capability of the GC to scale from small to large embedded/micro-controller systems - applications scale is not considered in this thesis.

| # | Focus | Description |
|---|---|---|
| **9.** | **GC Operation** | *Handling varying GC pressure levels* |
| 9.0 | | GC continues to complete M&S cycles at any pressure level |
| 9.1 | | M&S time is tightly bounded |
| 9.2 | | M&S time is proportionate to R/W/GetM frequency/size |
| 9.3 | | M&S only stalls (STW) if OOM and a GetM is waiting |
| 9.4 | | M&S time is predictable |
| 9.5 | | Consistent operation across range of GC utilisation for a given program |
| **10.** | **Memory** | *Managing different sizes of memory* |
| 10.0 | | Number of memory accesses is proportionate to number and size of tuples |
| 10.1 | | Directory memory size is proportional to number of tuples |
| 10.2 | | Memory for dead tuples is proportional to creation/death rate |
| 10.3 | | OOM condition(s) are predictable and calculable for a given program in a uni-processor, single-thread system. |
| **11.** | **CPU Speed** | *Handling higher CPU speeds* |
| 11.0 | | Longest path shows potential for optimisation to faster clock speeds |
| 11.1 | | GC clock speed can be the same as CPU clock speed (or faster) |
| **12.** | **CPU Efficiency** | *Handling varying memory bus utilisation levels* |
| 12.0 | | GC can operate under maximum CPU-IHGC bus utilisation |
| 12.1 | | GC slows max. bus speed by at most 1 order of magnitude |
| 12.2 | | GC correctly handles simultaneous GetM and R/W requests |
| **13.** | **Caches** | *Compatibility with cache hierarchies* |
| 13.0 | | L1/L2 caches can be placed between GC and CPU core |
| 13.1 | | L3 cache could be placed between GC and main memory |
| 13.2 | | Cache is not required for GC to meet other criteria (excl. directory) |
| **Total** | | **51 criteria** |

Table 5.4: Scalability feasibility criteria

Chapter Six

# *Approach to Implementation*

This chapter presents the approach to implementation of the IHGC (in bottom-to-top fashion), the system and the software benchmarks. Tight bounds for the IHGC are calculated and compared to the feasibility criteria.

The implementation is at Research Demonstration Level, using Feldman's technological maturity scale.[31] Future work could certainly improve upon the tight bounds, compactness and elegance of this implementation. This is a proof of concept, created in a limited time with little attention given to clock-cycle optimisation. The implementation's code and all original software is provided with this thesis.

## 6.1 The IHGC

### 6.1.1 Design and Implementation

For simplicity, and to demonstrate the portability of the design, the IHGC was implemented in a single, independent file forming a single Verilog module. Within this there is the main state machine and a few, very small supporting ones. In addition, a number of wires are created to help maximise re-use of common combinatorial expressions.

The design has 80 registers of varying sizes, many being only a few bits wide. In general, a one-hot encoding is used for the states of the state machines. Most of the additional registers not part of the original design (Appendix A) are used by the statistics gathering/outputting or are state registers.

Optimisation of the tasks for accessing memory and the directory is an important area of future work. At present, there is an inessential one clock cycle gap between every memory or directory access. Splitting the tasks into "begin" and "end" tasks would sometimes save up to 4 clock cycles.

To support the full reset mechanism for the test framework, the $init$ state was added to the main

state machine. This state clears all of memory, excluding the program, and initialises the directory to a clean state that includes a program object (guaranteed to be handle $0$). This significantly impacts the system start-up time but could be optimised in future work.

The main state machine has four possible routes on each clock cycle: Reset, GetM, R/W, or M&S. GetM is an interruption state that responds to an allocate memory request from the core. R/W are also interruption states and handle a read/write requests. M&S is the default case and executes the main states of the GC state machine.

Interruption states only occur when the main state machine transitions from one state to the next. Some of the main states have internal state machines which can loop on themselves, in which case, special provision is made for interrupting these internal loops at a safe moment.

There are 11 main states each of which follows a similar pattern. Firstly, any conditions which choose the action of that state are evaluated. Then a read or write is performed and the state waits for it to complete (by checking $*\_ready$). If a second read or write is required, a sub-state-machine is used to control the order. There is a one clock cycle delay between the first and second read or write that could be eliminated, but was useful for debugging. Lastly, registers are updated including the $msm\_state$ register. Sub-state registers are also reset for the next cycle.

Livesize is updated by setting the *add* or *set* registers and toggling the respective *do* bit.

The wires for values read from main memory or the directory automatically switch between current and cached values according to the $*\_ready$ signal, making it simple to use the signals in the main state machine. A write to memory does not alter the cached read values.

To aid debugging, $\$display$ statements are placed in every state and some key sub-states so that during simulation, lock-ups and incorrect progressions can be seen. These do not affect the synthesised design.

Memory is byte addressed so many of the address signals are left-shifted by two to translate from a word address to a byte address.

The original design was for a stack-based machine. However, the implementation in this research was for a register-based machine, so a good first-attempt was made to adapt the design. In the vast majority of cases, the implemented design functions correctly. However, there is a known rare case in which tuples may be swept too early. A program which builds a linked list, then, starting from the top of the list, rapidly loads a pointer to the next element into a register, then clears the link pointer in memory, discards the parent element pointer, and repeats this down the list, may cause the IHGC

to discard the entire list early. This is because the chain that the marking process is attempting to follow is broken before it can mark the complete chain, but the IHGC does not re-scan registers to check for new possible root pointers. However, the author expects such a program to be extremely rare, certainly inefficient and very likely to be optimised away by compilers.

In hardware-assisted GC systems the case described is usually avoided by marking the tuples of overwritten pointers or of pointers read from memory. Typically this is implemented using a read or write barrier. Bacon et al used a read-before-write mechanism and marked overwritten pointers.[3] However, this is not necessary as an alternative solution is already known. Future work will show that it is sufficient to simply iterate over the CPU's registers until the tuples of all pointers in the registers for a given snapshot are marked. This can be shown to always terminate and is likely to have the same M&S WCET as the current design.

### 6.1.2 Evaluation

The criteria for feasibility place a number of requirements on the implementation which are evaluated directly from the code and synthesis reports.

Items 1.0 to 1.3 are satisfied by the Verilog implementation that was created from February to April 2017 by this author, working approximately 5 days per week for 7.5 hours a day.

#### 6.1.2.1 Synthesis Criteria

Synthesis reports show that around 9% of FPGA logic cells are consumed and 55% of the block RAMs. Of this, $\tilde{4}8\%$ is the processor and $\tilde{5}0\%$ is the IHGC. The remainder is used by the system and memory. The IHGC includes the statistics gathering and output hardware as well as some debug mechanisms. Overall, the IHGC meets criteria 2.0 and 2.1.

The IHGC is synthesised with the rest of the system to use the same global clock signal as the CPU, thus satisfying 11.1. The size of the directory is given by equation 6.1. This shows that criterion 10.1 is met.

$$size = \frac{(address\_size + handle\_size + length\_size + 2) * handles}{8} \tag{6.1}$$

$size$ is in bytes, $address\_size$, $handle\_size$ and $length\_size$ are in bits and $handles$ is the maximum number of handles. An example calculation is provided in appendix C. A typical real-time, embedded processor, such as the ARM Cortex R4 [1], has between 4 and 64KiB of L1 cache

memory.[2] For a 32-bit word size, 65kWords main memory, 16-bit handles and maximum 4096 handles (determined by number of directory entries), the directory size is 23KiB, satisfying criterion 8.0.

The design does not require a specific size of main memory as this is specified as a parameter to the IHGC. For every allocation requested, the actual size allocated is increased by one word. The additional word is used to hold an immutable copy of the reference for that tuple. Thus at most one additional word is used per tuple, satisfying criterion 8.1. The IHGC design also requires a flag for every word in memory indicating whether that word contains a pointer or not ($mem_{pflag}$ - $pflag$ is used instead of $ptr$ as in the original design in order to reduce confusion when reading the code). The $pflag$ can be implemented as a single bit per word, thus criterion 8.2 is satisfied.

#### 6.1.2.2 Functional Criteria

The IHGC module is able to mark, sweep and respond to read, write and GetM requests independently of the processor core's state. This satisfies criterion 4.3 and is an important feature of the IHGC. In particular, it enables the core to allocate memory during interrupt requests, an extremely useful feature and apart from exceptional cases, not possible in any current system. This feature opens up a realm of new possibilities for software design, especially operating system and device driver design.

The code in *"gc.v"* (attached with this thesis) contains the code which controls when the main state machine should be interrupted to handle a read, write or allocate request.

$is\_safe\_to\_jump\_state$ consists of 3 conditions. Firstly, there should be no ongoing directory or memory access - these only happen during a M&S state or an interrupt state and are entirely contained - these conditions are not strictly necessary. Secondly, the IHGC should not be initialising - this will occur once at *reset* and $msm\_state = previous\_msm\_state$. Lastly, the main state machine should have just transitioned from one state to the next. However, this last condition is extended as several of the states are able to loop on themselves. The extra conditions enable interruption between the start and end of such a loop.

$out\_of\_handles$ and $out\_of\_space$ indicate whether the IHGC has run out of memory to allocate. Out-of-handles is detected by the $free$ list being nil. Out-of-space is only valid when a getm request is being made and detects whether there is sufficient space left in the heap for the allocation to be made - a simple overflow check is also included.

[1] ARM and Cortex are trademarks or registered trademarks of ARM Limited (or its subsidiaries) in the US and/or elsewhere.

$do\_getm$ and $do\_rw$ control whether their respective requests should be handled or not. $doing\_action$ ensures that the handler continues after its first clock cycle. GetM is always handled before RW (criterion 12.2). $do\_getm$ will only become true when it is safe to interrupt the main state machine and a getm request is being made. This will block for at most one M&S state if the IHGC isn't OOM (criterion 6.1). Otherwise, it will block until memory is available (i.e. a M&S cycle has occurred) and it is safe to interrupt (criterion 6.0). $do\_rw$ becomes true when it is safe to interrupt and an R/W request is being made. Thus R/W only blocks for at most one M&S cycle - criterion 7.0.

Criterion 9.0 requires the IHGC to continue to execute M&S cycles even under high pressure from the processor core. In this context, high pressure means when the core is making R/W/GetM requests as fast as possible (which can be achieved by holding the respective Valid signal high continuously). This requires that between each request, at least one main state executes. A request can only begin to be handled when the main state machine has transitioned between two states (i.e. a main state has been executed) or one of the single-step main states is looping on itself. However, the extra $!*\_ready$ conditions ensure that one clock cycle is available between requests for a single-step main state to start executing thus blocking the next request until at least one main state has executed. In the case that the processor issues GetM and R/W requests simultaneously, the clock cycle gap is not created. Thus criterion 9.0 is not fully satisfied. However, it is reasonable to expect that such a case will not occur with an unpipelined processor. Furthermore, it would be trivial to fix this case as described in future work.

Criterion 9.3 requires that the IHGC only stalls when it is OOM and a GetM request is waiting. The IHGC code (*"gc.v":1353-1381*) shows that OOM is only asserted when a GetM is waiting and a full M&S iteration has been completed with insufficient memory becoming available. Additionally, this is the only condition which may stall the main state machine. The interruption states may stall due to nil pointers or out of bounds accesses, which are not handled in this implementation. Full error handling will be explored in future work.

#### 6.1.2.3 Temporal Criteria

R/W is only blocked for at most one M&S cycle. Therefore, the tight upper bound on R/W blocking time is the maximum duration of any M&S cycle, plus the interrupt transition time. Appendix D presents clock-cycle-accurate overview flow diagrams of all the states of the IHGC and of memory

accesses. These show that the longest M&S state is $sweep_{scan}$ lasting at most 11 clock cycles. The tight upper bound on R/W blocking time is therefore 12 clock cycles. The R/W response time is max. 17 clock cycles for write and 7 clock cycles for read (for a valid pointer). Thus the tight upper bound (WCET) on any valid read/write request is 29 clock cycles, meeting criterion 5.2. Cutting-edge, real systems, generally have memory latencies of between 4 and 100 clock cycles. 29 clock cycles can be optimised further (as is evident from the wasted transition cycles shown in the diagrams) and is within 2 orders of magnitude of real and cutting edge systems. All this shows that criteria 5.1, 5.3 and 7.1 are met.

GetM has two interesting upper bounds; one when the system is not out of memory and the other when it is (and a M&S must occur to clean up dead memory). The case when the program tries to allocate more live memory than the heap has available is ignored. If the system is not out of memory, the upper bound is one M&S state plus transition plus response, which Appendix D shows is max. 28 clock cycles. If the system is out of memory, then the longest possible route through M&S must be considered.

The longest possible GetM blocking time occurs when the GetM is blocked by the mark and sweep cycle(s). The following proof provides the WCET for a single M&S cycle when the IHGC is blocking a GetM request during sweeping (but not necessarily during marking).

1. There are two phases to the algorithm: Marking and Sweeping.
2. If a tuple is not "deep", the marking phase does not scan it. So for WCET we assume all tuples are "deep" (for example, they all contain a valid pointer or the NIL pointer). We also assume no tuples are marked at the start of the marking phase.
3. For the first part of the proof, it must be noted that the number of memory operations per word of the heap is one of the two determining factors on M&S time. The second factor, used later, is the structure of the live tuples.
4. By the end of the Mark phase, there will be $h$ words in the heap.
5. Of the $h$ words, $l$ of them will be live and so will have been read during marking.
6. During sweeping, at most $l$ words will be moved by reading then writing them.
7. During sweeping, $h - l$ words will be zeroed, either by clearing just after reading or after compacting finishes.

8. Both the $sweep_{zero}$ state and the $sweep_{clear}$ perform a single memory write and no other operations. As such, in a good implementation, they should take the same length of time and so the operations can be considered together rather than independently.

9. Thus at most $l + (2 * l) + (h - l)$ memory operations will occur, provided that at least one dead word is beneath the live words (i.e. $l < h$). In simplified form the equation is $(2 * l) + h$, provided $l < h$).

10. Thus for WCET, we require maximum $l$ and $h$, with one word of dead memory at the bottom-most address. This corresponds to all but Word 0 being live.

11. The marking phase marks every available tuple in the system. Marking takes longest when all possible tuples are allocated and form a linked list. This is because the algorithm is a breadth-first search for tuples and so linked lists cause the longest route through the marking states. It takes longer to $mark_{scan} \rightarrow mark_{add}$ than to $mark_{init}$. In other words, marking a tuple of a pointer found in another tuple, takes longer than marking one found in a CPU register.

12. Figure 6.1 shows the memory layout for Mark and Sweep WCET.

13. Based on this diagram, the route through the M&S states is derived. Using the clock cycle bounds for each state provided in appendix D, the following equations for M&S WCET are derived.

14. $$\begin{aligned} T_{ms-wcet} = \; & Tmark_{init-markable} + Tmark_{next-notnil} * (N-1) + \\ & Tmark_{scan-readmem} * (M-N-1) + Tmark_{add-notmarked} * (N-2) + \\ & Tmark_{scan-endandcurrentisnil} * (N-1) + Tmark_{next-nextreg} * R + \\ & Tmark_{init-notmarkable} * (R-1) + Tsweep_{scan-deadandinsidelivesize} + \\ & Tsweep_{scan-liveandsrcnotdest} * (N-1) + Tsweep_{read-notend} * (M-3) + \\ & Tsweep_{write} * (M-3) + Tsweep_{read-endoftuple} * (N-2) + Tsweep_{clear} + \\ & Tsweep_{read-swepeend} + Tsweep_{end} \end{aligned}$$

15. $T_{ms-wcet} = 22N + 11M + 3R - 41$

16. Thus for 65,536 words of main memory ($M$), 4,096 handles ($N$), and 32 CPU registers ($R$), the WCET in clock cycles is: 811,063 clock cycles

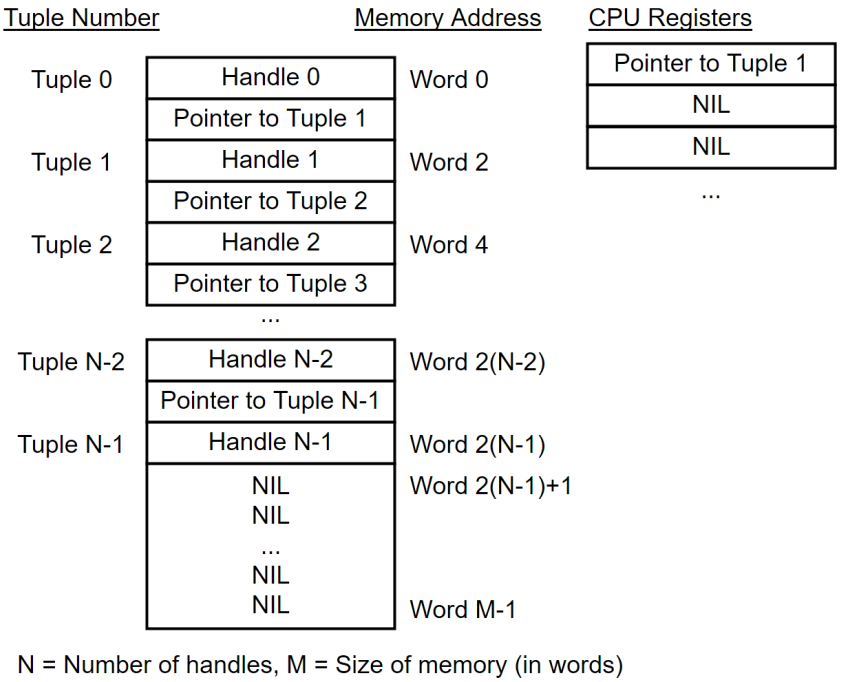



Figure 6.1: Memory layout for WCET of Mark-Sweep when blocking GetM during sweeping

The following argument extends upon the proof above to give the Worst Case Blocking Time (WCBT) of a single GetM request.

1. Assume a memory structure as shown in the previous proof.
2. The IHGC marks the pointer in register 0 but does not yet process the rest of the linked list.
3. The CPU now loads Ptr to T2 from T1 into register 0.
4. The CPU now makes a GetM request for 2 words.
5. The IHGC now proceeds with marking and sweeping. At the end of the first mark-sweep, only a single word (T0) has been freed, so the GetM request is still blocked.
6. The IHGC now performs a second mark-sweep cycle.

   The memory layout is exactly the same as the first time, only with 1 handle free (T0) and the top-most word of memory free.

   This time, the CPU cannot unlink T2 from the top of the list because it is still blocked by its previous GetM request.

   The second mark-sweep frees T1, freeing a second word of memory.

7. The heap-point is updated in $sweep_{end}$.

8. 2 words of memory are now available and so the GetM request is processed.

The WCBT of GetM is thus "1 M&S of maximum sized heap", and "1 M&S of max. heap minus 1 and 1 tuple free". Thus the WCBT of GetM is: $811,063 + 811030 = 1,622,093$ clock cycles (at 100MHz, 16.22ms). In practice, it is exceedingly difficult to produce this case with a real program (even one designed to produce the desired situation). Additionally, the system designed here always keeps the program object live and at the bottom of memory, so the WCET and WCBT would also be reduced.

If the processor attempts both a read/write and a GetM request simultaneously, the GetM will be handled first, so the WCET of GetM remains the same. The R/W will be blocked for at most: GetM plus one transition time plus another M&S state plus another transition time, which is at most 24 clock cycles on top of the GetM, so the WCET of R/W becomes 41 clock cycles (ignoring the GetM). However, the original IHGC design does not expect to handle simultaneous R/W and GetM requests.

The WCET for M&S and the WCBT for GetM are both tightly bounded in all cases, so criteria 4.1 and 9.1 are met. Additionally, since the amount of memory that has been allocated directly controls the M&S time ($l$ in the proof of WCET above), criterion 9.2 is met.

Lastly, the longest path through the M&S does show some potential for optimisation to faster clock speeds and for cutting out clock cycles entirely. Thus criterion 11.0 is met.

## 6.2 Choice of CPU ISA & RTL

A primary aim of this thesis was to demonstrate that the IHGC can be integrated with a real ISA with little modification to existing, working RTL. However, there was also a tight time constraint on the project. It was also desirable to choose an open-source ISA which would allow the full IHGC and ISA modifications to be published for wide review and hopefully, commercial use.

Three ISAs were reviewed: ARM-v8, MIPS32, and RISC-V (32-bit). These were chosen as they are the only widely used ISAs (RISC-V has had a growing userbase in the last few years). RTL code was only available for a MIPS32-based processor and various versions of RISC-V processors. MIPS32 was not chosen because the RTL obtained for the MIPS32 processor was too large and complex to be sufficiently understood in the time available (as it was for a full, commercially-available processor).

Additionally, modifications to the ISA could not be publicly released under the license agreement. Thus RISC-V was the only remaining option.

There are a number of good RISC-V implementations available open source: Rocket, VScale, Pulpino and PicoRV32. Each of these was reviewed. Ultimately, PicoRV32 by Clifford Wolf was chosen for its small, easy to understand design, use of Verilog and restricted feature set. The version of PicoRV32 chosen was the latest available at the time (commit a2107e[32]) and implemented only the core integer instruction set of RISC-V - sufficient for demonstration purposes. It could also be synthesised to a variety of FPGAs and using a variety of design tools. This research added support for the Zedboard (Zynq-7000) FPGA using Xilinx Vivado 2016.1 WebPACK Edition.

## 6.3 Integration

The IHGC design is integrated at Instruction Set Architecture (ISA) and micro-architectural levels. ISA level requires the addition of a GetM instruction and modification of existing arithmetic and logical instructions to correctly handle combinations of pointers and values. Move, branch and special instructions may need to be modified to throw relevant errors but otherwise remain the same and so were ignored in this implementation. Micro-architectural level requires the addition of the pointer flag bits to the registers and memory interface, along with the updates required to support the ISA changes.

These changes were applied to the PicoRV32 design. No formal new version of the RISC-V ISA was created (because of time constraints, though this should be done in future work). The following changes were applied:

1. The arithmetic operations $+$ and $-$ were modified to prevent adding two pointers together. Further, two values are added as 32-bit values, but when adding a value to a pointer, only the offset is added to - the handle portion is preserved.

2. The logic operations $==$ and $<_{unsigned}$ were modified to allow only comparison between two values or two pointers. In the case of two pointers, the handles are compared for equality and the offsets are compared either for equality or less-than (unsigned) respectively.

3. Extra flag registers were added to signal error conditions, such as adding two pointers or adding oversized offsets. This is to demonstrate such error checking is possible, but this

implementation did not add the code required to jump to the CPU's *trap* state in the event of such an error.

4. An extra bit was added to each of the internal registers to act as the pointer flag (except $pc$ and $sp$ which must always contain pointers).

5. The memory interface for reading and writing was updated to include the pointer flag.

6. The GetM instruction was added. It accepts a single source register - the size of the new allocation in words - and a single destination register (which can be the same register as the source) in which the new pointer will be stored.

7. For testing purposes, an OUT instruction was added and is described in the Testing Framework subsection below.

To make the code easier to navigate, the original, single file was split into three files, with the two new files containing the memory interface and decoder modules respectively.

The details of exception handling have not been tackled in this project but could be handled by any of the current standards for hardware exception handling (such as traps, JTAG debugging, halting the current process, etc.). Further research at the University of Bristol is creating a better way of handling hardware exceptions cleanly and in a software-model-compatible manner.

## 6.4 Compiler and Program Modifications and Testing Framework

Minimal changes are required to existing C programs in order to make use of the new architecture. The assumption is made that the programs are well formed - in essence, the programs do not arbitrarily convert between numbers and pointers. Though this used to be very common, especially in embedded programming that uses fixed/pre-allocated addresses for everything, in the author's experience it is now restricted to device addresses. Future work will explore a method for dynamically allocated device addresses or for allowing conversion of device address to a memory mapped pointer. Converting a program which conforms to the basic requirement is achieved by "defining" (e.g. #define in C) the *malloc* function to be the new *GetM* instruction and *free* to be no function at all. The *GetM* instruction can be added by defining it in an assembly code file. No modifications to the compiler are required nor to the main program code. The rules for pointer arithmetic enforced by

the architecture are compatible with well-formed programs in C and C# and with all programs in Java, Haskell and other OO languages. The design meets criteria 3.0 to 3.4.

To facilitate testing, an *out* instruction was added to the ISA. This outputs the lowest byte from the source operand register over the UART.

Significant hardware support was added to the system for testing. The IHGC includes fully automatic statistics gathering which are automatically outputted over the UART when the CPU traps or when the IHGC hits a terminal OOM condition. Furthermore, the system is capable of downloading a program over UART, setting up the program object in memory, resetting the IHGC and CPU core, then executing the program. When the program completes, the IHGC and CPU are reset again and the program executes again. This repeats for a third time before the system returns to waiting for a program to be uploaded. When a program is being uploaded, each byte is echoed to allow the uploader to confirm it was correctly received. This was necessary because the simplest version of UART was used, meaning that sometimes bit errors would occur.

A simple C# program for a Windows host computer was developed to automatically upload each program from a list of tests. Upload errors were automatically detected so if a test failed to upload correctly, the test software would restart at the last test which failed and continue. The statistics output from each of the three iterations of each test were automatically saved to machine readable files for use in Matlab graph-plotting scripts. Human readable log files and versions of the statistics were saved to log files and printed to the screen to allow live monitoring of results. The Matlab scripts would automatically exclude results for tests which failed to upload correctly or which hit a terminal OOM condition. In the latter case, the test would be highlighted to the Matlab user.

Chapter Seven

# *System-level Optimisations*

A number of system-level optimisations were required in order for the IHGC to behave correctly. The optimisations required were chosen by simulating the first version of the system and targeting the largest areas causing performance problems. Figure 7.1 shows the results of the simulation without optimisations.

The 63 clock cycle period in Figure 7.1 shows 3 memory requests from the processor core (orange signals). These are instruction fetches which occur prior to every instruction being executed as the core has no pipelining and only 1 prefetched word. The blue signal shows the state of IHGC. In this early version, R/W Requests were handled as a separate state. The yellow signals show requests going to main memory and the pink signals show requests going to the directory. Every instruction is one word in size and a fetch is performed for every instruction, so the core makes rapid requests to the GC with only a 3 clock cycle gap between each fetch. In this 3 clock cycle gap, the IHGC is only able to complete a single memory request and it has to block the next R/W request for 5 clock cycles.

The biggest area to optimise was the frequency of memory accesses, the majority of which were instruction fetches. Two optimisations were used. Firstly, Compressed ISA support was switched on

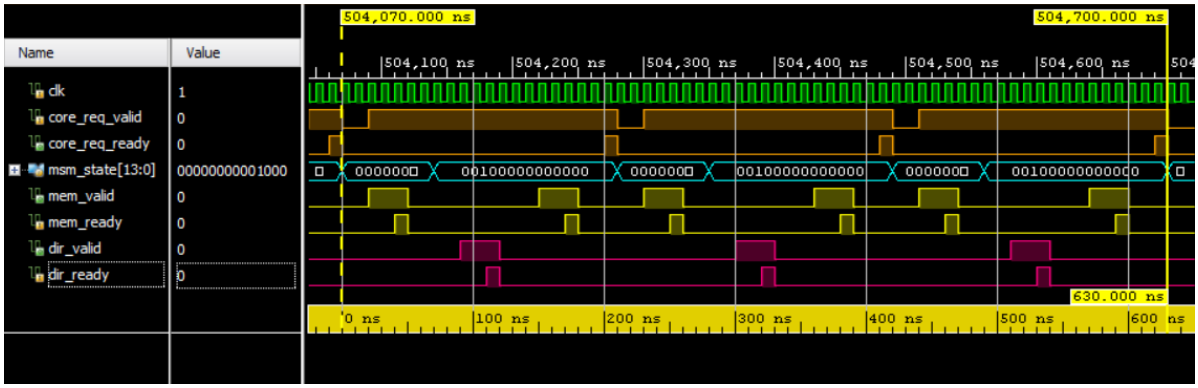



Figure 7.1: System simulation without Compressed ISA or an instruction cache

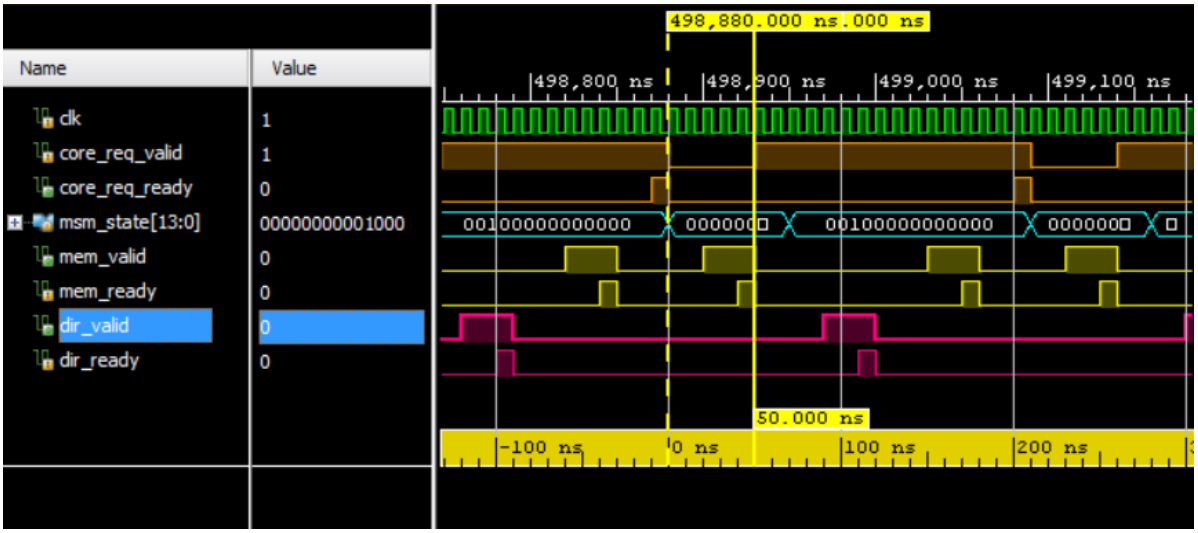



Figure 7.2: System simulation with Compressed ISA and without an instruction cache

(along with the required, extra 16 registers and recompiling the test programs). This reduced the majority of instructions to half-word size meaning two instructions could be fetched per memory access. The effect of this is shown in figure 7.2.

Figure 7.2 shows that the frequency of memory accesses remained the same in the worst case (when non-memory-access instructions were being executed) - 3 per 63 clock cycles. The gap between fetches grew by 2 clock cycles and the blocking time on each R/W request was reduced but it was still larger than the 5 clock cycle gap between fetches. The Compressed ISA significantly improved code density but greater performance was required, so an instruction cache was considered.

A 2KiB, direct-map, read-only instruction cache was added to the system designed to cache the tuple with handle 0 (which is guaranteed to be the program object in this implementation). This results in each instruction being fetched through the IHGC only once. All subsequent fetches are handled by the cache and never reach the IHGC. This frees up a large number of memory cycles for the IHGC and was sufficient to allow it to complete M&S cycles.

Compressed ISAs are commonly used in embedded and real-time systems, as are instruction caches. Types of cache vary but typical sizes are between 4KiB and 64KiB. Therefore, the 2KiB cache created here is reasonable. Furthermore, although it is a direct-map, read-only cache, there is nothing to suggest that other types of cache wouldn't work. Other caches may result in a lower hit rate thus slowing the IHGC, but the IHGC will still continue to work. Thus criterion 1.7 is met.

The use of the instruction cache also demonstrates that traditional cache mechanisms can be

adapted for the new memory model. This means criterion 13.0 is met. Furthermore, the IHGC accesses main memory in the same way as any other MMU, so criterion 13.1 is met.

However, the IHGC cannot function properly if every memory cycle is used by the core. This suggests that for applications-scale processors (which are highly pipelined and designed to maximise memory bus utilisation) the IHGC would require a different design capable of dual memory access. Avoiding every memory request reaching the IHGC in this system required an instruction cache and so criterion 13.2 is not met.

Chapter Eight

# *GetM comparative performance*

A primary function of the IHGC is to allocate memory. This is implemented through a new instruction called 'GetM'. This instruction allocates new tuples that are multiples of the word size (including zero-sized objects). GetM can be implemented as a direct replacement of standard allocation functions such as 'malloc' in many C standard libraries.

In this chapter the performance of the GetM instruction is compared to several versions of standard allocation functions. Firstly, a simple implementation of C malloc is compared by executing two equivalent programs on the IHGC system. The first version fully uses the IHGC and GetM, the second simply allocates a very large object using GetM and uses it as an ordinary heap.

The GNU Standard Library, Java New Object and C# New Object implementations are then compared by analysing the instructions generated at compile time.

## 8.1 Simple C malloc

The GetM instruction allocates new tuples that are guaranteed to contain only zeros marked as values. The equivalent C program (*"simple-malloc.c"*) was implemented as a simple stack-based heap which can be 'tagged' at one point and then reset to that point later on. Resetting also zeros out all cleaned up memory, in order to produce the same behaviour as GetM.

Note that the conversion from 0x0FFE0000 to a pointer is possible only because the hardware implementation does not check the pointer flag properly. This is because error handling has not yet been implemented. Future work would implement error detection and handling and thus prevent this test from working. The tuple is pre-allocated in assembly code so the handle 0x0FFE is available.

These routines compiled with option '-Os' produced the assembly code provided in "simple-malloc.s". There are 35 instructions which contrasts with the one instruction required for GetM. In addition, GetM can be inlined with the main code which cuts branch instructions and allows it to be predicted over by the compiler and processor.

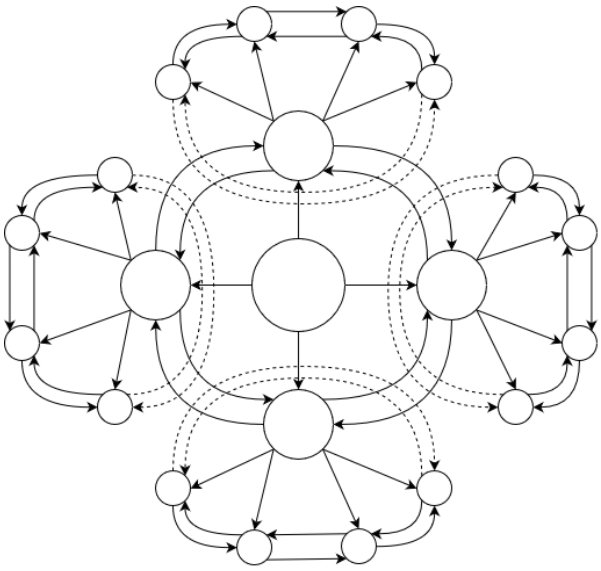


Figure 8.1: Ring network of depth 3 and ring size 4

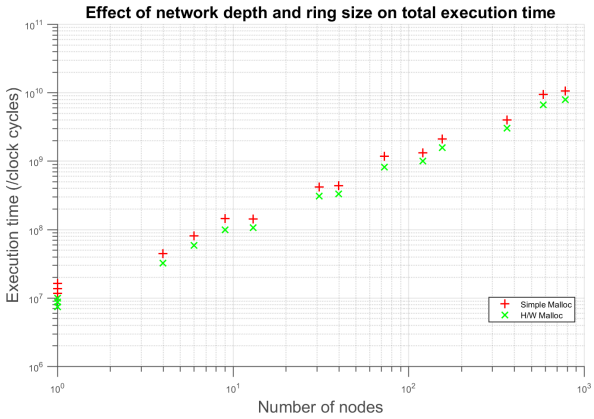
Reducing instructions reduces instruction fetches but when using GetM, it also cuts the time between GetM instructions, which can put greater pressure on the IHGC thus reducing the performance benefit.

The effect of GetM vs. such a simple C malloc is difficult to predict since using GetM requires the IHGC to garbage collect, which introduces blocking times. However, simple C malloc requires more instruction fetches, branches and sequential writes to zero-out memory.

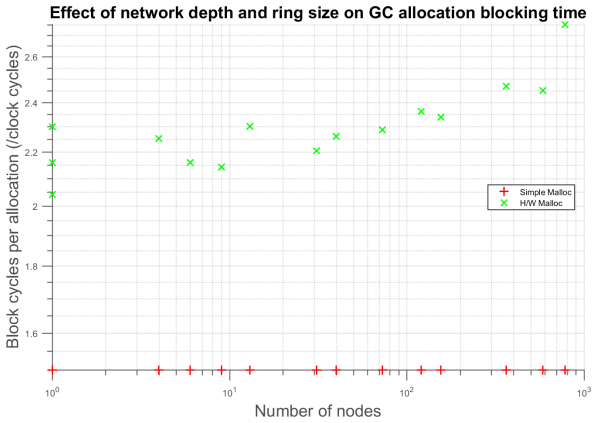
The two equivalent programs were run on the hardware as described. The programs create a tree of doubly-linked rings of nodes as shown in figure 8.1. The results in figure 8.2 show the variation as the number of nodes is increased by increasing the depth of the tree or the size of the rings.

Graph (a) shows that the execution time of the program using GetM is consistently faster than the simple C malloc version. Graph (d) appears to show the r/w blocking time growing linearly, however, as has already been shown in Chapter 4, the blocking time for read and write is upper bounded. It is important to note graphs (d) and (e) show that the read and write times for simple C malloc are faster than that of GetM and do not grow, thus simple C malloc is not being slowed down by the IHGC.

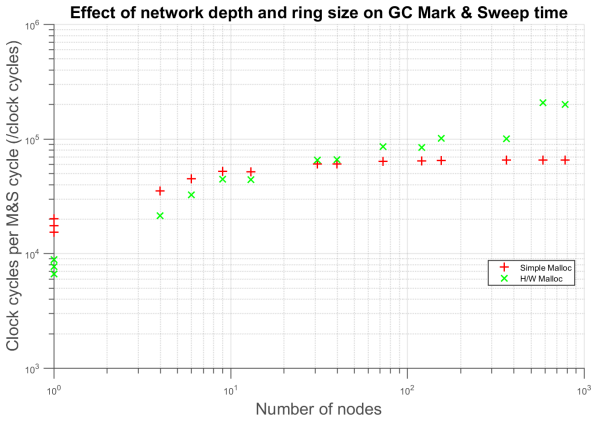
Graph (b) shows that the average blocking time for the program using GetM is slower than Simple C Malloc (which makes only 2 allocations). However, it grows only very slightly (despite the increase in M&S time - see below). This means the overall execution time is not significantly lengthened by the IHGC blocking GetM requests.



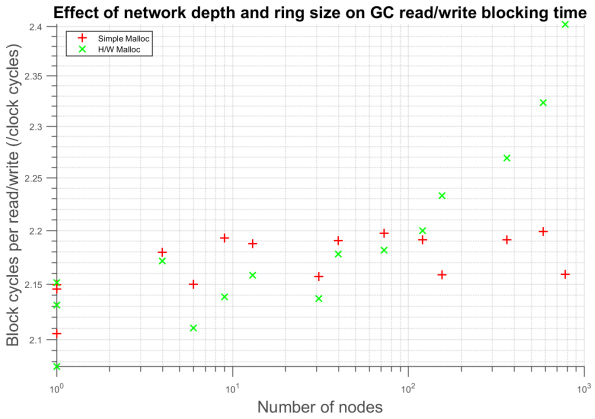

(a) Average execution time



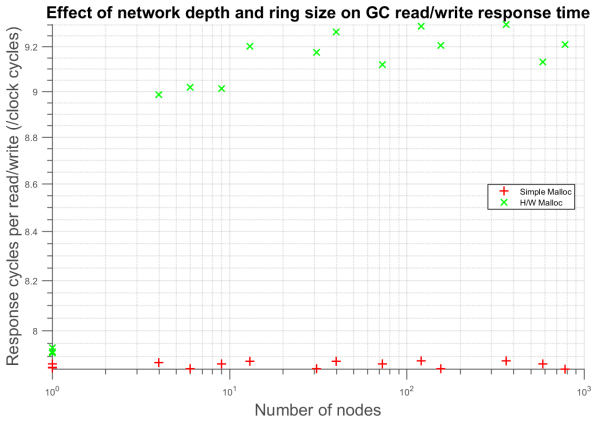

(b) Average GetM blocking time per allocation




(c) Average mark & sweep cycle time




(d) Average read/write blocking time per request




(e) Average read/write response time per request

Figure 8.2: GetM vs Simple C Malloc on FPGA hardware

Graph (c) shows that the M&S time for the GetM-based program is of a similar order of magnitude to simple C malloc version. There are two counter-balancing differences that cause this. Firstly, simple C malloc uses a large object for the heap which is not tight to the actual size required, meaning initially more memory is being marked and swept in each cycle thus the base time for M&S is longer. However, the linear increase in M&S time is caused by the number of pointers being found during marking. In the simple C malloc version, many pointers to the heap object will be found and added to the mark queue. The more nodes there are, the more pointers in the heap (to the heap), so the longer the mark time. In the GetM version, pointers are properly cleaned up and only the necessary live ones are found each cycle thus reducing marking time. The overall effect on the two M&S times is that they end up roughly the same. However, M&S time does not affect the total execution time of the simple C malloc version, as has already been shown.

## 8.2 GNU Standard Library C malloc

There are many implementations of C malloc in existence. *dlmalloc* and *ptmalloc2* are among the most commonly used because of their use in the main Linux kernel. The majority of Linux programs now use *ptmalloc2* or its derivative in *glibc*. However, *ptmalloc2* is specifically used for its threading support and has a complex *'arena'* system. It is not comparable to the IHGC, single-threaded system. Therefore, the single-threaded *dlmalloc* implementation is used for comparison here.

*dlmalloc* stands for 'Doug Lea's Memory Allocator' and is a variant of the dynamic storage allocation algorithm described by Knuth in the Art of Computer Programming.[17] The algorithm has been popular for a long time, as noted by Chang et al.[6] The algorithm allows for variable size chunks which are pre- and post-fixed by the size of the chunk. The algorithm also handles fragmentation by allocating larger chunks than requested if necessary and by occasionally compacting (*consolidating*) chunks when necessary. Additionally, *dlmalloc* has been used in many embedded applications. These are very similar operation features, overheads and use-cases to the IHGC and so is a fair basis for comparison.

In total, the *malloc* function of *dlmalloc* compiles to 1,159 RISC-V Integer-only, Compressed ISA instructions (when compiled with -Os to optimise for size and inlining has been allowed).[18] A copy of the assembly code for the *malloc* function is provided with this thesis. It is beyond the scope of this thesis to try to evaluate bounds on how many of these instructions may need to be executed in best, average or worst scenarios. However, it is apparent from the instructions that at

least 40 and possibly up to several thousand instructions may need to be executed per allocation.

It is clear, therefore, that the GetM instruction provided by the IHGC is capable of cutting hundreds of instructions (including many branches) and thus saving a lot of time. Furthermore, the *dlmalloc* implementation has a minimum allocation size of 8 bytes (by default) and an overhead of 2 words per allocation. In comparison, the IHGC has a minimum size of zero (with none of the normal risk associated with zero-sized objects) and an overhead of one word per allocation (in main memory). The IHGC's directory memory adds between 4 and 8 bytes per allocation (depending on system design) but these are considered offset against the removal of typical virtual memory hardware structures such as page tables. Nevertheless, even including the directory, the IHGC has much higher performance and lower overheads than *dlmalloc*.

This comparison is not quite representative though, since the *malloc* routine of *dlmalloc* does not initialise allocations to zero. The *calloc* routine must be considered to compare this functionality. The *calloc* routine sequentially clears memory using *memset* but otherwise is simply a wrapped call to *malloc*. However, *memset* clears memory sequentially which uses up processing time for the main application. In contrast, the IHGC clears memory in the background (in advance of an allocation request) thus hiding the time required. Thus the IHGC is significantly faster than the common standard C equivalent.

## 8.3 C# and Java New Object

A comparison to the C# and Java high-level "new object" instructions is almost impossible because of the extreme complexity of the two runtime environments. It is revealing, however, that the main GC file for C#'s Common Language Runtime available open source on GitHub ("src/gc/gc.cpp"[22]) is nearly 37,000 lines long and accompanied by numerous other files. There are numerous allocate functions and hundreds of backing functions for managing generational GC. In contrast, the IHGC module is 1,814 lines of Verilog including the complete interface and statistics gathering. The IHGC can also be relatively simply extended to include generational garbage collection, without the addition of thousands of lines of code, as will be shown by future work. It is apparent, therefore, that the IHGC is a significant simplification on state-of-the-art software implementations yet is able to retain high performance - real-time performance is better than C# is currently able to achieve.

Chapter Nine

# *Characteristics of GC performance*

Software GCs dynamically respond to mutator demand and are subject to interference from many parts of a system. This makes it difficult to drive software GC to consistent results. The IHGC is shielded from instruction fetches by the cache and the design is deterministic and relatively simple, so can be precisely driven from the CPU. Programs can be created with entirely predictable request patterns that allow characterisation of the IHGC's M&S, GetM, read, and write performance.

## 9.1 Measurements

Useful metrics were chosen to capture the IHGC's complete performance characteristics. The drive signals and metrics chosen are described in table 9.1.

Time was measured in clock cycles. Heappoint and Livesize were sampled during execution to provide an average value for a whole program. Response and blocking times were similarly averaged and hence, fractional quantities of clock cycles were recorded.

Data gathering is a separate module within the IHGC, operating in parallel and only requires integer hardware. The only exceptions are the Live and Dead handle counts, which do not slow the main state machine but are part of it.

## 9.2 Test Programs

Two kinds of test program were created. Firstly, assembly programs were used to precisely control particular IHGC inputs. Secondly, C programs representative of real programs were created (see Chapter 10).

The frequency of GetM, load, and store instructions can be precisely controlled in assembly programs. Six assembly code tests were developed and are described in table 9.2. Code files for the tests are attached with this thesis.

| Name | Driven? | Description |
|---|---|---|
| Read/Write/GetM requests | Yes | Total count during execution. |
| Read/Write pointer vs. value ratios | Yes | Calculated from program analysis. |
| GetM size per request | Yes | Size in words. |
| Livesize | Yes | Size in words. |
| Live handle count | Yes | Counted during each mark phase (may exceed number of allocated tuples). |
| Dead handle count | Yes | Counted during each sweep phase (cannot exceed number of allocated tuples). |
| Execution time | Yes | Total execution time (excl. initialisation). |
| Read/Write/GetM response time | No | Time between $doing\_action$ and $*_{ready}$ |
| Read/Write/GetM block time | No | Time between $*_{valid}$ and $doing\_action$ |
| GetM's blocked due to no handles count | No | Request blocked due to $nil\ free$ list. |
| GetM's blocked due to no space count | No | Request blocked due to no heap space. |
| Heappoint | No | Size in words. |
| Maximum heappoint | No | Size in words. |
| Directory & memory accesses count | No | Number of IHGC accesses to memory. |
| Mark & Sweep cycles count | No | Number of times $sweep_{end}$ is reached. |
| Out of memory count | No | OOM condition hit count (max. 1). |
| Initialisation time | No | Initialisation time of memory and directory. |

Table 9.1: Drive signals and metrics used to characterise the IHGC

## 9.3 Difficulties of evaluating results

Several problems were overcome to obtain fair results. Firstly, the IHGC initialisation stage blocks the first instruction fetch of the program for a long time. This blocking time occurs once per system reset and could be significantly optimised in a real system, so has been excluded from the results.

The test framework required *out* instructions to send the "Start" and "Finish" messages during each test. This is a constant time factor of each test program but could not easily be removed from the IHGC data. The time required has been left in the execution time measurement. Additionally, to minimise the proportion of execution time spent on start/end messages, each test was (where possible) executed for 100,000 iterations in order to give an unskewed average reading.

ASM Test (1) has two modes (Mode 0 and Mode 1), determined by the $retain$ parameter. However, Mode 1 results were not very useful because both the number of objects created and the

| No. | Compiler Inputs | Description |
|---|---|---|
| (1) | $size$, $frequency$, $iterations$, $retain$ | Allocates 1 tuple of $size$ words every 1 in $frequency$ instructions for $iterations$ number of repetitions. If $retain$ then the objects were retained in the stack. The object size and frequency were varied between tests. A wide range of object sizes were used with 3 fixed frequencies. |
| (2) | $size$, $frequency$, $iterations$ | As (1) but range of frequencies with 3 fixed object sizes and no $retain$ option. A slightly different program structure creates a different pattern of requests to the IHGC. |
| (3) | $frequency$, $mode$ $iterations$ | Loads (mode 0) or stores (mode 1) a value (*0xDEADBEEF*) every 1 in $frequency$ instructions for $iterations$ number of repetitions. |
| (4) | $frequency$, $mode$ $iterations$ | As (3) but stores/loads a pointer instead of a value. Pointer is pre-allocated and remains live throughout the program. |
| (5) | $frequency$, $iterations$ | Stores then loads a pointer or a value in each iteration. Ratio of pointers to values is 1 pointer for every $frequency$ values. Repeats for $iterations$ number of repetitions. |
| (6) | $frequency$, $iterations$ | As (5) but always uses a value instead of a pointer. This was used as a control test to determine what was GC behaviour and what was caused by the program. |

Table 9.2: Assembly code test programs

object size changed between each test (since retaining objects meant they remained live so fewer objects could be created).

Lastly, in order to fairly represent the full range of values tested, the majority of graphs are plotted on log X and log Y scales. The graphs shown confirm expected results or showed something unexpected - a copy of all graphs plotted are provided in separate files with this thesis.

## 9.4 Results

The following sub-sections present the ASM test results grouped by characteristic. The list below specifies what is shown by each of the possible result graphs. Each test was run with up to three variants (plotted as separate lines). Tests were created using a code file which was then compiled to separate executables for each value of each variable. The hardware is deterministic, so all three repeats of a test gave identical results. (This is also evidence that the hardware reset correctly

between each test.)

**Avg. allocation size per clock cycle** The total size of all allocations made (including single stack tuple but excluding program tuple) divided by the total execution time of the program.

**Execution time** The total execution time of the program in units of clock cycles (the FPGA used 100MHz clock speed).

**Avg. GetM blocking time** The total number of clock cycles the GC blocked GetM requests divided by the number of GetM requests (excluding the program object).

**GetM block: No handles count** The number of times a GetM request was initially blocked because the IHGC was out of handles ($free$ register was $nil$). This is not mutually exclusive with the "No space count".

**GetM block: No space count** The number of times a GetM request was initially blocked because the IHGC was out of space ($heappiont + getm_{size} + 1 >= MAIN\_MEMORY\_SIZE$). This is not mutually exclusive with the "No handle count".

**Avg. heap-point** The average of all samples of the heap-point taken at 10,000 clock cycle intervals during execution (excludes IHGC initialisation).

**Live/dead handles count** The total number of live and dead handles found during all mark and sweep cycles during execution. These indicate if the IHGC is functioning correctly for a given program.

**Avg. live/dead handles per M&S cycle** The average number of live and dead handles found per mark and sweep cycle.

**Avg. live-size** The average of all samples of the live-size taken at 10,000 clock cycle intervals during execution (excludes IHGC initialisation).

**Avg. Mark-Sweep cycle time** The average number of clock cycles taken to complete a mark-sweep cycle during execution (excludes IHGC initialisation).

**Avg. read/write blocking time** The total number of clock cycles the GC blocked read/write requests divided by the number of read/write requests (includes instruction fetches which occur exactly once per execution to fill the cache).

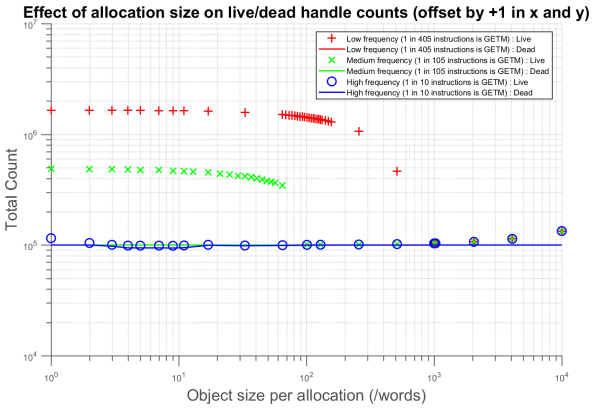



Figure 9.1: Total live and dead handle counts for ASM Test (1), objects not retained

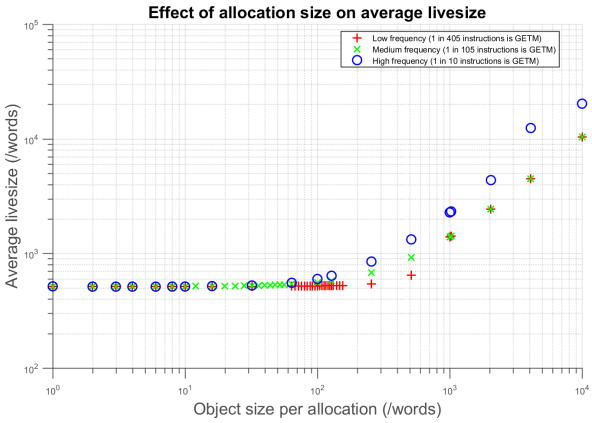



Figure 9.2: Average livesize for ASM Test (1), objects not retained

**Avg. read/write response time** The total number of clock cycles the GC spent processing read-/write requests divided by the number of read/write requests. This is not constant due to the difference between read and write and between writing pointers and values.

**Heap-point overhead** The average heap-point divided by the average livesize, as a percentage.

**Max. heap-point overhead** The maximum heap-point divided by the average livesize, as a percentage

Graphs of directory and memory accesses provided no further insight and initialisation time and GetM response time are constant values so were not plotted.

In some cases graphs have been plotted with an offset of $+1$ in the x or y directions. This is so that values at zero aren't lost on the logarithmic axes.

### 9.4.1 Correctness

ASM Test (1) allocates one object per iteration and in all cases was run with 100,000 iterations. Figure 9.1 shows, as expected, that the live handle count is always greater than or equal to 100,000. A handle may be counted as live more than once if it is live in more than one mark cycle. A handle is counted as dead when it is cleared during a sweep cycle and can only be dead once per allocation. The graph shows this behaviour as expected. In some tests, a lower dead count is recorded as not all tuples were swept before the test ended. Overall, the graph shows that the IHGC does not lose handles and it does not free them twice.

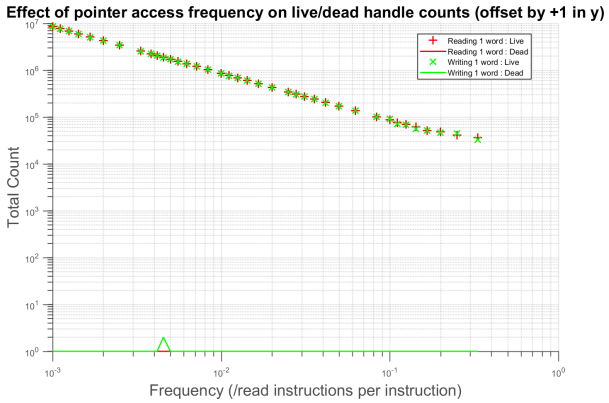



Figure 9.3: Total live and dead handle counts for ASM Test (4)

The minimum livesize in this implementation is the fixed program size of 512 words. Figure 9.2 shows that the average livesize is always at least 512 words and grows with larger or more frequently allocated objects.

Figure 9.3 shows that at frequency *1 in 220*, the dead handle count for the writing test spikes to a value of 1. This is not an error. Simulation confirms that the IHGC sweeps the test's objects between the main code completing and the test issuing the *ebreak* termination instruction.

### 9.4.2 Responsiveness

Figure 9.4 shows that for the majority of requests during ASM Test (1) (Mode 0), enough handles were immediately available. However, for high allocation frequencies and small, non-zero object sizes, the $free$ list ran out. For zero-sized objects, M&S can free handles very quickly as very little memory has to be scanned. However, for larger objects, more sweeping takes place resulting in slower M&S. For object sizes 2 to 6, handles ran out approximately 22 times. For sizes 8 and 10, handles ran out less often: 14 and 17 times respectively. This slight reduction, and then the return to zero OOH for larger objects, is explained by OOS starting to occur, shown by figure 9.5. OOS can occur with fewer than the maximum number of tuples so at least one handle is always free. A system designer will need to plan carefully the ratio of tuples to space.

Figure 9.5 also shows that the OOS counts converge for all three frequencies. In other words, the GC blocking time becomes the dominant performance factor meaning it doesn't matter how fast the program tries to allocate memory. Objects in ASM Test (1) are not scanned during marking since they do not contain pointers (they are not *deep*). In this test, therefore, the slow part of M&S is the $sweep_{zero}$ phase.

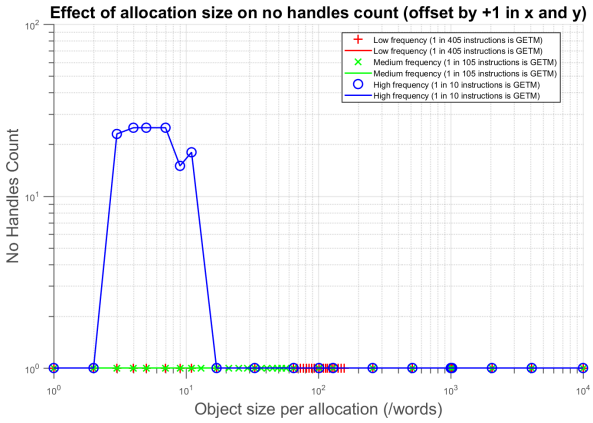



Figure 9.4: Total GetM blocked by OOH count for ASM Test (1), objects not retained

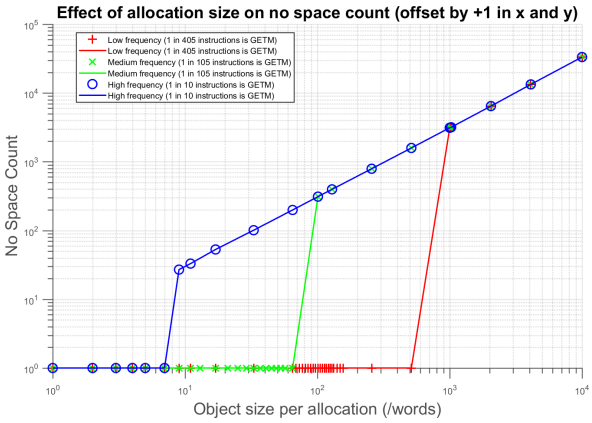



Figure 9.5: Total GetM blocked by OOS count for ASM Test (1), objects not retained

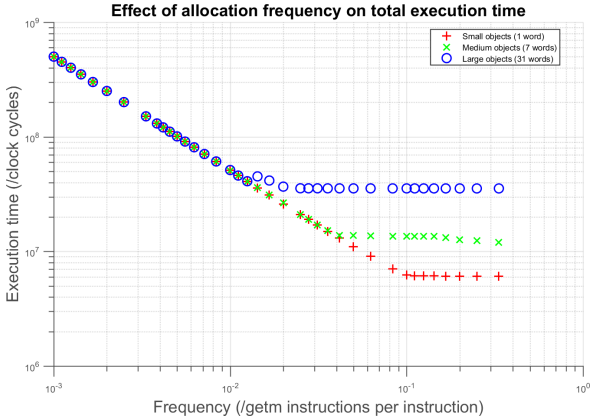



Figure 9.6: Average execution time for ASM Test (2)

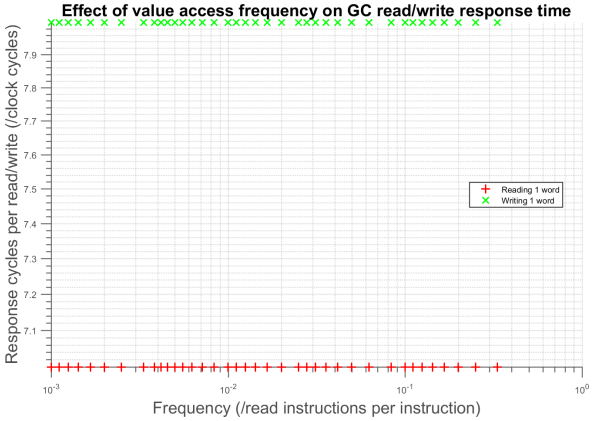



Figure 9.7: Average read/write response time for ASM Test (3)

However, the OOS counts shown in figure 9.5 grow linearly with object size suggesting that the IHGC M&S is predictable. The sweep behaviour results in lots of space suddenly becoming available at the end of each cycle and so many new objects can be allocated without blocking. The OOS range is 0.027% to 30% of the total 100,000 objects allocated during the tests, even though with object sizes of 10,000 words all of main memory would be consumed by just 6 objects. This suggests that the GC is doing an effective job at keeping up with the program.

Figure 9.6 shows the relationship between object size, allocation frequency and execution time for ASM Test (2). In all tests, 100,000 objects were allocated. At low allocation frequency, the total execution time is determined by the program alone. At higher frequencies larger objects take longer to sweep and so the total execution time is longer. Smaller objects take less time to sweep so the system never runs out of space but does run out of handles.

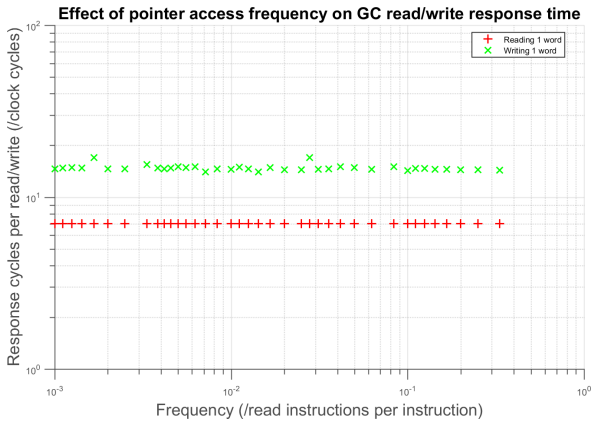



Figure 9.8: Average read/write response time for ASM Test (4)

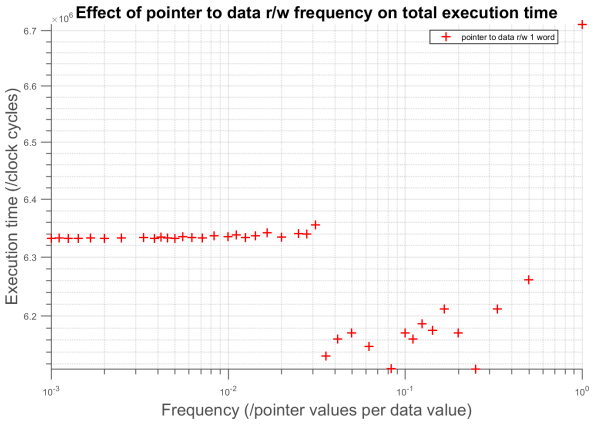



Figure 9.9: Average execution time for ASM Test (5)

Figures 9.7 and 9.8 show that reading a value or pointer on average takes the proven upper bound time of 7 clock cycles. This suggests that requesting the word that the IHGC is currently sweeping is a very rare case. Writing a value takes on average the upper bound time of 8 clock cycles where as writing a pointer takes, on average, 14.67 clock cycles. This is biased towards the upper bound which suggests that most often when writing a pointer to memory, the related tuple is not already marked and marking is ongoing (since the read tests have established that the alternative route is very rare).

ASM Test (5) writes a pointer one in every so many value writes. ASM Test (6) was designed as a control test for ASM Test (5) to distinguish program effects from IHGC function. It uses exactly the same program structure but a value is written instead of a pointer. Figure 9.10 shows consistency and that higher frequency of writing results in a lower execution time, as expected. However, figure 9.9 shows less consistency and at the highest frequency (every write is a pointer), the execution time is significantly increased. This suggests that writing pointers to memory introduces unexpected effects that software developers would need to be made aware of this in order to optimise their code. Furthermore, it may be beneficial for future work to look at compiler optimisations that minimise pointer stores.

The results around the turning points for low and medium frequencies shown in figure 9.11 are unexpected. The graphs show that for the same object size near the turning points, allocating at a lower frequency results in faster execution time. Figure 9.6 shows the same effect occurring in ASM Test (2) for a given object size (the bump for large objects is the most visible). This result can be explained by the interaction between GetM and the internal Mark-Sweep functionality.

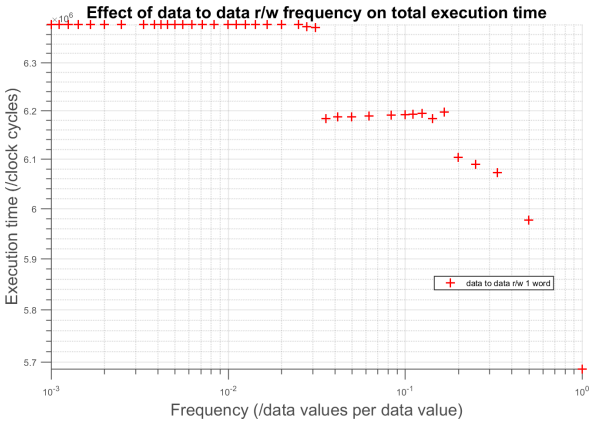



Figure 9.10: Average execution time for ASM Test (6)

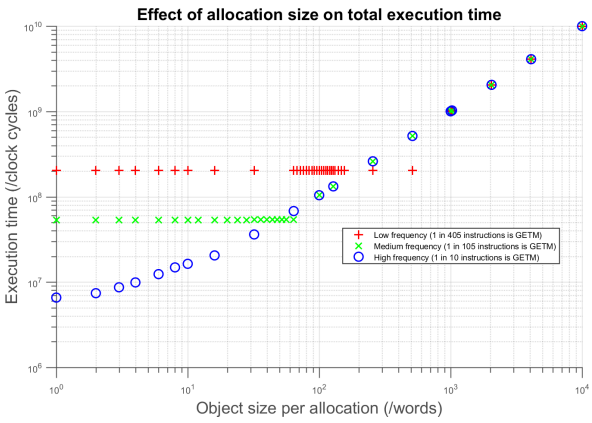



Figure 9.11: Average execution time for ASM Test (1), objects not retained

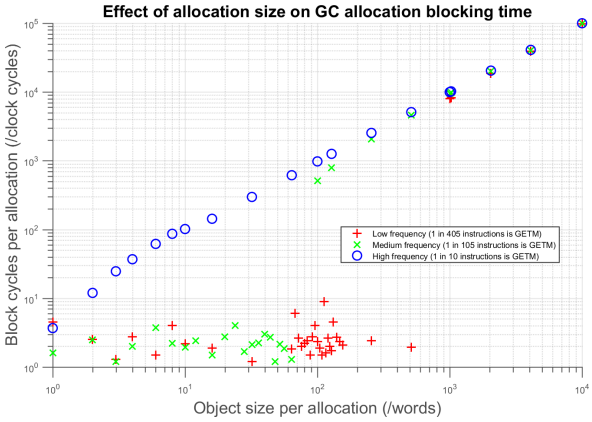



Figure 9.12: Average GetM blocking time for ASM Test (1), objects not retained

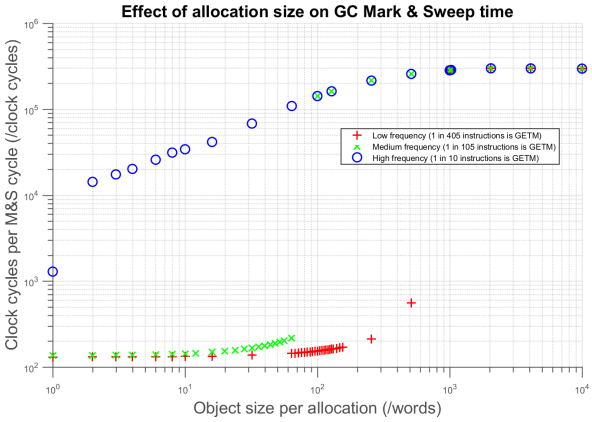



Figure 9.13: Average M&S time for ASM Test (1), objects not retained

With small object sizes, M&S cycles complete faster than memory is allocated meaning the system doesn't run out of space and so the program runs at the fastest speed possible (determined by the allocation frequency). At large object sizes, the sweep phase does not complete before all free memory has been allocated. This means a request has to be blocked for another entire M&S cycle before space becomes available (because $heappoint$ is updated in $sweep_{end}$). This creates a large blocking time (approximately the M&S time - see figure 9.12) which dwarfs time between allocation requests and so the M&S time determines the program's total execution time.

Within the turning region, the interaction between allocation frequency and M&S cycles must be inspected more carefully. At high frequency, the reasoning from the previous paragraph applies. However, at lower frequency for the same object size, the IHGC is able to reach $sweep_{end}$ before all available memory runs out. In other words, it is able to compact new objects faster than they

are being allocated. Thus the next mark phase starts, detects lots of dead objects, and sweep then clears lots of space (again, before memory runs out of space). If the IHGC can reach $sweep_{end}$ before available memory runs out then it does not need to block any GetM requests. Since no allocations are blocked, the program continues to execute at its maximum speed. Thus allocating less frequently can decrease overall execution time.

From this we can conclude the following: The time taken to wait for a M&S to finish immediately might be quite small (100s of clock cycles) compared to if the system runs out of space and the program has to wait for an entire M&S to finish later (100,000s of clock cycles). Overall, it is clear that software developers will need to consider performance of the underlying GC in order to optimise their code. Pushing the IHGC to allocate as fast as possible at one time may drastically slow down allocations at a future time, which could significantly reduce overall performance. Future work should investigate ways to mitigate this effect, for example, by making freshly zeroed-out memory available immediately rather than delaying to $sweep_{end}$ or by delaying starting M&S cycles until they will have most effect (e.g. when only a chosen percentage of memory is free).

Collectively these graphs show that for given directory and memory sizes, it is possible to estimate the relative performance of programs by knowing their average allocation frequencies and sizes, and their read/write frequencies and types. Future work would test more variations of memory and directory sizes, to form a general expression for the relationships.

### 9.4.3 Overheads

Figure 9.14 shows that, for ASM Test (1) (Mode 0), at low and medium allocation frequency, the maximum heappoint is 100% of livesize, meaning there is no significant overhead. At high frequency or with larger objects, the maximum heappoint is capped by the size of main memory (65,536 words). At larger object sizes, the average livesize increases (as objects remain live for longer while they are marked and swept) and so the maximum heappoint is proportionally closer to livesize. The jumps in maximum heappoint also relate to the transition points where the IHGC can no longer keep up with allocations.

Figure 9.15 shows, for ASM Test (2), that at high allocation frequencies the blocking time follows a smooth trend. This is because the blocking time is dominated by waiting for the few M&S cycles that block a small number of allocations - the small variations from allocations that aren't blocked are absorbed and hidden. At low frequencies, the graph shows no trend within or

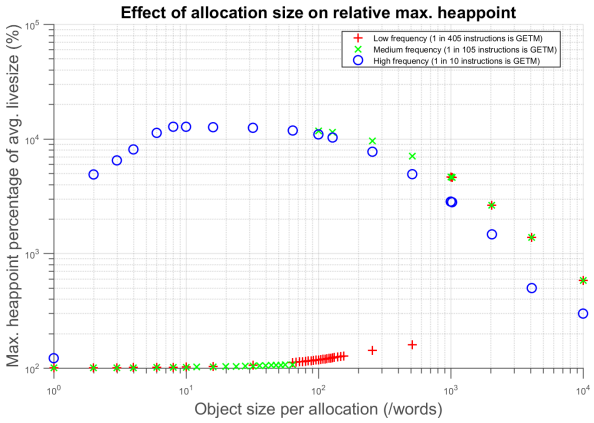



Figure 9.14: Maximum heappoint overhead for ASM Test (1), objects not retained

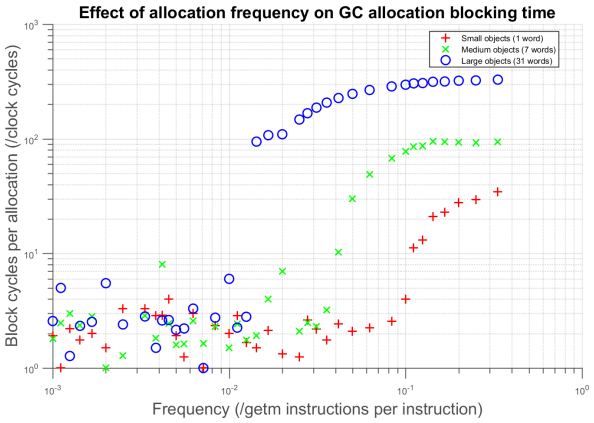



Figure 9.15: Average GetM blocking time for ASM Test (2)

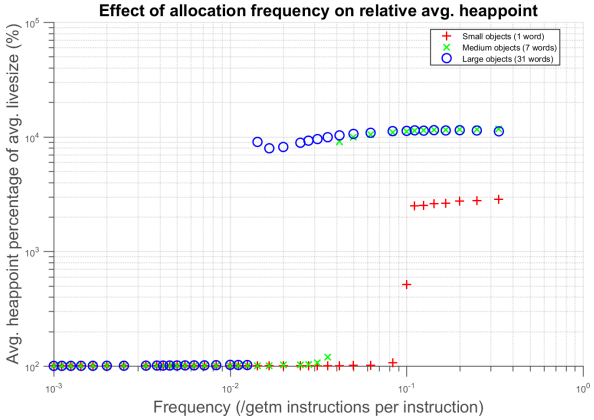



Figure 9.16: Average heappoint overhead for ASM Test (2)

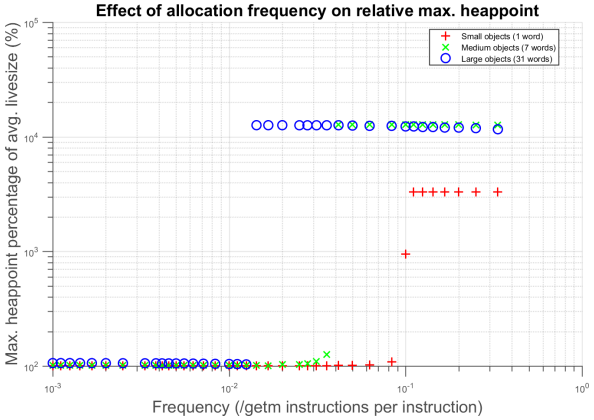



Figure 9.17: Maximum heappoint overhead for ASM Test (2)

between object sizes. This is expected as the blocking time for a request is determined by the time to complete any ongoing M&S state. The upper and lower bounds on this time (see Appendix D) range between 1 and 11 clock cycles. Requests occur (effectively) randomly with respect to M&S states and so the average blocking time (when not OOH or OOS) is tightly bounded but random between those bounds.

Figures 9.16 and 9.17 show, for ASM Test (2), that for medium and large objects at high frequency, the heap overhead is capped by the maximum heap size. For small objects, the overhead is capped by the system running out of handles. It is interesting to note that the average heappoint overhead shows a bump in the largest object curve next to the transition point. The cause of this is explained in the previous subsection. The maximum heappoint overhead does not show this bump, as is expected because it is a single upper bound value.

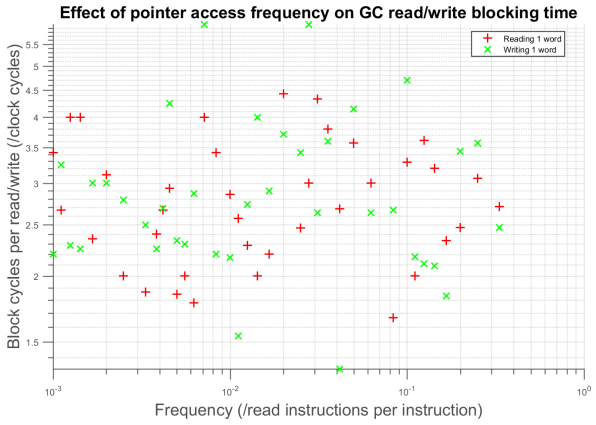



Figure 9.18: Average read/write blocking time for ASM Test (4)

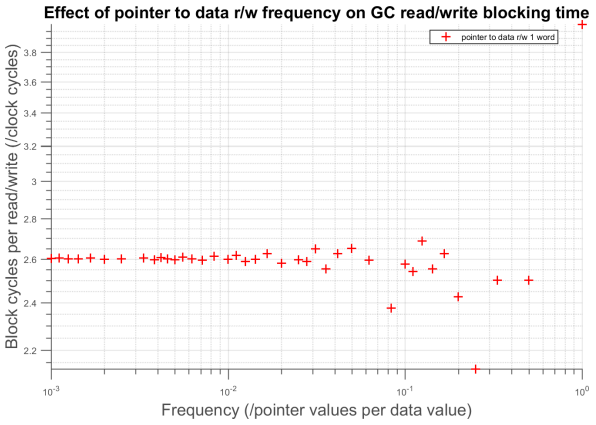



Figure 9.19: Average read/write blocking time for ASM Test (5)

However, it is also interesting to note that the maximum heappoint for small objects is the same as the average. This is further evidence that the limiting factor is the number of handles (since clearly more space was available) but also that the style of the tests programs means that the IHGC is put under maximum stress. Despite this taxing workload, it is capable of keeping up. For objects of 31 words or less (which is typical of current C# programs)[1], 1 in 80 instructions can be an allocation and the IHGC will not block for more than 11 clock cycles. This is significantly higher performance than the best-case sustained workload supported by the latest .NET Framework (developed by Microsoft), which requires hundreds of instructions between (or during) allocations.

The read/write blocking time for ASM Test (3) and for ASM Test (4) (figure 9.18) show no correlation to frequency. Additionally, the M&S cycles do not appear to affect read/write time - only the blocking time taken to complete any single M&S state matters.

However, figures 9.19 and 9.20 show that writing pointers more often than values does affect average blocking time in an apparently chaotic way. This suggests that writing pointers causes the IHGC to take a less consistent route through its M&S states resulting in requests randomly interrupting a greater variety of M&S states, thus causing the blocking time to vary more. Further work will be required to prove this suggestion.

[1]This is based off the author's experience from an ongoing study looking at object sizes in modern high-level software. It is hoped that this study will be completed and made available later in 2017.

[1]This thesis is an independent publication and is neither affiliated with, nor authorized, sponsored, or approved by, Microsoft Corporation.

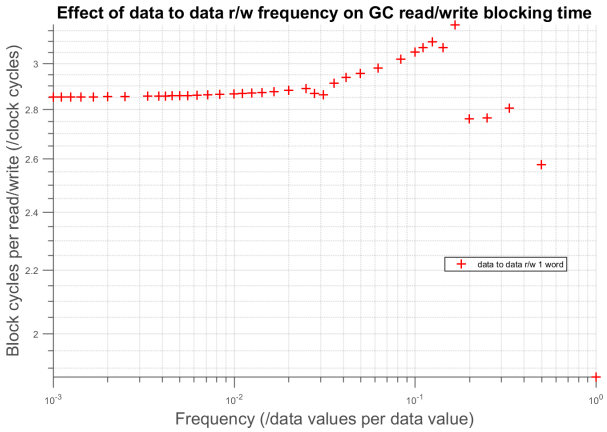


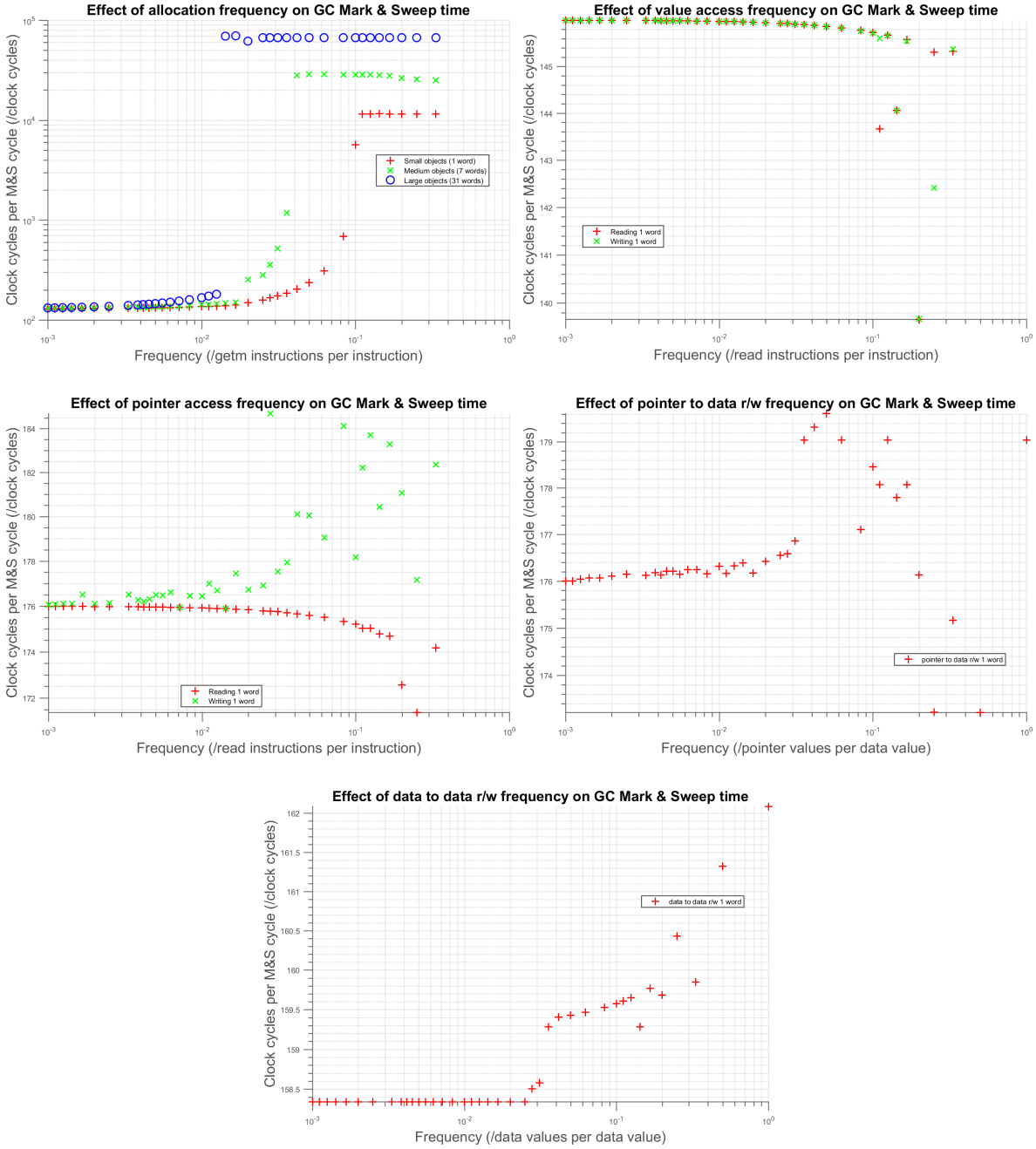

Figure 9.20: Average read/write blocking time for ASM Test (6)

### 9.4.4 Scalability

Figure 9.21 shows the average mark and sweep cycle times for ASM Tests (2) to (6). They suggest that the frequency of reads, writes and accessing pointers influence mark and sweep time by up to a factor of 1.5. However, the results from ASM Test (2) show that M&S time is most affected by allocation size and frequency, up to almost three orders of magnitude if the system runs out of memory. This suggests that providing live statistics to programs may allow them to cooperate better with the IHGC. However, prior work has suggested that the cost versus benefit of responsive applications is not worthwhile when compared to conventional approaches to optimising software.

A few tests were conducted that deliberately ran the system out of memory. The results of these tests are not shown in the graphs as they would significantly and misleadingly skew the results. The OOM tests showed that the OOM condition is entirely predictable based on the number of handles and amount of memory and the allocations made by the programs. The exact moment the system would hit the OOM error and stop could be predicted to the exact instruction and clock cycle for programs which went directly to OOM (i.e. did no dynamic allocation activity before allocating enough handles or memory to reach OOM).

Figure 9.21: Average M&S time for ASM Tests (2) to (6)



Chapter Ten

# *Performance of typical programs*

This chapter discusses three programs that were created to be representative of some typical C programs. The first program created linked lists of variable length and either discarded them or stored them in a circular buffer of 1024 items. The second program has already been presented in Chapter 8. The third program created binary trees of varying depths between various maximum and minimum points. This last program was adapted from the binary trees toy benchmark from The Computer Language Benchmark's Game.[1][9]

Figures 10.1 and 10.2 show graphs of interesting results from C Tests (1) and (3). Figure 8.2 shows the results of C Test (2). The linked list test shows that when linked lists were discarded soon after being created, the GC blocking time (and thus overall execution time) grows nearly linearly until the final few tests when the number of available handles started to run out. When the linked lists were retained in a circular buffer, the execution time initially grew at a similar rate to the discarded lists test. However, it starts to run out of handles much sooner and so allocation blocking time grew rapidly. Results for larger retained lists could not be taken as the number of objects required would have been greater than the number available.

Both the discarded and retained tests created the same number of objects, so as can be seen, the dead handle counts were identical. However, in the case of the retained objects, the objects remained live across more M&S cycles so the live object count increases exponentially. However, the execution time would not be able to continue to grow exponentially because of the proven upper bounds on M&S time and the limited number of handles in the system. It is interesting to note that for the tests which contained similar numbers of live objects (at any given moment in the program) had similar M&S times (and overall execution times). This suggests that in future, adding more handles to the system would allow larger retained linked lists and reduce the execution time as OOH

[1] A comparison to The CLBG's results is not made since it is not of interest. The binary trees program is used here only as a basis for an example of some typical program work.

would not be reached so quickly. In other words, it would simply stretch the graphs shown to larger lists.

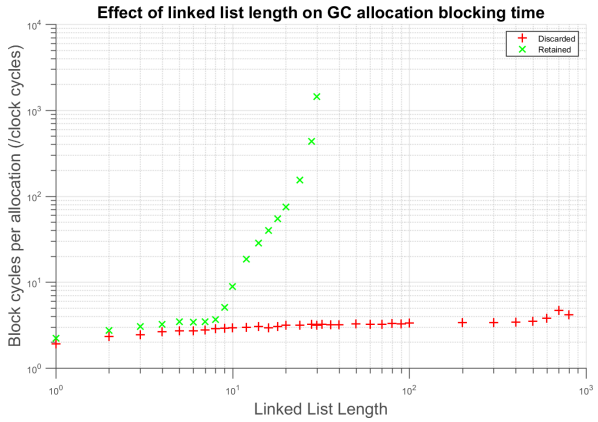



Figure 10.1: Average blocking time per allocation for C Test (1)

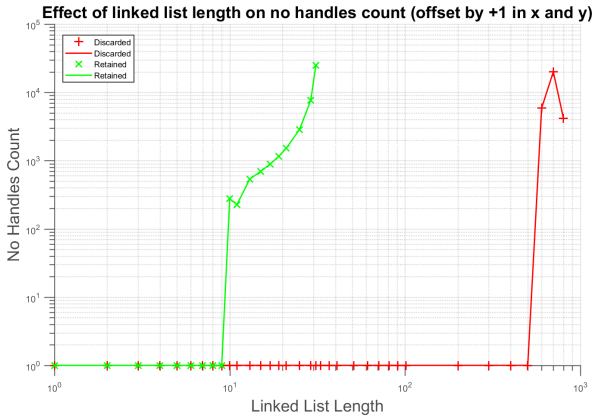
C Test (3) shows the same result as C Test (2) in that H/W Malloc was faster than Simple C Malloc. The average heappoint overhead for simple C malloc declines as the tree depth increases because the tests last longer and so the average livesize approaches its actual value thus making the overhead appear less. However, it is possible to see that for H/W Malloc the heappoint overhead is approximately 10% to 15% of the average livesize. This is the case so long as the IHGC can keep up with the program. However, if the IHGC starts to hit OOS conditions then the overhead reaches the maximum of 12,800% (for this amount of main memory). As such, the system fails criterion 8.4 because it is not always met. Future work should investigate how to make free memory available as soon as possible, even during a sweep cycle, so as to reduce the effect of blocking heappoint update to $sweep_{end}$.

Additionally, criterion 8.3 is not met by C Test (1) or (2) because C programs store pointers on the stack and the stack is not cleaned up after a function exits. As a result, memory which C considers to be dead, is detected as live by the IHGC and so the livesize and heappoint are kept much larger than in the original C programs. A solution to this to allocate a new object for each stack frame, which offers other potential benefits for exception handling.

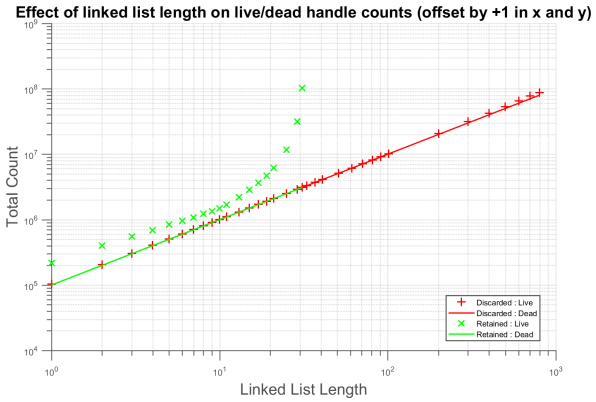
The typical programs shown behave as would be predicted from the characterisation of the IHGC in Chapter 9.

Figure 10.2: Results for C Test (1) and C Test (3)

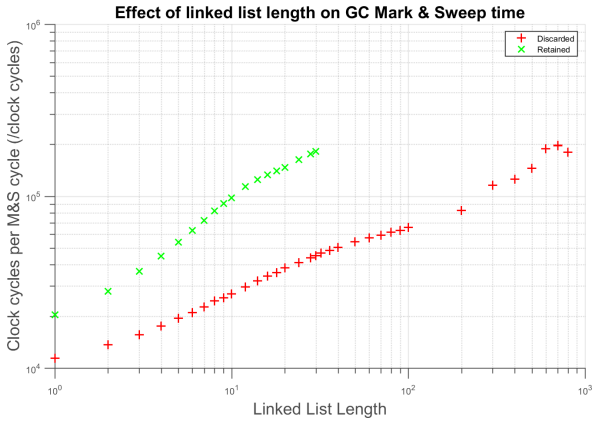
(a) Total GetM blocked by OOH count for C Test (1)



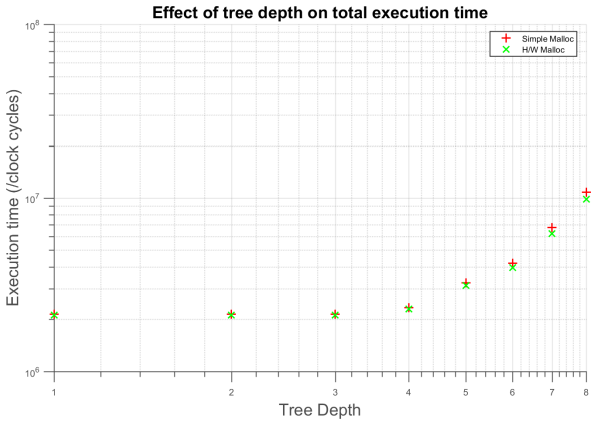

(b) Total live and dead handle counts for C Test (1)



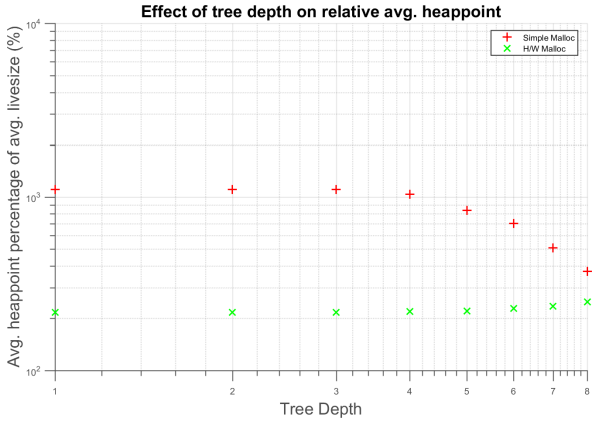

(c) Average M&S time for C Test (1)



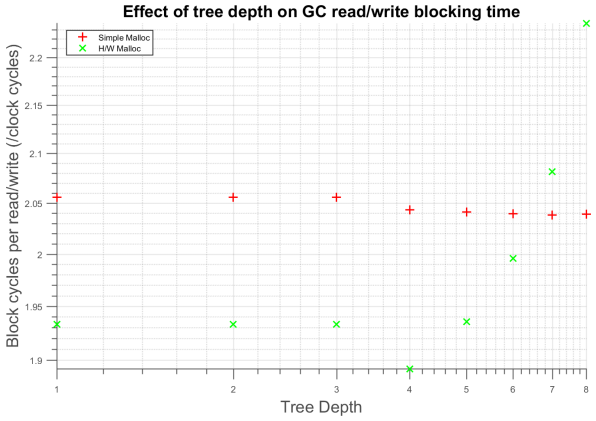

(d) Average execution time for C Test (3)




(e) Average heappoint overhead for C Test (3)




(f) Average read/write blocking time for C Test (3)



Chapter Eleven

# *Outcomes and Conclusions*

This thesis has shown that a general purpose hardware garbage collector can be created and is viable for implementation in current commercial and cutting edge technology standards. The Integrated Hardware Garbage Collector design improves upon prior efforts in this field by being more flexible, configurable and having greater scope for future work. Of the 51 criteria for feasibility, all but 3 have been met. Of the 3 which were not met, 13.2 is reasonably not met since the majority of modern systems contain at least a minimal instruction cache, and 8.3 and 8.4 can be minimised (and probably met) by good system design, improved compiler support and future work on optimisations.

Software GC comes with big trade-offs, and large, unavoidable overheads that make it unsuitable for real-time or embedded systems. This thesis has demonstrated that the IHGC has real-time performance and can be made small enough for embedded devices. As a result, for the first, time languages such as C# and Java could be used for programming such devices without restriction on language features or usage. This is a significant step forward for hardware and software development, as high level languages significantly improve security, reliability and implementation time.

Traditional hardware memory models offer no tangible security to the programmer. Security is either enforced by an operating system or ignored altogether. With the increasingly large IoT market, where devices execute multiple threads and machine code may be downloaded across the internet, providing hardware guarantees is becoming a bigger concern. This thesis has shown the IHGC can be integrated with any of the ISAs currently in popular use and as such, could be integrated with the processor in the majority of current devices. Furthermore, this thesis has shown that the software changes required of well-written, C/C++ or high-level language programs would be minimal and generally apply only to the standard library or compiler. As such, it is viable to rapidly bring the necessary hardware guarantees to future IoT, automotive and similar devices.

When compared to current software memory management, even the simplest equivalent C implementation is slower than using the IHGC. Although it is possible to not zero-out memory using

standard C malloc functions, which saves execution time, it is becoming increasingly apparent that zeroing out memory before use is a necessary practice in order to avoid bugs. Furthermore, the speed of standard malloc and free functions is between 1 and 3 orders of magnitude slower than the single GetM instruction of the IHGC.

This thesis has presented and proven feasible the first general purpose, fully concurrent garbage collector which can be implemented in hardware and integrated with an existing ISA.

Chapter Twelve

# *Future Work*

There are a significant number of ideas to pursue in future work. This chapter presents ideas in three categories: ideas which follow on directly from this thesis, hardware research ideas, and software research ideas. The ideas are presented in no particular order and regardless of scale or scope, each is significant and worthy of investigation.

## 12.1 Continuation of work

- The author will be starting a PhD in September 2017, initially looking at formal verification of IHGC design.
- Bug fix the design: A particular pathological program could destroy a linked list from the top downwards, causing it to be freed too early. The IHGC should iterate over the CPU registers until all pointers are marked in order to avoid this early-freeing case. It is probably possible to do this in a way which will not affect the WCET of Mark and Sweep.
- Bug fix implementation: The main state machine should still be able to progress even with simultaneous GetM/R/W requests. This can be achieved by adding an extra condition to the interruption logic.
- The modified version of the RISC-V Compressed ISA specification should be written up and published.
- The implementation should be optimised to overlap memory and directory requests wherever possible. This will reduce the time required for some of the main states.
- The implementation should be optimised to reduce the number of clock cycles lost to state transitions

- The implementation should be extended to allow the program object to be dynamically allocated, changed and written to.
- There is no existing suite of programs for benchmarking GCs. It would be useful to define a set of metrics for GC analysis and create a set of portable benchmarks.
- A range of main memory and directory sizes should be tested and the final, large set of data analysed to create general characteristic equations for the IHGC. These could then be used to predict performance of a program by static analysis of the programs use of memory.
- Proper mechanisms for error handling and recovery from errors should be investigated.
- The current design relies on a proportion of memory cycles being unused by the CPU core. This is a problem when integrating with highly pipelined processors, such as applications scale processors. Methods to overcome this should be investigated, for example, dual ported memory.
- Dual ported memory and/or directory could allow simultaneous mark/sweep and read/write.
- There is a possible trade-off for the directory. It is feasible to put the directory into main memory and retain a smaller data cache in the IHGC. The effect of such a design is not yet understood and will need to be explored.
- Although this thesis has shown that caches can be placed on either side of the IHGC, the design of such caches and the effect they may have has not been explored.
- Compiler support for other languages e.g. C#, Java, Haskell.
- The reset mechanism could be optimised in ASIC design by creating custom reset circuitry.
- It would be interesting to synthesise and manufacture a chip at around the 160nm scale to see if full ASIC design creates any interesting differences to FPGA.

## 12.2 Hardware

- A method for making freed memory accessible before the heap-point is updated at the end of sweeping.

- Not starting the next mark or next sweep until some condition is met, for example, only 25% of memory left free or some estimate of the M&S work required reaches a minimum level.
- The current design forces all objects to have the same maximum size and size granularity. However, real systems typically have many small objects and a few very big ones. It would be useful to extend the current design to allow two sets of objects, some in a "large" category and others in a "small" category, where the bits of a pointer used for handle and offset are divided differently in the two categories.
- The IHGC is in an excellent position to provide fast memory copy and fast memory set functions. However, before implementing such features, it would be useful to know how often they could be used by real software (allowing for the fact that *memset(0)* of any new objects can be eliminated by the existing IHGC design).
- Generational garbage collection is a common optimisation that should be relatively easy to add to this design by simply tracking the age of objects in a small counter and leaving such objects marked.
- It may be useful to provide a predictive or multiple allocation interface in order to optimise software performance.
- For multi-core systems, multiple simultaneous allocations or reads/writes must be made possible
- In some contexts it may be useful to provide a mechanism for protecting objects, such as making them immutable or only accessible from a particular CPU state.
- The IHGC prevents conversion from values to pointers and vice-versa. This poses a problem for the majority of embedded device driver code, which relies on fixed addressing. A mechanism for dynamically allocating memory-mapped objects for devices and detecting those objects would be extremely useful and significantly simplify the software ecosystem.
- The IHGC design does not yet work with multi-processing/multi-threaded systems
- In a multi-process system, there is a risk that a malicious program attempts to consume all the system resources or slow down other processes in the system by rapidly allocating and destroying objects. A mechanism for detecting and handling such behaviour will be required.

- Support for high performance computing systems - there are undoubtedly numerous new possibilities that we have not thought of yet using this new design.
- Traditional paged memory creates virtual memory space by "paging" to non-volatile media such as hard disk drives. The IHGC could potentially know precisely all the memory associated with a process and so save it directly to disk (DMA) with no wasted space. In addition, it may be possible to provide compression and encryption in hardware in the stream between the media and the memory.
- The IHGC knows precisely what memory is in use. It may, therefore, be possible and beneficial to power down individual memory modules when they are not in use, thus saving a lot of power. This could have a big impact on battery life or energy consumption for large memory devices such as laptops and mobiles.
- It would be worth re-investigating multi-processing communication mechanisms to see if GC can be performed across a network of processors, thus allowing automatic detection and clean-up of deadlocked processes.
- This thesis has looked only at CPU. It would be interesting to apply the design to GPU and DSP style architectures.
- Per-allocation statistics/flags/trace information could be made available to software developer at runtime. This would allow detection of performance issues and program pinch points.
- Functional processing is a small but growing field, especially as languages such as Haskell become increasingly popular. Functional programs rely heavily on GC and so previous efforts to create a general purpose functional processor have been stuck at the research stages. The IHGC design could be the breakthrough required to produce a general purpose functional processor.

## 12.3 Software

There are far more ideas for software that will emerge from this new design than can currently be predicted. Presented here are some of the author's more exciting or radical ideas.

- Reduction of OS functionality/complexity/time/energy consumption: Current operating systems spend vast amounts of time, energy and code on memory management and security patching the traditional memory model. The IHGC could eliminate the majority of this code.
- Aside from in a few special cases, it has never before been possible to allocate memory during a processor interrupt routine. The IHGC operates independently from the CPU state and as such, does not care if the CPU is in an ordinary function or an interrupt routine. This means memory can be allocated during an interrupt. This is a small change with profound implications for software design. For example, a network switch will no longer need to pre-allocate memory for a circular buffer of packets and use complex mechanisms to handle receiving and forwarding the packets. Instead, memory can be allocated during the interrupt and passed around more easily, being discarded automatically. This could significantly reduce software complexity and improve performance. As another example, most device drivers rely on begin, interrupt and end functions, where begin and end are used primarily to allocate and clean up memory resources. These aspects could be eliminated.
- Combination with statically checked languages such as Rust. Such languages aim to provide compile-time guarantees but there are still some features that cannot be statically checked. Additionally, an attacker can use a different language to create machine code. Combining static checking with guaranteed runtime checks could produce a highly reliable and trustworthy environment for programmers.
- Software prompts to the IHGC to optimise behaviour, such as prompting when to start Mark-and-sweep cycles.
- Support for software destructors. Some languages, such as C#, allow the developer to define functions to be called when an object is going to be destroyed (freed). The IHGC could provide a mechanism (interrupt or queue) that allows software to execute such destructors before the IHGC frees an object.
- Software patching value to pointer conversions may be possible but would weaken the guarantees provided by the hardware.
- Compiler optimisations for example, having the compiler produce code which writes pointers to memory less often.

Appendix A

# *GC State Machine*

## A.1 Addresses and the Directory

Each word $w$ in memory or in a register has a flag $w_{ptr}$ and a word $w_{word}$. If $w_{word}$ is a pointer, indicated by $w_{ptr}$ being $true$, then:

$w_{handle}$ Upper bits: identifies a directory entry of a tuple

$w_{offset}$ Lower bits: identifies an address within a tuple

Each directory entry $d$ of a tuple has five components:

$d_{addr}$ the address of the tuple in memory

$d_{size}$ the size of the tuple (in words)

$d_{mark}$ the marking flag for garbage collection

$d_{deep}$ a flag to indicate whether the tuple contains pointers

$d_{list}$ a linked list of directory entries used during mark/sweep

For a 32-bit wordlength, one possible configuration would allow up to 65536 tuples each of size up to 65536 bytes and the directory would have 65536 entries each with 62 bits:

$$d_{addr}, d_{size}, d_{mark}, d_{deep}, d_{list} \leftarrow 30, 14, 1, 1, 16$$

## A.2 Garbage collector variables

| | |
|---|---|
| $mem$ | represents the memory |
| $initreg$ | the index of the next CPU register to mark (initialised at reset to 0) |
| $current$ | points to list of directory entries currently being scanned |
| $next$ | points to a list of directory entries to be scanned next |
| $free$ | points to a list of free directory entries |
| $tuple$ | is the tuple being processed |
| $size$ | is the number of words in the tuple being processed |
| $index$ | is the offset of a word within the tuple being processed |
| $livesize$ | is the total size of the data that has been marked |
| $heappoint$ | is the highest location in the heap |
| $src$ | is the memory address from which a tuple is copied during sweeping |
| $dest$ | is the memory address to which a tuple is copied during sweeping |
| $oomcount$ | whether the GC has already reached $sweep_{end}$ while out of memory |
| $error_{oom}$ | whether the GC out of memory and unable to service the current Get Memory request |

## A.3 Memory access variables

| | |
|---|---|
| $buffer$ | holds a word being read from or written to memory |
| $rwindex$ | is the offset of a word being read from or written to memory |
| $pointer$ | is a pointer used to access a tuple |
| $offset$ | is a word offset used to access a word within a tuple |

## A.4 CPU Request/Access variables

| | |
|---|---|
| $getm_{waiting}$ | indicates if a Get Memory request is waiting or not |
| $getm_{space}$ | the size of memory to allocate (valid iff $getm_waiting$ is $true$) |
| $cpuhan_n$ | the handle value of the nth CPU register |
| $cpuptr_n$ | the ptr flag of the nth CPU register |
| $cpuregs_{available}$ | whether the CPU registers are available for access or not |
| $cpuregs_{count}$ | the number of CPU registers |

## A.5 Marking

$mark_{init}$: if $cpuregs_{available}$
then
{ if $cpuptr_{initreg} \wedge cpuhan_{initreg} \neq nil$
then

```
            { dir[cpuhan_initreg]_list ← nil
            & if ¬dir[cpuhan_initreg]_mark
              then dir[cpuhan_initreg]_mark, livesize ← true, livesize + dir[cpuhan_initreg]_size + 1
              else skip
            & if dir[cpuhan_initreg]_deep
              then next ← cpuhan_initreg
              else skip
            }
            else skip
          & state ← mark_next
          }
          else skip

mark_scan: if index ≤ size
           then
             if mem[src + index]_ptr ∧ (mem[src + index]_handle ≠ nil)
             then
             { tuple, index ← mem[src + index]_handle, index + 1
             & state ← mark_add
             }
             else index ← index + 1
           else
             if current = nil
             then state ← mark_next
             else
             { src, size, index ← dir[current]_addr, dir[current]_size, 1
             & current ← dir[current]_list
             }

mark_add :{ if ¬dir[tuple]_mark
            then
            { dir[tuple]_mark, livesize ← true, livesize + dir[tuple]_size + 1
            & if dir[tuple]_deep
              then dir[tuple]_list, next ← next, tuple
              else skip
            }
            else skip
          & state ← mark_scan
          }

mark_next : if next = nil
            then
              if initreg < cpuregs_count
```

```
        then state, initreg ← mark_init, initreg + 1
        else
        { dest, src, tuple ← 0, 0, mem[0]_handle
        & state, initreg ← sweep_scan, 0
        }
    else
    { src, size, index ← dir[next]_addr, dir[next]_size, 1
    & current, next, state ← dir[next]_list, nil, mark_scan
    }
```

## A.6 Sweeping

```
sweep_scan: val size, nsrc = dir[tuple]_size, src + dir[tuple]_size + 1 in
    if src = heappoint
    then heappoint, livesize, state ← dest, 0, sweep_end
    else
        if dir[tuple]_mark
        then
            if src = dest
            then
                if nsrc = heappoint
                then livesize, state ← 0, sweep_end
                else src, dest, dir[tuple]_mark, tuple ← nsrc, nsrc, false, mem[nsrc]_handle
            else dir[tuple]_mark, index, state ← false, 0, sweep_read
        else
        { dir[tuple]_list, free ← free, tuple
        & if (src + size) < livesize
            then src, tuple ← nsrc, mem[nsrc]_handle
            else
            if src < livesize
            then index, state ← livesize − src, sweep_zero
            else index, state ← 0, sweep_zero
        }

sweep_read: if index <= size
    then
    { copy_word ← mem[src + index]_word
    & copy_ptr ← mem[src + index]_ptr
    & if (src + index) < livesize
        then state ← sweep_write
        else state ← sweep_clear
    }
    else
    { dir[tuple]_addr, dest ← dest, dest + size + 1
    & if (src + size + 1) = heappoint
```

then $heappoint, livesize, state \leftarrow dest + size + 1, 0, sweep_{end}$
else
{ $src, tuple \leftarrow src + size + 1, mem[src + size + 1]_{handle}$
& $state \leftarrow sweep_{scan}$
}
}

$sweep_{clear}$: { $mem[src + index]_{word}, mem[src + index]_{ptr} \leftarrow 0, false$
& $state \leftarrow sweep_{write}$
}

$sweep_{write}$: { $mem[dest + index]_{word} \leftarrow copy_{word}$
& $mem[dest + index]_{ptr} \leftarrow copy_{ptr}$
& $index, state \leftarrow index + 1, sweep_{read}$
}

$sweep_{zero}$: if $index <= size$
then
{ $mem[src + index]_{word} \leftarrow 0$
& $mem[src + index]_{ptr} \leftarrow false$
& $index \leftarrow index + 1$
}
else
if $(src + size + 1) = heappoint$
then $heappoint, livesize, state \leftarrow dest, 0, sweep_{end}$
else
{ $src, tuple \leftarrow src + size + 1, mem[src + size + 1]_{handle}$
& $state \leftarrow sweep_{scan}$
}

$sweep_{end}$: if $getm_{waiting} \wedge free = nil \wedge heappoint + getm_{space} + 1 >= heappoint_{max}$
then
if $oomcount = 1$
then $error_{oom} \leftarrow 1$
else $state, oomcount \leftarrow mark_{init}, 1$
else $state, oomcount \leftarrow mark_{init}, 0$

## A.7 Memory allocation and access

$getmem$: { $dir[free]_{addr}, dir[free]_{size} \leftarrow heappoint, space$
& $dir[free]_{mark}, dir[free]_{deep} \leftarrow true, false$
& $heappoint, livesize \leftarrow heappoint + space + 1, livesize + space + 1$
& $mem[heappoint]_{handle}, mem[heappoint]_{ptr} \leftarrow free, false$
& $areg_{handle}, areg_{ptr}, free \leftarrow free, true, dir[free]$

```
        & oomcount ← 0
        }


read: val rwindex = pointer_offset + offset + 1 in
      if (pointer_handle = tuple) ∧ (rwindex = index)∧
        ((state = sweep_clear) ∨ (state = sweep_write))
      then buffer_word, buffer_ptr ← copy_word, copy_ptr
      else
      if (pointer_handle = tuple) ∧ (rwindex < index)∧
        ((state = sweep_read) ∨ (state = sweep_clear) ∨ (state = sweep_write))
      then
        val address = dest + rwindex in
        buffer_word, buffer_ptr ← mem[address]_word, mem[address]_ptr
      else
       val address = dir[pointer_handle]_addr + rwindex in
       buffer_word, buffer_ptr ← mem[address]_word, mem[address]_ptr


write: val rwindex = pointer_offset + offset + 1 in
       if (pointer_handle = tuple) ∧ (rwindex = index)∧
         ((state = sweep_clear) ∨ (state = sweep_write))
       then copy_word, copy_ptr ← buffer_word, buffer_ptr
       else
       if (pointer_handle = tuple) ∧ (rwindex < index)∧
         ((state = sweep_read) ∨ (state = sweep_clear) ∨ (state = sweep_write))
       then
         val address = dest + rwindex in
         mem[address]_word, mem[address]_ptr ← buffer_word, buffer_ptr
       else
        val address = dir[pointer_handle]_addr + rwindex in
        mem[address]_word, mem[address]_ptr ← buffer_word, buffer_ptr

       if buffer_ptr
       {  if (¬dir[buffer_handle]_mark)∧
            ((state = mark_scan) ∨ (state = mark_add) ∨ (state = mark_next))
          then
          {  dir[buffer_handle]_mark, ← true
          &  livesize ← livesize + dir[buffer_handle]_size + 1
          &  dir[buffer_handle]_list, next ← next, buffer_handle
          }
          else skip
       &  dir[pointer_handle]_deep ← true
       }
```

Appendix B

# *Software Used*

The following software was used to make the project and this thesis. The software written by the author or other code adapted for use in the project has been referenced at appropriate points in this thesis, and is provided in attached files.

- Xilinx® Vivado® v2016.1 WebPACK™ Edition
- Draw.io
- Miktex
- TexnicCenter
- Microsoft® Windows®
- Cadence® JasperGold®
- Git
- GitKraken®
- Notepad++
- Gitlab®
- SumatraPDF
- Techsmith® Snagit®
- Adobe® Photoshop® CS3

Appendix C

# *Sample Calculations*

The following is an example of calculating the directory size. The $address\_size$ is:

$$log_2(memory\_size) - 2$$

where $memory\_size$ is the size of main memory in bytes. For main memory size of 65,536 words, the $address\_size$ equals 16 bits. The handle size is:

$$log_2(handles)$$

where $handles$ is the number of handles. For 4,096 handles, 12 bits are required. The $length\_size$ is:

$$log_2(max\_object\_size) - 2$$

where $max\_object\_size$ is in bytes. For objects of maximum size 65,536 words, 16 bits are required. Putting that into equation 6.1, the directory size is 23,552 bytes which is 23KiB.

Appendix D

# *Bounds Analysis Diagrams*

The following figures were used to compute the lower and upper bounds of block and response times for the IHGC. The bounds were then checked in behavioural simulation in Vivado v2016.1 64-bit edition.

Response times do not include the transition time taken to enter the first state of the operation. Blocking times are computed separately (see evaluation in Chapter 4) taking into account transition times and these response times.

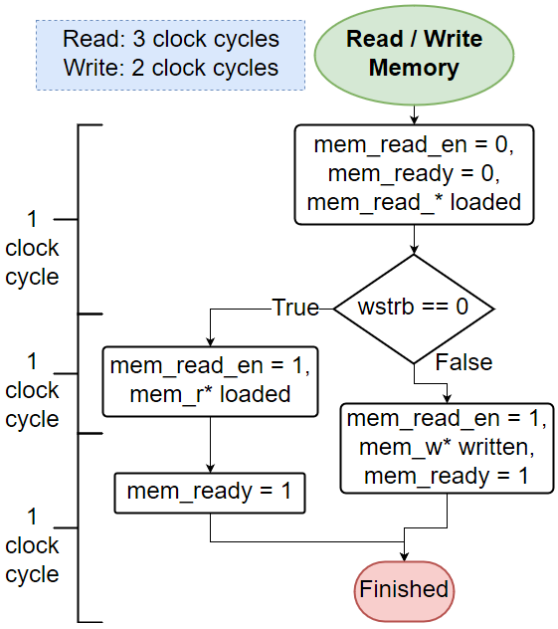



Figure D.1: Response Time for Read/Write Memory

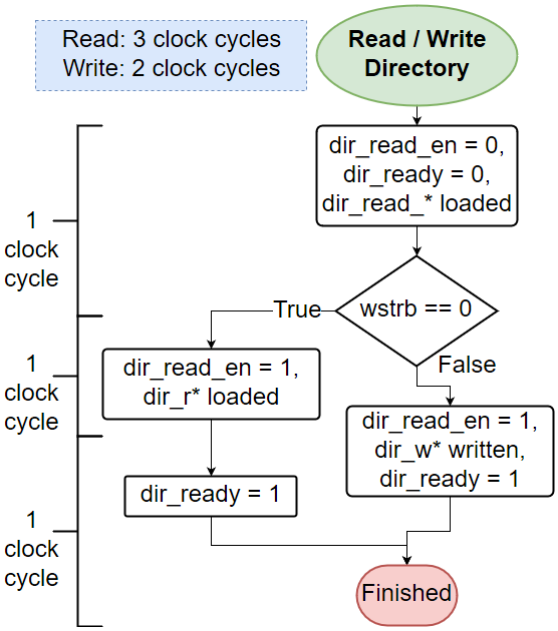



Figure D.2: Response Time for Read/Write Directory

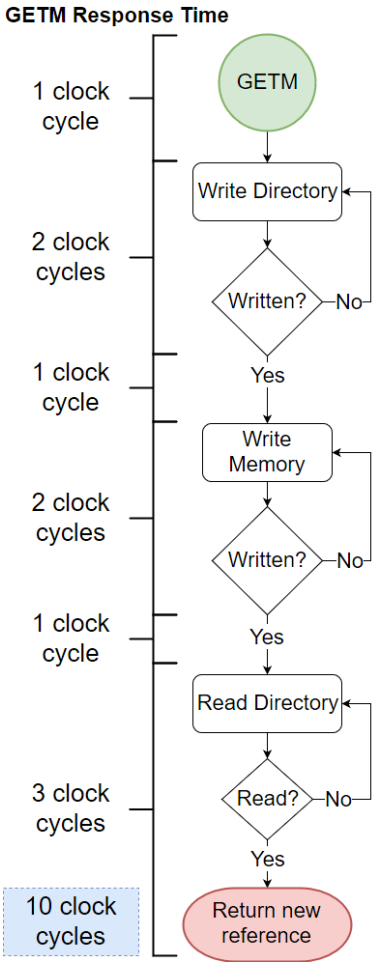



Figure D.3: Response Time for Get Memory

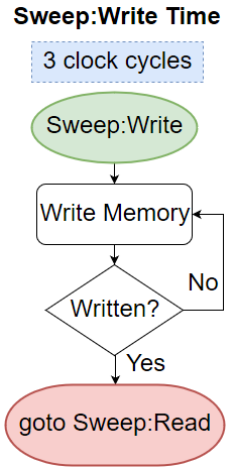



Figure D.4: Sweep:Write State Time

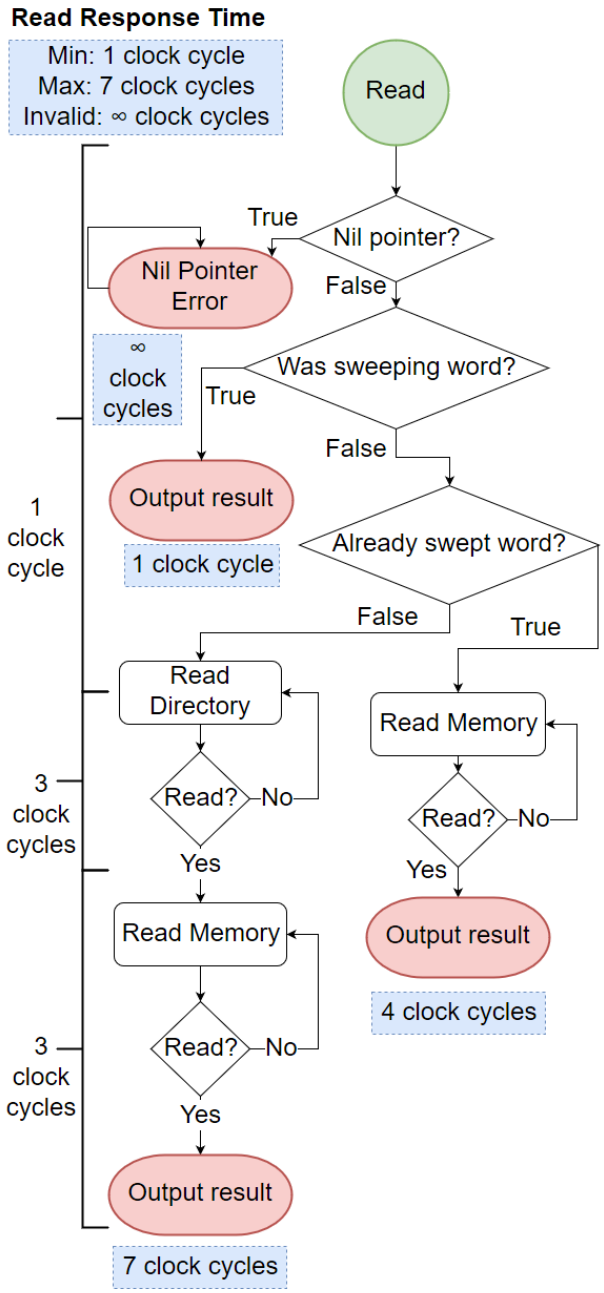



Figure D.5: Response Time for IHGC Request to Read

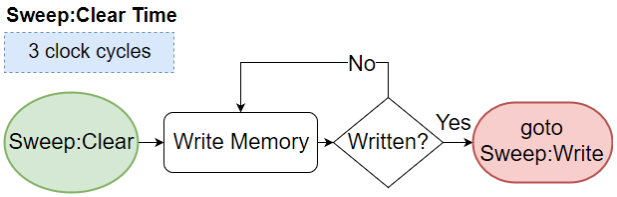



Figure D.6: Sweep:Clear State Time

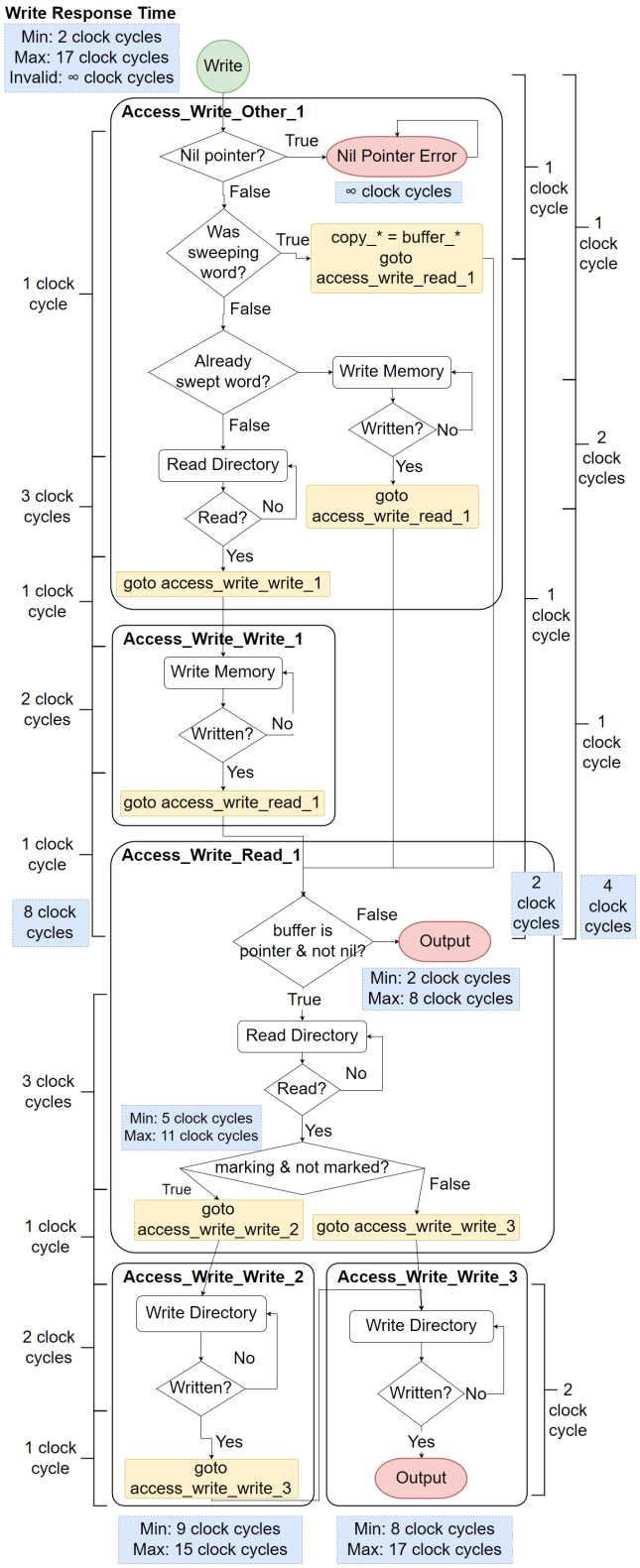



Figure D.7: Response Time for IHGC Request to Write

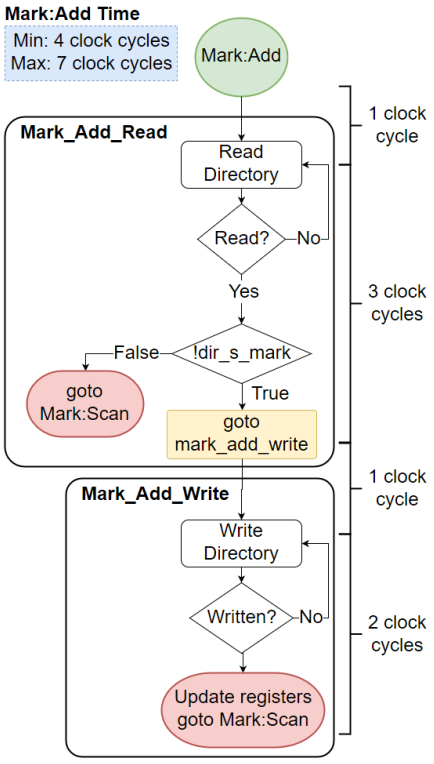



Figure D.8: Mark:Add State Time

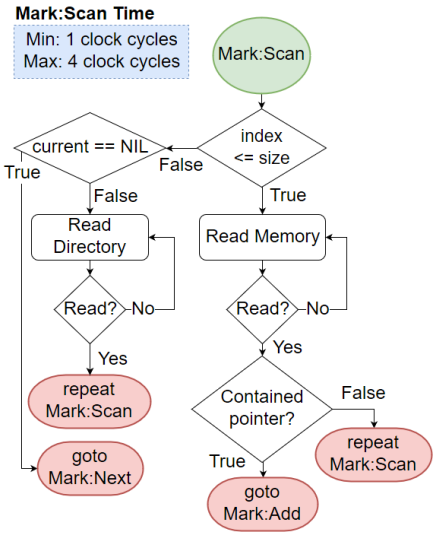



Figure D.9: Mark:Scan State Time

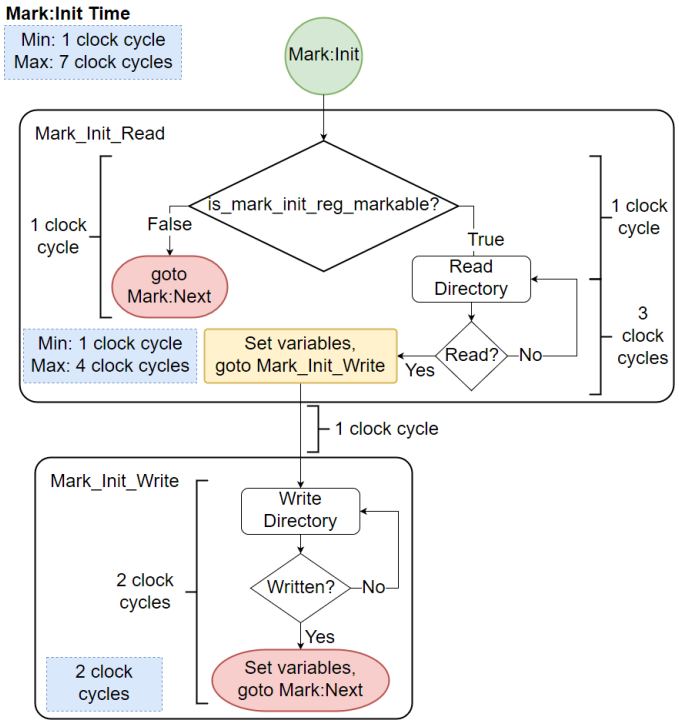



Figure D.10: Mark:Init State Time

*The condition requiring the CPU's registers to be available is ignored as it is assumed that under the conditions necessary for WCET of M&S, the core registers must be available at all times. It is impossible to estimate the blocking time of the CPU core in the general case.*

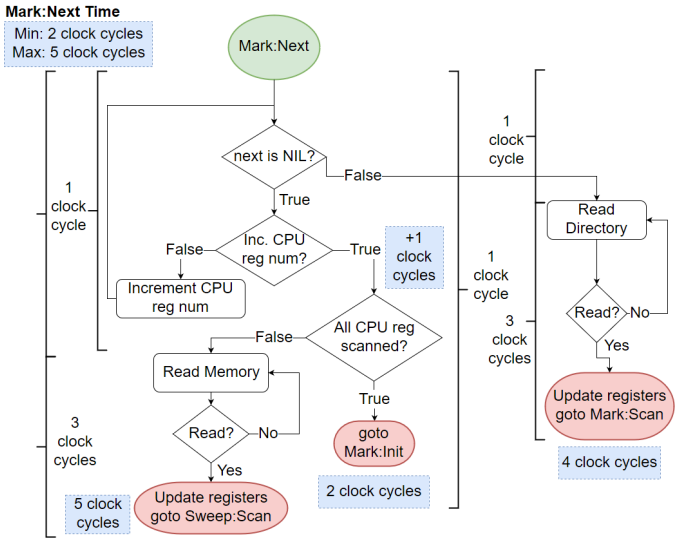



Figure D.11: Mark:Next State Time

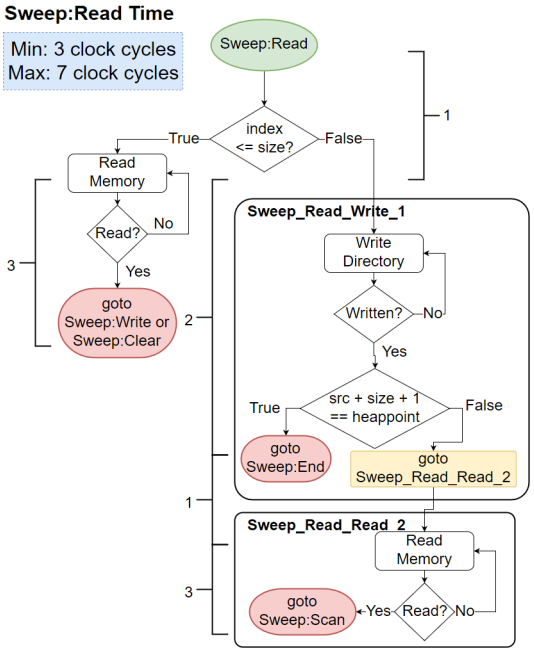



Figure D.12: Sweep:Read State Time

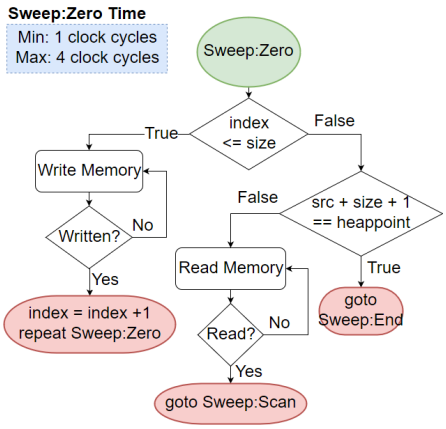



Figure D.13: Sweep:Zero State Time

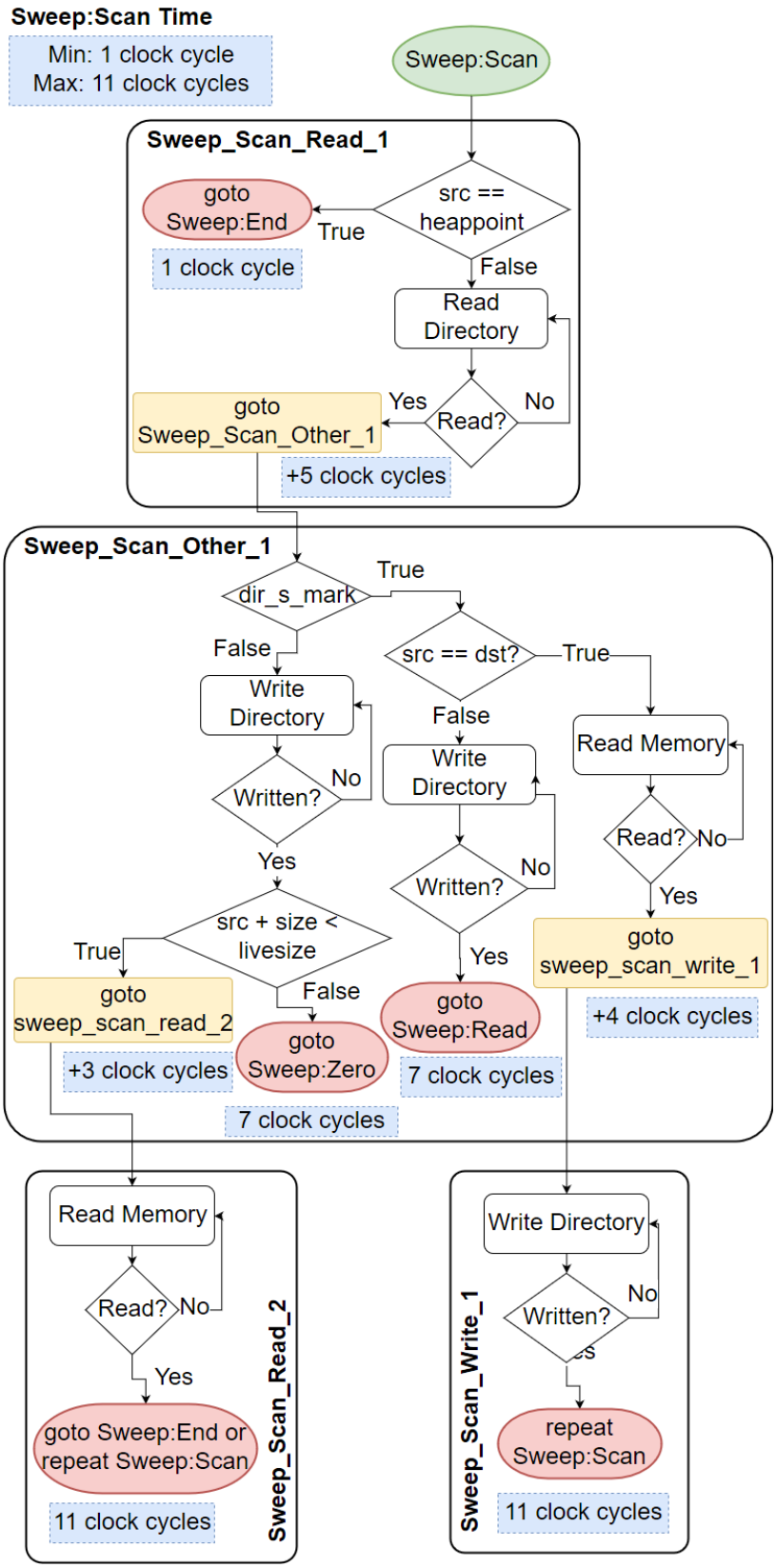



Figure D.14: Sweep:Scan State Time

# *Bibliography*


[1] B. Alpern et al., The jikes research virtual machine project: Building an open-source research community, in IBM Systems Journal, vol. 44, 2005, pp. 399–417.

[2] ARM Limited, Cortex-R4 Overview, April 2017. https://developer.arm.com/products/processors/cortex-r/cortex-r4.

[3] D. F. Bacon, P. Cheng, and S. Shukla, A stall-free real-time garbage collector for FPGAs, in The Second Workshop on the Intersections of Computer Architecture and Reconfigurable Logic, 2012.

[4] S. M. Blackburn et al., The dacapo benchmarks: Java benchmarking development and analysis, in OOPSLA, 2006.

[5] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, The yin and yang of power and performance for asymmetric hardware and managed software, in 39th Annual International Symposium on Computer Architecture, June 2012, pp. 225–236.

[6] J. Chang, W. Srisa-an, C.-T. D. Lo, and E. F. Gehringer, DMMX: Dynamic memory management extensions, Journal of Systems and Software, 63 (2002), pp. 187 – 199.

[7] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, On-the-fly garbage collection: An exercise in cooperation, Communications of the ACM, 21 (1978), pp. 965–975.

[8] E. F. Gehringer, Hardware-assisted memory management, in Object-Oriented Programming, Systems, Languages, and Applications, 1993.

[9] I. Gouy, The Computer Language Benchmarks Game, May 2017.

http://benchmarksgame.alioth.debian.org/.

[10] F. Gruian and Z. Salcic, Designing a concurrent hardware garbage collector for small embedded systems, in Advances in Computer Systems Architecture, T. Srikanthan, J. Xue, and C.-H. Chang, eds., vol. 3740 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 281–294.

[11] T. Heil and J. E. Smith, Concurrent garbage collection using hardware assisted profiling, in Chambers and Hosking, 2000, pp. 80–93.

[12] M. T. Higuera-Toledano, V. Issarny, M. Banatre, G. Cabillic, J.-P. Lesot, and F. Parain, Memory management for real-time Java: an efficient solution using hardware support, Real-Time Systems Journal, 26 (2004), pp. 63–87.

[13] U. Hölzle and D. Ungar, Do object-oriented languages need special hardware support?, in European Conference on Object-Oriented Programming, vol. 952, August 1995, pp. 283–302.

[14] A. Ive, Towards an embedded real-time Java virtual machine, in Lic.Thesis 20, Dept. of Computer Science, Lund University, 2003.

[15] J. A. Joao, O. Mutlu, and Y. N. Patt, Flexible reference-counting-based hardware acceleration for garbage collection, in 36th annual international symposium on computer architecture, June 2009, pp. 418–428.

[16] R. Jones, A. Hosking, and E. Moss, The Garbage Collection Handbook, John Wiley & Sons, 2016.

[17] D. E. Knuth, Fundamental Algorithms, vol. 1, Addison-Wesley, 3rd ed., 1997.

[18] D. Lea, dlmalloc version 2.8.6, August 2012.
ftp://g.oswego.edu/pub/misc/malloc.c.

[19] M. Maas, K. Asanović, and J. Kubiatowicz, Grail quest: A new proposal for hardware-assisted garbage collection, in Sixth Workshop on Architectures and Systems for Big Data, 2016.

[20] M. Maas, P. Reames, J. Morlan, K. Asanović, A. D. Joseph, and J. Kubiatowicz, GPUs as an opportunity for offloading garbage collection, in the 2012 International Symposium on Memory Management, June 2012, pp. 25–36.

[21] M. Meyer, An on-chip garbage collection coprocessor for embedded real-time systems, in 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, August 2005, pp. 517–524.

[22] Microsoft Corporation, .Net CoreCLR on GitHub, May 2017. https://raw.githubusercontent.com/dotnet/coreclr/master/src/gc/gc.cpp.

[23] K. D. Nilsen and W. J. Schmidt, Hardware-Assisted General-Purpose Garbage Collection for Hard Real-Time Systems, no. 102, Digital Repository Iowa State University, October 1992.

[24] W. Puffitsch, Hard real-time garbage collection for a Java chip multi-processor, in 9th International Workshop on Java Technologies for Real-Time and Embedded Systems, September 2011, pp. 64–73.

[25] W. J. Schmidt and K. D. Nilsen, Performance of a hardware-assisted real-time garbage collector, in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VI, New York, NY, USA, 1994, ACM, pp. 76–85.

[26] Y. W. Shing, R. Li, and A. S. Fong, Hardware concurrent garbage collection for short-lived objects in mobile java devices, in Embedded And Ubiquitous Computing, vol. 3824, 2005, pp. 47–56.

[27] W. Srisa-an, C.-T. D. Lo, and J. M. Chang, Active memory processor: A hardware garbage collector for real-time Java embeded devices, IEEE Transactions on Mobile Computing, 2 (2003), pp. 89–101.

[28] S. Stanchina and M. Meyer, Mark-sweep or copying?: A "best of both worlds" algorithm and a hardware-supported real-time implementation, in Proceedings of the 6th International Symposium on Memory Management, ISMM '07, New York, NY, USA, 2007, ACM, pp. 173–182.

[29] J. Tang, S. Liu, Z. Gu, X.-F. Li, and J.-L. Gaudiot, Hardware-assisted middleware: Acceleration of garbage collection operations, in 21st IEEE International Conference on Application-specific Systems, Architectures and Processors, July 2010.

[30] A. Waterman, Y. Lee, D. Patterson, and K. Asanović, The RISC-V Instruction Set Manual, May 2017.

[31] D. S. Wise, B. Heck, C. Hess, W. Hunt, and E. Ost, Research demonstration of a hardware reference-counting heap, (1997).

[32] C. Wolf, PicoRV32, GitHub Commit a2107e, 2017-01-27.
https://github.com/cliffordwolf/picorv32/commit/a2107ed4ffa72b48a15c67459adf8fbe576cbcab.

[33] G. Wright, M. L. Seidl, and M. Wolczko, An object-aware memory architecture, Science of Computer Programming, 62 (2006), pp. 145–163.