Integrated Hardware Garbage Collection for Real-Time Embedded Systems

By

ANDRÉS AMAYA GARCÍA

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of DOCTOR OF PHILOSOPHY in the Faculty of Engineering.

Department of Computer Science
UNIVERSITY OF BRISTOL

AUGUST, 2021

Word count: sixty-five thousand

Abstract

Modern programming languages, like Python and C#, provide productivity and trust benefits that are key in managing the growing complexity of computer systems. However, modern language implementations rely on software garbage collection which imposes high overheads and unpredictable pauses. This is tolerable in large computer systems, like desktops and servers, but impractical for real-time embedded systems. Hence modern languages are rarely used to program embedded devices.

This thesis investigates a shift in architecture towards hardware garbage collection to better support modern languages in embedded devices while meeting their unique performance and real-time requirements. We present an *Integrated Hardware Garbage Collector* (IHGC) that demonstrates this approach: a collector that is tightly coupled with the processor and runs continuously in the background. Our design allocates a memory cycle to the collector when the processor is not using the memory. The IHGC achieves this by careful subdivision of collection work into single-memory-access steps that are interleaved with the processor's memory accesses. We also introduce a static analysis technique to guarantee that real-time programs are never paused by the IHGC. As a result, our collector eliminates run-time overheads and is suitable for real-time embedded systems.

The IHGC is evaluated through simulation based on a hardware implementation model using modern fabrication technologies. Our experiments indicate that the IHGC offers 1.5-7 times better performance compared to a conventional processor running a software garbage collector. In addition, our static, real-time analysis technique was evaluated through practical use cases showing that an IHGC system meets specific timing constraints. This thesis concludes that the IHGC delivers in real-time the benefits of garbage collected languages without the complexity and overheads inherent in software collectors.

Dedication and acknowledgements

My research studies at the University of Bristol have been a wonderful journey culminating in the completion of this thesis. This experience would not have been possible without my supervisor, David May, who has provided guidance, support and encouragement from start to finish. Many of David's ideas motivated and inspired the work presented in this thesis. I also owe special thanks to Kerstin Eder, head of the Trustworthy Systems Laboratory, who welcomed me to the research group and provided valuable advice throughout my studies.

I am grateful to many other people who contributed to this work in various ways. Ed Nutting, a fellow student, who shared his knowledge on hardware garbage collection and collaborated with me in several research projects. Kyriakos Georgiou who introduced me to real-time computing and provided access to critical tools to get my own investigations underway. Jaeden Amero and Douglas Orr who provided valuable advice and feedback that guided my work. Jeremy Morse whose advice and help was crucial to developing compiler backends with LLVM. Henk Muller, Simon Clemow and James Hanlon who provided valuable expertise that enabled me to put my research in context.

Finally, I would like to thank the Engineering and Physical Sciences Research Council (EPSRC) for providing the financial means to support my studies.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's *Regulations and Code of Practice for Research Degree Programmes* and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ANDRÉS AMAYA GARCÍA..... DATE: 22/08/2021.....

TABLE OF CONTENTS

List of Tables

List of Figures

List of Listings

CHAPTER 1

INTRODUCTION

Modern programming languages, like Python and C#, are now the most popular and fastest growing among programmers [45, 166]. The defining feature of these languages is that they provide high-level data representation and control structures which address two key challenges: productivity and trust. Modern languages accelerate software development because they allow engineers to focus on coding the program's functions without the burden of managing tedious implementation details. So these languages increase productivity compared to older technologies like C. Modern languages also boost trust as they facilitate error prevention, detection and containment by, for example, eliminating memory usage errors which account for 50-70% of software vulnerabilities [174, 177]. As a result, these languages now dominate rapid-development environments, such as mobile phone apps and web services.

However, existing implementations of modern languages incur high performance and memory overheads. This is tolerable in large systems, like desktops and servers, which have abundant memory and processing resources, and where low software development costs and short time-to-market are valued above performance. But the drawbacks of modern language implementations are unacceptable for the embedded systems discussed in this thesis. Memory and processing are scarce commodities in these systems because they are small, constrained and have flat memory hierarchies sometimes even without virtual memory support or caching. Hence, modern languages are rarely used in embedded systems, despite their productivity and trust benefits, as they would exacerbate performance problems.

Further, embedded systems increasingly communicate with each other or interact with their physical environment. Both activities require these devices to perform tasks in response to external stimuli, often within a time deadline. So embedded systems are *real-time* because the time when the output of a task is delivered is as important as the output value itself. Performing

the task after a deadline expires is equivalent to a failure and may have lethal consequences. For example, an embedded device managing an airplane sensor must deliver measurements in a timely fashion to prevent accidents. However, existing implementations of modern languages introduce unpredictable pauses during program execution making it difficult to ensure that tasks are performed on time. Therefore, modern languages are not used to program real-time embedded systems.

Emerging markets, such as the Internet of Things (IoT), have fueled a desire to incorporate more functionality into embedded systems. For example, a household thermostat that could only be used to set the room temperature nowadays also has internet connectivity, network security, voice controls, power-saving features, etc. The extra functionality resulted in growing software complexity giving rise to productivity and trust issues. Hence, there is renewed interest in supporting modern languages in embedded systems. Multiple companies and open-source projects, like MicroPython and Zerynth, are attempting to develop implementations of modern languages suitable for embedded devices [84, 115, 196]. However, their applications are limited because these implementations rely on software *garbage collectors*.

The garbage collector is an essential component in the implementation of modern languages. It automatically identifies and reclaims unused memory on behalf of the programmer, a task that is otherwise tedious and error-prone. But software garbage collectors are the cause of the performance and real-time drawbacks of modern languages. The main motivation behind this thesis is to solve these problems with garbage collection to enable the practical use of modern languages in embedded systems.

1.1 Memory Management

Memory is a fundamental component of every computing system. It is a finite resource that must be managed to ensure that the program's space requirements are met. However, memory management has remained an open problem for decades despite extensive research in the area. It is a recurrent and important cause of safety and reliability issues in computer systems. There are two main approaches to memory management in embedded systems: *static* and *dynamic*.

When using static memory management, the space allocated for variables, arrays and even the program stack is reserved at compilation time and allocated at startup. The advantage is that the program's execution time is highly predictable, so static memory is suitable for real-time systems [79]. However, the memory layout is difficult to change at run-time, so enough space must be reserved in advance to satisfy the program's memory requirements. For example, a buffer must have enough capacity to accommodate the largest possible packet that might be delivered via a network interface. This restriction causes memory overheads because the system must reserve sufficient storage in advance to accommodate the theoretical worst-case memory requirement of every object that the program allocates.

The limitations of static memory management are difficult to circumvent in systems whose behavior can only be determined at run-time. For example, a processor that is shared by multiple applications must allocate and reclaim memory as processes start and terminate. Similarly, programs written using modern languages often allocate variable amounts of memory depending on the type of the input values supplied at run-time. These problems are addressed by dynamic memory management algorithms that allocate space as requested by the program at run-time. The memory is reclaimed when it is no longer needed and can be repurposed to fulfil future allocation requests. Dynamic memory managers can be broadly classified as *explicit* or *automatic* according to the interface presented to the programmer.

1.1.1 Explicit Memory Managers

In explicit memory management, the programmer must indicate, usually via a function call, when allocated memory becomes unused and can be reclaimed. The `malloc` and `free` functions in the C standard library are perhaps the most famous realization of an explicit memory management interface.

Explicit memory management effectively delegates the tedious and error-prone memory management burden to the programmer which commonly results in *dangling pointers* and *memory leaks*. Dangling pointers are references to already freed objects. Hence, the pointers are invalid, so using them to access memory causes serious memory safety violations. Memory leaks occur when unused memory is not freed. The leaked memory will never be reclaimed and the system eventually appears to run out of storage space. Dangling pointers and memory leaks remain a major problem and are difficult to find, so there is much existing research attempting to detect them automatically. For example, static program analysis tools, like Coverity, issue a warning when such bugs are found [51, 78, 175], but they generate many false positive warnings and do not guarantee the absence of memory management bugs because the analysis problem is undecidable.

Explicit memory managers have been studied extensively, but most proposals are unsuitable for embedded systems. This is because the run-times of explicit memory allocation and free operations are difficult to analyze statically, so these memory managers cannot be used in real-time systems [136]. In addition, explicit memory managers rarely compact the heap, so they suffer from *fragmentation*: after repeated allocations and deallocations, the free memory becomes segmented into small blocks that cannot be used to satisfy allocation requests. As a result of these problems, programming safety standards, like MISRA C [116], discourage the use of explicit memory managers.

The drawbacks of explicit memory managers have forced programmers to develop domain-specific memory managers for embedded systems. Thus, new domain-specific, explicit memory managers are coded from scratch for almost every embedded project in an effort to fulfil the system's real-time, performance and reliability constraints. For example, the open-source lwIP

networking stack includes a custom memory manager to efficiently handle packet data [59]. Similarly, the open-source FreeRTOS process scheduler provides a choice of five custom memory managers [12]. But domain-specific memory managers increase program complexity because they require a substantial amount of code to implement and operate correctly. The design of these managers is very closely tied to a particular application, so there are few opportunities for code reuse. Domain-specific memory managers are also difficult to use, often inefficient and rarely tested thoroughly giving rise to performance and reliability issues as Zorn found in a previous study [198].

In general, modern programming languages do not rely on explicit memory management interfaces due to the reliability and usage problems outlined above. Therefore, we cannot rely on explicit memory managers to support modern languages in real-time embedded systems.

1.1.2 Automatic Memory Managers

In automatic memory management, unused, or *garbage*, objects in memory are identified and reclaimed by the system without programmer intervention. Automatic memory management is implemented by a component of the runtime called the *garbage collector*.

Automatic memory managers relieve the programmer from the memory management responsibility, so they overcome most drawbacks of explicit memory managers. Namely, programming errors, like dangling pointers and memory leaks to unreachable objects, are eliminated and garbage collectors often compact memory to avoid fragmentation. However, garbage collection is often performed in software, so it incurs high run-time and memory overheads because the collection algorithms are unsuitable to be efficiently implemented in conventional computers. Previous studies found that software collectors account for up to 40% of a program's run-time [43, 67]. In addition, software garbage collectors introduce unpredictable pauses during program execution that prevent systems from meeting their deadlines. As a result, existing automatic memory managers are unsuitable for real-time embedded systems.

Automatic memory management, and garbage collection, are defining features of modern programming languages. They underpin the implementation of these languages and facilitate error prevention, detection and containment. Therefore, overcoming the performance and real-time drawbacks of garbage collection is critical to supporting modern programming languages in embedded systems and delivering their productivity and trust benefits.

1.2 Real-Time Systems

Any discussion on embedded systems requires considering real-time behavior. In these systems, calculating the correct result is as important as performing the calculation on time. Delivering the result late is considered a failure. Real-time systems can be broadly classified in two groups.

Soft Real-Time: It is desirable for the system to meet all its deadlines, but occasionally missing one is acceptable. For example, an interactive system with a display is soft real-time. A missed deadline that causes a delay to refresh the screen only degrades user experience, but does not lead to serious consequences.

Hard Real-Time: The system must meet all its timing deadlines. Missing a deadline is equivalent to an incorrect calculation and might result in catastrophic consequences. For example, an autonomous vehicle not reacting in time to a change in its environment might crash.

Real-time systems must be evaluated by their ability to meet deadlines instead of only considering overall run-time performance. It is not guaranteed that systems with the shortest overall run-time, i.e. the least performance overheads, will always meet their deadlines. For example, the software collectors with the shortest overall run-times often perform memory management operations that take a long time to complete. The system could miss a deadline during that time because the program cannot perform any other work while the collector is running. Therefore, all operations that the real-time system performs, such as memory allocations, must have predictable and short run-time to ensure that deadlines are met.

Distinguishing between average-case and worst-case behavior is also important when discussing real-time systems [79]. For most applications, including soft real-time ones, it is the average-case that matters since it occurs frequently. Furthermore, it is often acceptable to optimize the average-case at the expense of longer worst-case overheads. The development and testing of these systems is mostly based on empirical experimentation and observations to ensure that performance and response times are within acceptable parameters, but the worst-case can still occur. Most garbage collectors and modern language implementations, are intended for these soft real-time applications.

Hard real-time systems guarantee that all deadlines are met, so it is the worst-case behavior that matters. These systems must be analytically shown to not violate timing constraints. In other words, a formal analysis is used to demonstrate that the run-time of all operation, such as load, stores and memory allocations, is bound by a small constant in the worst-case, so it is impossible for the system to miss a deadline. Hard real-time systems cannot rely on empirical experimentation, observations or average-case costs [123]. Also, these systems are often safety critical and cannot fail due to issues like fragmentation. It is hard real-time systems that this thesis discusses.

Software garbage collectors for hard real-time systems have been proposed in the literature. But these collectors incur high run-time penalties, often utilizing over 50% of the processor's time [28], and large memory overheads [28, 50, 87, 92]. The real-time analysis formulations of these collectors are often incomplete, like Schoeberl's that is missing a timing model of their collector (see Section 3.4.5) [148], or flawed, like the Metronome's timing model that is based on empirical observations instead of analytical worst-case bounds (see Section 3.4.2) [27]. In addition,

real-time analysis is inherently complex, so programmers require assistance from automated tools, such as AbsInt's aiT [2]. However, the literature rarely discusses these automated tools in the context of automatic memory management and it is unclear how the experimental evaluation of existing software real-time garbage collectors was performed. As a result, modern programming language implementations using existing software garbage collectors for embedded systems are unusable in practical hard real-time settings.

Nowadays, hard real-time systems use neither automatic nor explicit memory managers due to the problems outlined previously. Instead, these systems are designed alongside domain-specific memory managers to ensure that they meet strict performance, memory and real-time constraints. But domain-specific memory managers have serious drawbacks as discussed in Section 1.1.1. So there is a need to resolve the issues of garbage collection in order to enable the productivity and trust benefits of modern languages in hard real-time embedded systems.

1.3 Thesis Questions and Contributions

The aim of this thesis is to enable the use of modern, garbage collected programming languages in embedded systems. This will bring the well-known productivity and trust benefits of these languages to the millions of embedded devices that are deployed annually. To achieve this, we adapt existing techniques from hardware design, garbage collection and real-time analysis to fulfil the unique requirements of embedded devices. Our main focus is to answer two research questions:

- **How can hardware garbage collectors be designed to deliver better run-time performance and minimize memory overheads compared to existing collectors?** Existing implementations of modern languages for embedded devices rely on well-understood software garbage collectors. However, these collectors incur high overheads because the algorithms are unsuitable to be implemented efficiently in conventional computers. We investigate a shift in architecture, towards hardware garbage collection, to address these problems. Little research on hardware garbage collection has considered the specific needs of embedded systems.

- **How can a hardware garbage collector's timing properties be analyzed to guarantee that hard real-time requirements are met?** A timing model of the collector is required for the system to be provably hard real-time. This model is combined with information extracted from the program to derive safe bounds on the use of resources such as memory and processing time. Previous research has neglected to apply this rigorous analysis to hardware garbage collectors. Instead, existing hard real-time collectors are implemented in software and have impractical overheads for embedded devices.

Our approach is to design an *Integrated Hardware Garbage Collector* (IHGC) that demonstrates the shift in architecture from software to hardware garbage collection. Throughout this thesis, we thoroughly investigate the impact of hardware garbage collection on the design of embedded systems, evaluate the performance of our proposal and analyze its real-time capabilities. Our main contributions are:

Design of the IHGC

1. A garbage collector fully implemented in hardware that runs continuously in the background reclaiming memory independently from the processor.
2. A garbage collector implemented as a small state machine having the property that each state transition is performed in a single memory cycle. The collector does not normally pause the user's program as it performs state transitions when the processor is not using the memory for instruction execution.
3. A garbage collector optimized by implementing an indirection through handles using a fast directory memory that contains metadata for allocated objects.
4. The use of a directory and hardware in the processor to efficiently mark pointers loaded from memory and redirect memory accesses while compacting. This guarantees program correctness in a system running a hardware collector concurrently with the processor.
5. The design of a hardware garbage collector implementing a mark-compact algorithm. The collector's run-time can be estimated using static analysis methods, so it is suitable for hard real-time applications.
6. An exact garbage collector implemented in hardware for weakly typed languages such as C.
7. Detailed investigation and discussions about how design decisions in the architecture of a hardware collector affect its timing properties.

Real-time analysis with the IHGC

8. A hard real-time analysis technique that estimates the amount of memory needed to guarantee that the IHGC never pauses the user's program, or determines that the program cannot be run without pauses. The input to the analysis are parameters extracted from the program and the worst-case run-time of a collection cycle for the same program.
9. A timing model that estimates the worst-case run-time of a collection cycle in the IHGC given a set of parameters extracted from a real-time program, like the amount of memory allocated.

10. An automated software tool that uses static program analysis techniques adapted from Worst-Case Execution Time (WCET) research to estimate the amount of allocated memory and of memory cycles available to the IHGC from the compiled binary of a given program. The tool enables using the IHGC in a practical real-time system.

The IHGC in an embedded system

11. An analysis of the impact of a hardware garbage collector, such as the IHGC, on RISC Instruction Set Architectures (ISA) and proposals for changes to maximize the benefits of the hardware collector at the architecture level.

12. A compiler specifically developed for an architecture with the IHGC and an assessment on the changes and amount of effort required to port existing embedded software, including existing modern languages implementations, to a system with the IHGC using that compiler.

13. The microarchitecture of the IHGC alongside a pipelined embedded processor supporting a RISC ISA. The design takes into account the features and constraints of a modern fabrication technology suitable for embedded systems.

Evaluation of the IHGC

14. Estimation of the hardware costs and clock speed of the proposed IHGC microarchitecture demonstrating that the design is within the parameters of modern embedded systems.

15. Evaluation, through simulation, demonstrating that the IHGC offers similar or better performance when running C programs in comparison to a system without the IHGC. Also, the IHGC offers 1.5-7 times better performance when running programs written in Python, a modern programming language, in comparison to a software garbage collector.

16. Evaluation demonstrating that the IHGC introduces few program pauses when the heap size is minimally increased, e.g. by a factor of 1.5, beyond the minimum operational requirement. The experiments also shows how design parameters, like the memory's data bus width, can reduce pauses by up to 20% without significant hardware overheads.

17. A demonstration of the IHGC and its real-time analysis in practice. The study also shows how parameters, like the clock speed, affect the estimated memory requirements to eliminate collection pauses.

1.4 Thesis Outline

The remaining chapters in this thesis are organized in four parts. Background information is presented in Part I along with a review of the literature on real-time and hardware garbage collection. A novel hardware garbage collector, the IHGC, is proposed in Part II and its hard real-time properties are analyzed. A simulation-based evaluation of the collector's performance and the real-time analysis are also presented. Part III describes and evaluates the architecture and microarchitecture of an embedded system integrated with the proposed garbage collector. Finally, Part IV identifies future work and concludes the thesis.

Part I: Background

- *Chapter 2: Fundamentals of Garbage Collection*
  Background information on garbage collection theory is presented including basic algorithms and techniques. Garbage collection design principles are also discussed.
- *Chapter 3: Real-Time Garbage Collection*
  Software garbage collectors with an emphasis on hard real-time systems are described and compared. Their performance and memory requirements are analyzed and their suitability is assessed in the context of constrained embedded systems.
- *Chapter 4: Hardware Garbage Collection*
  The literature on hardware garbage collection is discussed and knowledge gaps are highlighted. Multiple proposals are analyzed and compared in the context of real-time embedded systems.

Part II: Integrated Hardware Garbage Collection

- *Chapter 5: Designing an Integrated Hardware Garbage Collector*
  The design of the Integrated Hardware Garbage Collector (IHGC) is presented at a high-level. The most important tradeoffs and decisions are explained based on information from the literature and experimental results.
- *Chapter 6: Hard Real-Time Analysis with the IHGC*
  A static analysis technique is presented to guarantee that the IHGC never pauses a hard real-time program. The analysis relies on a rigorous timing model of the IHGC along with a software tool to automatically estimate the program's execution time and allocation rate.
- *Chapter 7: Experimental Evaluation*
  Presents the methodology and results of empirically evaluating the IHGC alongside an embedded processor. The experiments are performed in simulation using off-the-shelf compilers and open-source benchmarks. Hard real-time programs are also analyzed using the technique proposed in Chapter 6.

Part III: Architecting a Garbage Collected Embedded System

- *Chapter 8: Garbage Collection in Instruction Set Design*
  Discusses the impact of the IHGC on ISA design. It explores how fundamental building blocks, like procedure calling, of an architecture are affected by the garbage collector. Changes to existing and novel ISAs are also proposed to leverage the hardware collector's benefits at the architecture level.
- *Chapter 9: Microarchitecture of the IHGC*
  Describes the microarchitecture of the IHGC alongside a pipelined embedded processor. The design is a realistic implementation of the IHGC based on the capabilities of a modern fabrication technology. The timing and area requirements of the hypothetical system are also estimated and discussed.
- *Chapter 10: Evaluation of the IHGC Microarchitecture*
  Presents the methodology and results for a performance evaluation of the microarchitecture proposed in Chapter 9. The experiments are conducted using a simulator that models the previously presented hypothetical system.

Part IV: Conclusions

- *Chapter 11: Conclusions and Future Work*
  The thesis contributions are outlined and conclusions are discussed. Future research directions on hardware garbage collection and the IHGC are proposed.

1.5 Related Publications

Parts of the research presented in this thesis previously appeared in the following publications:

- A. AMAYA GARCÍA, D. MAY AND E. NUTTING, *Garbage collection for edge computing*, in 2020 IEEE/ACM Symposium on Edge Computing (SEC), 2020, pp. 319–319.
- A. AMAYA GARCÍA, D. MAY AND E. NUTTING, *Integrated hardware garbage collection*, ACM Transactions on Embedded Computer Systems (TECS), 2021.

Part I

Background

CHAPTER 2

FUNDAMENTALS OF GARBAGE COLLECTION

Automatic memory management is an integral part of most modern programming languages. Python, C#, JavaScript and many other modern languages rely on automatic memory management to increase productivity and trust. This is because the programmer is no longer burdened with tedious and error-prone memory management tasks. Therefore, it is essential that garbage collectors, the components responsible for automatic memory management, are reliable and efficient to deliver the benefits of modern languages.

The task of a garbage collector is conceptually simple: to identify and reclaim unused memory. But in practice, designing and implementing efficient collectors is difficult and countless schemes have been proposed in the literature. They attempt to balance three conflicting goals:

Low run-time overheads: Garbage collection is only an administrative task, similar to process scheduling. So the system should spend the least time possible in these operations and maximize the time dedicated to actual application work.

Low memory overheads: All memory management algorithms (static or dynamic) incur memory overheads, for example, to store the collector's internal data structures. These requirements must be minimized.

Short pauses: Garbage collectors often pause program execution, for example, if there is not enough free memory to satisfy a pending allocation request. These pauses are hard to predict, degrade performance and cause the program to appear unresponsive, so they must be short in duration.

Garbage collectors often optimize one or two of these goals at the expense of the others. For example, the collectors with the least run-time and memory overheads are executed uninterruptedly when the free memory is exhausted, but in this case the user's program is paused for a

long time. Alternatively, garbage collection can be performed concurrently with the execution of the user's program to reduce the duration of pauses. However, these schemes often have high performance drawbacks due to context switching and synchronization overheads. In this chapter, we introduce basic garbage collection theory and explain how the collector's design attempts to balance the three conflicting goals.

2.1 Basic Garbage Collection Algorithms

Identifying the exact set of unused objects is a difficult task. Instead, garbage collectors are conservative in their definition of unused: an object is unused if the program cannot *reach* it. The concept of reachability associates the program (or *mutator* in garbage collection terminology) with a *root* set of *pointers* (or *references*). An object can be reached if the program can access it by following a chain of pointers starting from the root set. Reachable objects (also known as *live*) must be retained by the collector while unreachable (or *dead*) objects are considered *garbage* and are reclaimed.

There are two basic approaches to garbage collection: *tracing* and *reference counting*. These are explained briefly and four basic algorithms are compared.

2.1.1 Tracing

Tracing is a technique to indirectly identify the garbage objects in memory. Tracing algorithms recursively mark the live objects starting from a root set of pointers, such as the processor's registers and the function call stack. The unreachable objects remain unmarked at the end of tracing and the collector can reclaim them.

Tracing simply identifies live objects. It must be complemented with a further technique capable of reclaiming the unused memory to construct a garbage collector. In their basic form, these collectors are run when the free memory is exhausted. The user's program is paused while the collector performs a full *cycle*, i.e. traces the memory once and reclaims all garbage objects. We discuss three basic tracing collectors: *mark-sweep*, *mark-compact* and *copying*.

2.1.1.1 Mark-Sweep and Mark-Compact

Mark-sweep collectors were initially proposed by McCarthy for the LISP programming language [109]. These algorithms operate in two stages: *mark* and *sweep*. The marking stage traces the live objects in memory as shown in Listing 2.1. The collector examines every root pointer in the root set. Each object found during tracing is marked and then scanned in a search for pointers to objects not yet discovered. This operation has a memory overhead because the references to marked but not scanned objects must be recorded. Listing 2.1 uses the function call stack for this purpose as it is recursive, but implementations using other data structures, such as

```
1  procedure trace():
2    foreach root in rootSet:
3      markAndScanObject(root)
4
5  procedure markAndScanObject(objref):
6    if isMarked(objref):
7      skip
8    else:
9      foreach pointer in object referenced by objref:
10       setMarked(pointer)
11       markAndScanObject(pointer)
```

Listing 2.1: Recursive implementation of the basic tracing algorithm.

linked lists, are also possible. Marking terminates when all the roots have been scanned and all reachable objects are marked and scanned.

During the sweeping stage, the collector scans every object in memory: unmarked objects are reclaimed while marked objects are retained. The mark flags are also cleared in preparation for the next mark-sweep cycle. However, the details of the sweeping stage change substantially depending on the collector's approach to tackle fragmentation. In the simplest case, the collector only labels memory from dead objects as 'free'. These collectors do not address the fragmentation problem, so memory allocation is often complex to implement efficiently. However, the collector itself is simple and has low overheads.

Systems that rely on basic mark-sweep collectors often mitigate fragmentation by only allocating objects within fixed, equally-sized memory blocks. But this causes *internal* fragmentation when the allocated object is smaller than the block size; the excess memory in the block will not be used and is wasted. For example, the memory space could be split into 32 byte blocks and every object is allocated using one or more such blocks. So an 8 byte object would be allocated using a 32 byte memory block; the remaining $32 - 8 = 24$ bytes in the block remain unused. In addition, allocations larger than the block size must be split across potentially discontiguous memory chunks. For instance, an array allocation of 1 KB needs to be split across multiple 32 byte blocks. Accesses to the nth element in the array are no longer constant-time because the data structure linking the object's memory blocks must be traversed until the requested location is found. These problems cause overheads and increase software complexity.

Mark-compact is a collection algorithm that prevents fragmentation. Instead of sweeping, these collectors copy the live objects towards one end of the memory so that the free space is clustered at the opposite end; the marking stage remains as described before. The LISP 2 garbage collector is an example of mark-compact [88]. It performs three passes over the memory during the compact stage as shown in Listing 2.2. The collector calculates the new locations of the live objects during the first pass. Then, the second pass updates all pointers in the live objects to reference the new addresses of the objects. Finally, each live object is copied towards one end of the address space to compact the memory in the third pass.

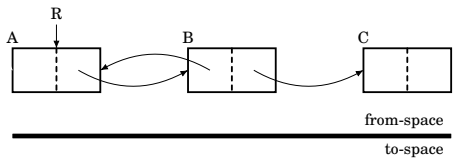
```
1  procedure compact():
2    calculateNewLocations()
3    updateReferences()
4    copyObjects()
5
6  procedure calculateNewLocations():
7    scan, free ← MemStart, MemStart
8    while scan < MemEnd:
9      if isMarked(scan):
10       setNewLocation(scan, free)
11       free ← free + getObjectSize(scan)
12     scan ← scan + getObjectSize(scan)
13
14 procedure updateReferences():
15   foreach root in rootSet:
16     root ← getNewLocation(root)
17
18   scan ← MemStart
19   while scan < MemEnd:
20     if isMarked(scan):
21       foreach pointer in object referenced by scan:
22         pointer ← getNewLocation(scan)
23     scan ← scan + getObjectSize(scan)
24
25 procedure copyObjects():
26   scan ← MemStart
27   while scan < MemEnd:
28     if isMarked(scan):
29       copy(getNewLocation(scan), scan)
30       unsetMarked(getNewLocation(scan))
31     scan ← scan + getObjectSize(scan)
```

Listing 2.2: Compact stage of the LISP 2 garbage collector taken from [88].

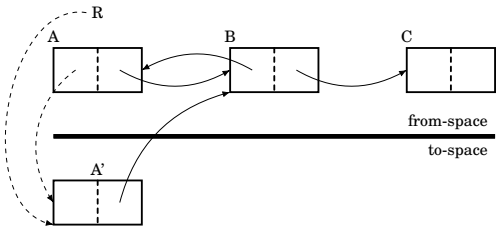
Mark-compact collectors have higher performance penalties than mark-sweep. This is because mark-compact entails costly memory copying and the collector must update pointers to copied objects to reflect their new memory location, otherwise the program will not operate correctly. An approach to mitigate this problem is using an indirection through *handles* [89]. The idea is that pointers reference handles which are memory locations containing the object's base address; pointers no longer refer to absolute memory addresses within an object. When an object is copied while compacting, the collector only has to update the address in the handle as opposed to every pointer referencing it. Therefore, the performance impact of the compact stage is mitigated although memory accesses are potentially slower because every load and store must read the address at the handle before accessing the object's contents.

2.1.1.2 Copying

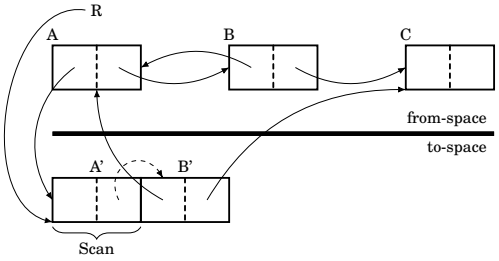
Copying algorithms address the performance drawbacks of mark-sweep and mark-compact collectors at the expense of higher memory overheads. Copying algorithms only require a single



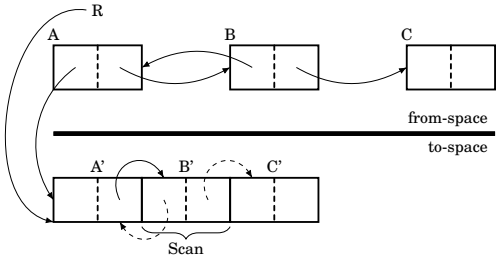
(a) The semispaces switch roles at the start of the collection cycle.



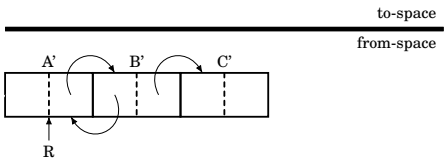
(b) Every object directly reachable from the roots is copied into *to-space*.



(c) The first object in *to-space* (A') is scanned in search for pointers to objects not yet discovered. A *to-space* copy of every object discovered is made and the pointer to that object is updated.



(d) Every remaining object in *to-space* is scanned in search for pointers. Newly discovered objects are copied and pointers to already copied objects are updated to reference the *to-space* copy.

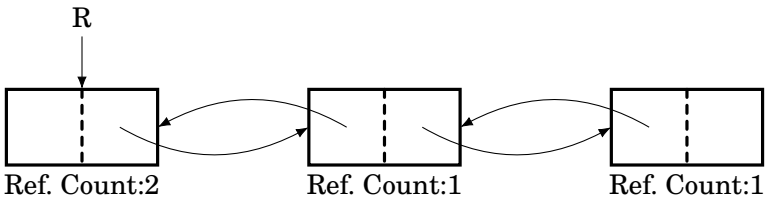


(e) The collection cycle terminates when every reachable object has been copied and all pointers reference a *to-space* location.

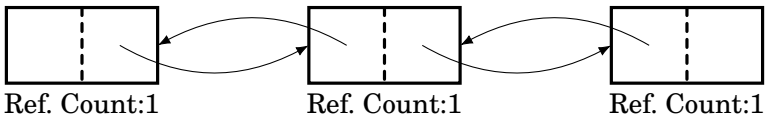
Figure 2.1: An example of Cheney's copying collector in operation.

pass through the memory instead of multiple ones. They split the memory space into two *semispaces* of equal size labeled *from-space* and *to-space*. The program allocates objects and operates in *to-space*. A collection cycle starts when there is not enough free memory in *to-space* to satisfy an allocation: a *flip* is performed by switching the roles of the semispaces. Then the memory is traced and live objects are *evacuated* (copied) from *from-space* to *to-space*. Finally, the data in *from-space* is simply discarded.

Cheney's algorithm is an example of a copying garbage collector [49]. Its operation is illustrated in Figure 2.1 on a system that has three live objects. The collector first performs a flip and starts tracing. The objects directly reachable from the root R are copied into *to-space* as shown in Figure 2.1(b). Also, a forwarding reference to the *to-space* copy of the object is stored in the *from-space* copy. The root pointers are updated to reference the *to-space* copy. The collector then starts scanning each *to-space* object copy as shown in Figure 2.1(c). If a pointer to an object



(a) Reference counted doubly-linked list. The head element is referenced from a root R.



(b) Reference counted doubly-linked list that is unreachable from the roots.

Figure 2.2: Doubly-linked lists cannot be reclaimed using reference counting because the pointers in the data structure form a cycle.

that has not been copied is found, then the newly discovered object is copied into *to-space*. For example, a copy B' of object B was created in Figure 2.1(c) because B is reachable from the object being scanned A'. Additionally, the collector updates the pointers found while scanning *to-space* object copies. The process is repeated until all reachable objects are copied and their pointers updated as illustrated in Figure 2.1(d). Finally, the *from-space* copies are simply discarded and the roles of the semi-spaces are switched as shown in Figure 2.1(e).

Copying collectors effectively double the system's memory requirements because only one semispace is used at a time. Also, copying collectors do not maintain the relative position of objects in memory which can be detrimental in some systems. For example, a reachable object A at a lower address than another object B before the collection cycle could be found at a higher memory address relative to B at the end of the cycle. But compared to mark-compact, the performance overheads of copying collectors are smaller. This is because the run-time of copying collectors is proportional to the size of the live data while the run-time of mark-compact collectors is proportional to the memory size.

2.1.2 Reference Counting

Reference counting is a direct garbage collection method proposed by Collins [54]. It identifies dead objects in memory as opposed to the live objects found by tracing algorithms. Reference counting associates a count with every allocated object. The count records the number of pointers that the program holds to that object. Garbage collection is simple as objects are reclaimed when their count reaches 0.

The advantage of reference counting is that objects can be reclaimed as soon as they become unreachable. In contrast, a full collection cycle must be completed for tracing algorithms to

	Mark-Sweep	**Mark-Compact**	**Copying**	**Reference Counting**
Defragment	No	Yes	Yes	No
Collects cycles	Yes	Yes	Yes	No
Memory overhead	Medium	Medium	High	Low
Starting point	Roots	Roots	Roots	Garbage objects
Objects traversed	Live	Live	Live	Garbage
Collection style	Batch	Batch	Batch	Incremental
Reclamation delay	Yes	Yes	Yes	Data structure with cycles

Table 2.1: Comparison of four basic garbage collection algorithms.

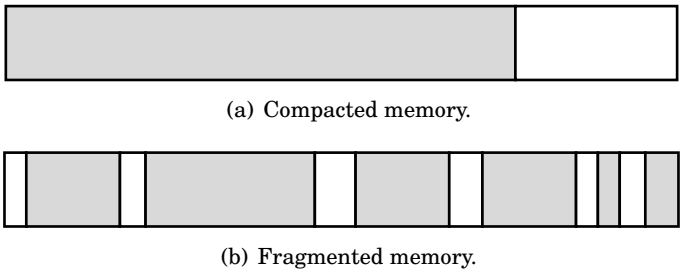
reclaim garbage objects. However, reference counting incurs memory overheads as each object must have space for the counts. There are also run-time overheads as potentially every memory access operation requires one or more counter updates. Admittedly, there are techniques, such as *deferred reference counting*, to mitigate these problems [38, 56].

Another important disadvantage of reference counting is its inability to reclaim objects forming data structures with *cycles*. For example, the doubly-linked list in Figure 2.2(a) cannot be reclaimed because its head element is referenced by a root. Eliminating the root pointer only decreases the head element's count by 1. The resulting garbage data structure in Figure 2.2(b) cannot be reclaimed purely based on the count. Therefore, reference counting is usually supplemented by another garbage collector capable of reclaiming data structures forming cycles, like mark-sweep [46]. An alternative solution is to introduce *weak pointers*; pointers that do not increase the counts of the objects they reference [82]. However, weak pointers require the programmer to explicitly manage the lifetime of pointers which is prone to errors.

Reference counting does not compact the memory, so fragmentation may occur; more advanced reference counting algorithms tackle this problem [158]. In addition, reference counting introduces unexpected pauses during program execution. These pauses occur when reclaiming an object triggers a long chain of counter update operations and deallocations, also known as a *cascade*, that take a long time to process. For example, eliminating the last pointer to the root of a tree data structure will cause every tree node to be reclaimed. Once again, there is research that attempts to mitigate these problems [185].

2.1.3 Comparing Basic Garbage Collection Algorithms

The main features of the basic garbage collection algorithms are compared in Table 2.1. Mark-compact and copying are the only collectors that directly tackle fragmentation because they compact the memory. Allocations alongside these collectors are simple and have predictable run-times as the free space is clustered at one end of the memory as shown in Figure 2.3(a). Therefore, an allocation only involves moving a pointer by the requested amount of storage space.



(a) Compacted memory.

(b) Fragmented memory.

Figure 2.3: Fragmentation segments the free memory space. Allocating is complex, time-consuming and prone to failure when the memory is fragmented. In contrast, compaction clusters the free and live memory, so allocations are simple to implement.

In contrast, algorithms that do not compact the memory require complex allocation routines that are prone to failure. This is because the free space is segmented by blocks of allocated memory as shown in Figure 2.3(b), so the algorithm must operate lists of free memory blocks until one with enough space to satisfy the allocation request is found. Also, the run-time of allocate operations increases as fragmentation occurs. In the worst-case, suitable space cannot be found to fulfil an allocation and the system fails. These issues degrade reliability and are unacceptable in embedded devices that are often expected to run for a long time in inaccessible locations.

Reference counting generally has the lowest memory overheads as it only requires additional storage for the counts. Previous research showed that very few bits are needed to store these counts [156]. Mark-sweep and mark-compact have higher memory overheads because they need space for the processing stack and mark flags. In contrast, copying collectors have the highest memory overheads as the usable memory space for dynamic allocations is effectively halved.

Tracing algorithms always start garbage collection from a root set of pointers. They identify the live objects and then reclaim those that are dead. This enables them to reliably reclaim any kind of data structure including those forming cycles. However, tracing garbage collectors must scan all reachable objects before reclaiming the garbage in batch. So there is a delay from the time that an object becomes unreachable until it is reclaimed. In contrast, reference counting operates incrementally alongside the program. For example, a count for an object is incremented as a pointer is written to a memory location. Similarly, the objects are reclaimed incrementally when their counts drop to 0, so the collector only traverses unreachable objects. But this is also a weakness as data structures forming cycles cannot be reclaimed without using an alternative collector.

2.2 Generational Garbage Collection

Researchers observed that the majority of objects become garbage shortly after they are allocated [101, 178]. These objects that die young are later reclaimed by the collector. But objects

that are retained longer are repeatedly and unnecessarily processed by the garbage collector. This is particularly problematic with copying algorithms that must evacuate every live object during a collection cycle.

Generational garbage collectors segregate objects by age in an attempt to reduce run-time overheads [13, 101, 178]. They partition the memory space into two or more *generations* that are garbage collected separately. Objects are allocated into a young (*nursery*) generation and are promoted (*tenured*) to other generations as they become older. Nurseries have higher object mortality rates, so they are garbage collected more often than generations with older objects. But performance is improved because it takes less time to collect a nursery than the full memory space. The root set of pointers for a nursery collection includes the regular roots as well as pointers from older to younger generations. This is because young objects that are referenced from outside the nursery must be retained, but objects from older generations are not scanned when the nurseries are collected.

Parameters, such as when to collect, generation sizes, among others, are normally all configurable by the user. For example, the tenuring parameter can be set to the number of collection cycles that the object has survived. In addition, the different generations may be collected using different garbage collection algorithms.

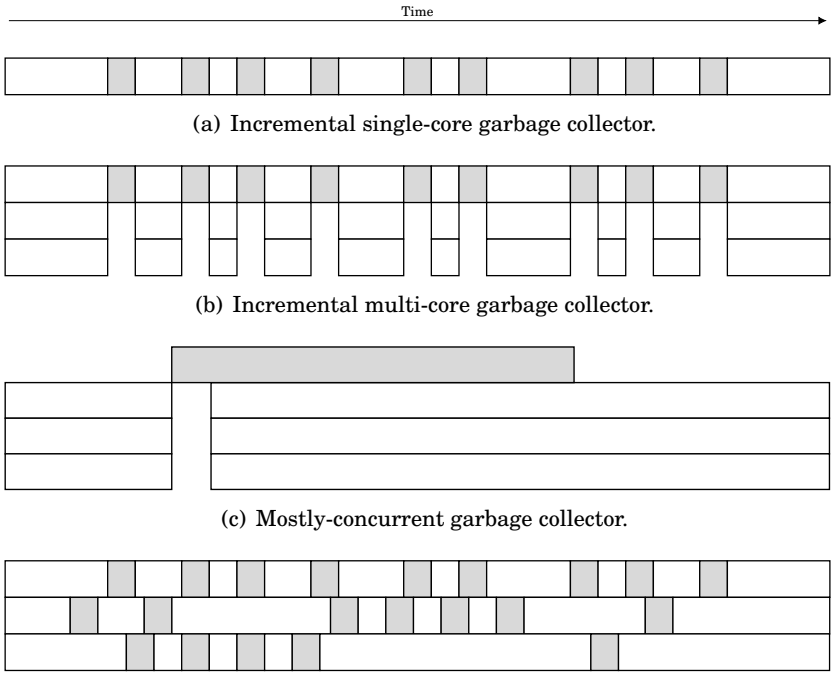
Generational garbage collectors are widely used due to their performance benefits. But they are complex to design and configure correctly. Compared to the basic algorithms, they have higher memory overheads due to the need to record references from older to younger generations. Generational collectors improve the run-time in the average-case only, so they are not used in real-time systems where the worst-case performance is critical [46, 79]. For these reasons, we do not consider generational garbage collection further in this thesis.

2.3 Incremental and Concurrent Garbage Collection

The basic algorithms discussed in Section 2.1 assume that the program is paused for the full duration of a collection cycle. These collectors effectively *stop-the-world* and are only invoked when the program has exhausted the free memory space. Therefore, stop-the-world collectors can be used in systems where pauses are not important. But they are unsuitable for interactive and real-time applications.

Incremental algorithms attempt to reduce the duration of garbage collection pauses. These collectors divide their operation into small units of work called *increments* that are executed interleaved with the user's program as shown in Figure 2.4(a). The program is paused for the duration of each increment, even in multi-core systems as shown in Figure 2.4(b). Ideally, the work increments are sufficiently small so that the pauses are very short.

Concurrent garbage collectors further eliminate pauses by allowing the user's program to execute in parallel with the collector. For example, Figure 2.4(c) shows a system where a core is



(a) Incremental single-core garbage collector.

(b) Incremental multi-core garbage collector.

(c) Mostly-concurrent garbage collector.

(d) On-the-fly incremental garbage collector.

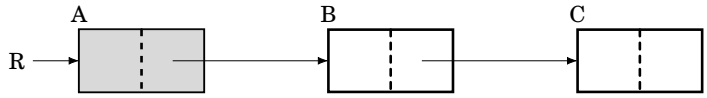
Figure 2.4: Incremental and concurrent garbage collection. Each bar represents the execution in a single core. White regions are the execution of the user's program threads and shaded regions correspond to the collector's execution. This illustration was partially extracted from [88].

fully reserved to execute a mostly-concurrent collector. A short pause is only introduced at the beginning of the collection cycle to perform some synchronization. In the literature, concurrent collectors that do not introduce this synchronization pause, as shown in Figure 2.4(d), are called *on-the-fly* [88]. Concurrent collectors are mainly intended for multi-core systems where it is possible to take advantage of parallelism. However, embedded systems are normally single-core, so it is common to use incremental collectors for these devices instead of concurrent ones.

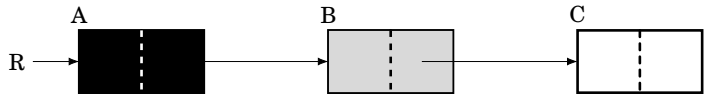
Incremental and concurrent garbage collectors make real-time operation possible because pauses are reduced significantly when compared to stop-the-world collectors. However, the user's program is no longer paused for the full duration of the collection cycle, so race conditions can occur. In Section 2.4 we will discuss techniques to prevent these race conditions and maintain correctness when using incremental or concurrent collectors.

2.4 Correctness of Incremental and Concurrent Collectors

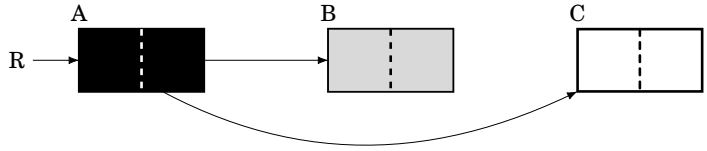
The user's program is not paused while incremental and concurrent collectors run, so it is possible that reachable objects mistakenly remain unmarked at the end of the collection cycle. The *tri-color* abstraction proposed by Dijkstra et al is used to reason about these issues [57, 58]. Consider the



(a) A is marked (shaded gray) as it is reachable from root R.



(b) A is fully processed (shaded black) and B is marked (shaded gray).



(c) The user's program stores a pointer to unmarked (white) object C in black object A and deletes the pointer from gray object B. C will remain unmarked at the end of tracing.

Figure 2.5: Sequence of events that results in reachable objects remaining unmarked after tracing when an incremental collector runs concurrently with the user's program.

objects A, B and C with the pointers shown in Figure 2.5; all objects are *white* at the beginning of the collection cycle since they are unmarked. A is marked first and shaded *gray* in Figure 2.5(a) because it is reachable from root R. The collector then scans A and finds a pointer to B. A is shaded *black* in Figure 2.5(b) because it has been both marked and scanned while B is shaded gray as it is marked but not scanned yet. However, the user's program concurrently stores a pointer to C in A and deletes the pointer to C from B before B is fully processed, i.e. shaded black, as illustrated in Figure 2.5(c). C will not be marked at the end of tracing because A will not be scanned again. Therefore, C will be incorrectly reclaimed potentially causing a failure.

Garbage collectors prevent live objects from being reclaimed incorrectly by enforcing either one of the following two invariants [88]:

Strong tri-color invariant: Black objects do not contain pointers to white objects.

Weak tri-color invariant: White objects referenced from black objects are also reachable from gray objects either directly or indirectly through a chain of white objects.

Garbage collection *read* and *write barriers* are used to prevent the correctness problems described above by preserving either of the invariants when using incremental or concurrent algorithms. The barriers perform some action, like marking an object, as pointers are inserted or deleted. Garbage collection algorithms, along with their barrier techniques, can be classified in two groups according to how the roots are colored: *black* and *gray mutator*.

2.4.1 Black Mutator

The user program's root set may only contains pointers to black objects if the garbage collector operates with a *black mutator*. The following barriers are used for these collectors:

Read barrier: The idea was originally proposed by Baker for an incremental copying collector that maintains the strong tri-color invariant [32]. Baker's algorithm performs a flip and scans the roots within a single work increment. Then the collector performs a small amount of work during each subsequent increment until all reachable objects are shaded black. However, the collector must ensure that black objects never contain pointers that reference white objects, otherwise the situation illustrated in Figure 2.5 could occur. The read barrier enforces this invariant by processing for marking, i.e. shading gray, every pointer that the program loads into the roots.

Write (deletion) barrier: This barrier processes for marking a pointer that is being *deleted* from an object while storing to memory. In other words, the write barrier marks the object referenced from the location that is about to be overwritten. This ensures that white objects are never referenced from black objects only, so the weak tri-color invariant is maintained. Collectors using write deletion barriers, like Yuasa's [195], are also known as *snapshot-at-the-beginning* because they retain all objects that are reachable at the start of the collection cycle.

2.4.2 Gray Mutator

A garbage collection algorithm is said to use a *gray mutator* if the program's root set may refer to objects that are black, gray or white according to the tri-color abstraction. Dijkstra et al proposed a garbage collector for a gray mutator that enforces the strong invariant using a *write (insertion) barrier* [58]. The write barrier marks the pointers being written (or *inserted*) into memory to ensure that black objects never contain pointers to white ones. Steele proposed another write (insertion) barrier that changes the color of the object where a pointer is written from black to gray [169]. So the collector scans the formerly black, now gray, object once again to ensure that reachable white objects are marked. However, garbage collectors operating with gray mutators allow the program to load pointers to white objects into the roots after root scanning. So compared to black mutator algorithms, gray mutator ones have a more complex termination condition because the roots must be scanned multiple times to ensure that all reachable objects are shaded black.

2.4.3 Incremental and Concurrent Compacting

Another problem with incremental and concurrent collectors occurs when the program accesses an object that is currently being compacted. The collector must ensure that the correct memory

location is accessed to guarantee the program's correct behavior. Read and write barriers implementing address forwarding operations can be used to address the problem. For example, every object in Brooks' collector contains a pointer to the correct copy of the object that the program must access [42]. Similarly, Steele's collector contains flags to indicate whether an object has been relocated [169]. But most of these incremental collectors rely on each object being copied atomically before the program can access it, so delays are introduced.

2.4.4 The Cost of Read and Write Barriers

Incremental and concurrent garbage collectors are an absolute necessity in soft and hard real-time systems. But read and write barriers incur high performance overheads, so incremental collectors are less efficient than their stop-the-world counterparts. According to Zorn, the cost of software-implemented read and write barriers accounts for as much as 20% and 8% of the program's run-time respectively [197]. However, more recent studies have shown that the impact of collection barriers has progressively decreased in modern large-scale systems [37, 194]. Read and write barriers can be implemented in several ways:

- Compiling additional instructions into the program's code to check for read and write barrier conditions. This technique can significantly increase code size.
- Using the virtual memory system to protect pages. Appel et al pioneered this technique to efficiently enforce a version of Baker's read barrier [14]. However, this can be costly as it constantly forces the processor to switch context. It is also unsuitable for embedded systems that do not have hardware support for virtual memory.
- Introduce special-purpose hardware to accelerate read and write barriers. We discuss this technique in Chapter 3.

Read and write barriers are essential to reduce the duration of individual pauses, yet they incur significant performance penalties. The design of the barrier must carefully consider the needs and hardware resources in the system. For example, read and write barriers using virtual memory are unsuitable for embedded systems that lack these facilities.

2.5 Characterizing Garbage Collection Pauses

As discussed in Section 1.2, real-time systems are expected to perform tasks correctly and deliver results on time. So it is important that garbage collection pauses, due to operations like read and write barriers or work increments, do not delay the user's program such that deadlines are missed. However, traditional statistical metrics, like maximum or average duration of pauses, are insufficient to characterize collection algorithms. For example, consider a real-time system performing a task that takes 15 ms in every 30 ms period. It is unclear whether a garbage collector

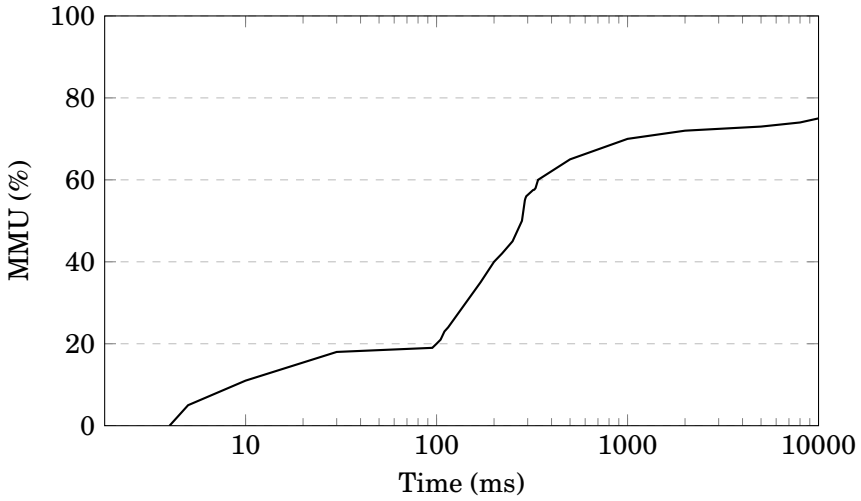


Figure 2.6: Minimum Mutator Utilization (MMU) plots graphically characterize the distribution and duration of garbage collection pauses.

that pauses for a maximum of 10 ms is suitable for the real-time system because it is not know how many of these pauses could occur in a 30 ms period. Therefore, measures to characterize the distribution of pauses are needed [88].

Cheng and Blelloch invented the *Minimum Mutator Utilization (MMU)*, a useful metric commonly used to characterize the distribution and duration of garbage collection pauses [50]. MMU defines the utilization as the fraction of time that the user's program executes in a given time window. A MMU plot is a graphical representation of the minimum utilization for many time windows. For example, the MMU in Figure 2.6 shows that 10% of the processor's time is available for program execution in any 10 ms window. The plot also shows the total fraction of time used for garbage collection at the y-intercept and the maximum pause time at the x-intercept.

MMU is a valuable measure to characterize pauses in soft real-time systems. But it does not guarantee that a garbage collector is hard real-time because the MMU is an empirical measurement. There could be corner cases resulting in lower minimum utilization that were not captured by the run used to produce the MMU. As we will discuss in Chapter 3 and Chapter 6, hard real-time garbage collectors must be shown to not violate timing requirements in the absolute worst-case.

2.6 Identifying Pointers

At the heart of any garbage collector is the ability to identify unused objects. A safe assumption is that an object is unused if the program does not hold a pointer to it because the program could never (legally) access that object again. But this raises the question of how collectors can distinguish pointers from data values when scanning the program's memory. A solution is to

assume that a pointer is any word in memory or the registers whose value corresponds to the address of an allocated object in memory. The advantage of these *conservative* collectors is that they do not require type information to operate correctly, so they can be integrated with weakly typed languages like C [198].

Conservative collectors have the disadvantage that data values occasionally alias pointers, so garbage objects would be retained longer than needed. Thus, it is difficult to analyze such a collector's real-time behavior because the amount of time that garbage objects will remain in memory is unknown. In fact, conservative collectors can increase heap memory requirements by 0.01-6% [157], or as much as 30-150% [198]. These collectors also incur performance overheads because they implement heuristics that attempt to minimize pointer aliasing. For example, a collector could check whether a word contains a value within the valid range of memory addresses supported by the machine, so it is potentially a pointer, or outside that range, so it has regular data. Implementing such heuristics typically requires executing the equivalent of about 30 RISC instructions for every word examined during tracing [88]. Another critical problem is that conservative collectors do not work well with compaction because the collector cannot reliably distinguish whether a word contains a pointer that needs to be updated as it references a relocated object, or an integer that must remain unchanged.

The alternative to conservative collectors is using type information to unequivocally identify the set of live objects. These collectors are also called *exact* and we discuss four techniques to implementing them: *bit stealing*, *extra tag bits*, *partitioning* and *type maps*. Bit stealing is a tagging technique that uses one or more bits in a word to indicate whether the value is a pointer or simply data [88]. The most or least significant bit of every word is normally repurposed as a tag. But it has the disadvantage that the range of integer values that the word can represent is reduced. For example, the maximum unsigned integer is $2^{31} - 1$ instead of $2^{32} - 1$ on a 32-bit processor if a bit is used as a tag. This causes compatibility problems with some programming languages, such as C and C++, because standard integer types are not easily supported. In addition, most Instruction Set Architectures (ISA) are not designed to disregard the stolen tag bit when executing instructions, like add and subtract, so extra code is needed to deal with these problems at the expense of performance.

The compatibility and performance problems caused by bit stealing can be tackled by adding extra tag bits to every word using hardware. For example, a word can be thought of as having 33 bits: 32 for data and 1 for tag. Instructions operate on the natural word size once again at the cost of memory overheads for the tags. The extent of these overheads depends on how the tag bits are implemented in hardware [85]. The tags can be stored as vectors in dedicated memory components, but this increases hardware requirements. Another approach is to extend the word size of the physical memory component. This is difficult to achieve in large machines, like desktops and servers, because off-the-shelf external memories are mostly manufactured in standard bit widths. However, extending the word size is not a problem in embedded devices that

Technique	Compatibility	Performance Overheads	Memory Overheads	Exact
Conservative	High	Medium	Medium	No
Bit stealing	Low	High	Low	Yes
Extra tag bit	High	Low	Medium	Yes
Partitioning	Low	Low	Low	Yes
Type map	Low	High	High	Yes

Table 2.2: Comparison of techniques to identify pointers in memory.

mostly have internal memories. In this case, the memory overheads for a 1-bit tag are about 3% in a 32-bit machine.

Another approach to exact garbage collection is to use the type information from the objects. It is possible to identify the type of every member of an object in strongly typed languages, such as Java and C#, but not in weakly typed languages because values can be cast to arbitrary types. The type information can be supplied to the collector in two ways. Pointers and data can be segregated into partitions within every object, so it is easy to identify the values that need to be scanned during marking [104]. This idea has low performance and memory overheads, but it is difficult to implement alongside languages that support polymorphism. The type information can also be supplied as a type map that indicates the layout of each object in memory. However, type maps incur memory and performance overheads as they require storage space and the collector must execute instructions to decode and use them.

A comparison of the collection approaches to distinguishing pointers from data is shown in Table 2.2. The memory requirements of conservative collectors are difficult to analyze statically, so they are unsuitable for real-time systems. Bit stealing has low memory overheads that are beneficial for embedded systems. However, it is difficult to support standard integer types, so bit stealing is incompatible with much existing software. Partitioning and type maps cannot be used with weakly typed languages. In addition, partitioning limits flexibility when the garbage collector is implemented in hardware because the object layout must be hard-coded in the runtime system. Conversely, the more flexible type maps are difficult to use by hardware collectors and incur high memory overheads. We conclude that adding extra tag bits is the best choice for embedded real-time systems if the hardware overheads are low.

2.7 Summary

In this chapter we described the three conflicting goals of garbage collection. We explained four basic garbage collection algorithms: reference counting, mark-sweep, mark-compact and copying. We also discussed the advantages and disadvantages of these algorithms. Reference counting cannot reclaim data structures forming cycles and introduces pauses that are unsuitable for

real-time programs. Copying collectors have very high memory overheads, but their performance is better compared to mark-compact and mark-sweep collectors. In addition, only incremental or concurrent garbage collectors are suitable for real-time systems.

We considered how generational garbage collectors can be used to improve performance. But we concluded that generational algorithms are unsuitable for real-time systems because they optimize the average-case instead of the worst-case run-time. We explored various strategies for distinguishing pointers from data. It was found that only exact hardware collectors implemented using extra tag bits are suitable for real-time systems as their memory space requirements can be analyzed statically.

We conclude that incremental and exact collectors are essential for real-time embedded systems. In addition, embedded devices often have limited resources and lack support for features, like virtual memory, so the design of the garbage collector must carefully consider the implementation challenges in those systems.

CHAPTER 3

REAL-TIME GARBAGE COLLECTION

Real-time requirements are a common denominator in many embedded systems. A 2017 survey by AspenCore found that about 60% of embedded projects require real-time capabilities [23]. This is because embedded applications usually entail devices communicating with each other and interacting with the physical environment; both tasks are subject to timing constraints. For example, a received network packet must be acknowledged before the connection times out. Similarly, a display must refresh sufficiently fast in response to stimuli to provide a good user experience. But garbage collectors introduce unpredictable pauses that cause devices to not meet their timing requirements, so modern languages are rarely used to develop real-time embedded systems.

Explicit memory allocators are also unsuitable for real-time systems [136]. These allocators have analytical worst-case run-times that are far detached from the algorithm's average-case, so the system may unexpectedly miss deadlines. In addition, explicit memory allocators are error-prone and often unreliable due to fragmentation. Without alternatives, programmers are forced to either write domain-specific, real-time dynamic memory allocators or use static memory. For example, the open-source networking library lwIP includes a custom allocator to manage storage for packet data [59]. However, domain-specific allocators cause safety and reliability problems as they are rarely tested and verified sufficiently. Domain-specific allocators and static memory management also waste space, are often inefficient and increase software complexity. This is because domain-specific allocators often place objects within fixed-size memory blocks and static memory management reserves the worst-case storage requirements at compile-time.

Previous research proposed specialized garbage collectors to tackle the memory management problem in real-time embedded systems. In this chapter we explain and compare notable works from the literature.

3.1 System Requirements and Garbage Collectors

Systems have different demands depending on their target application. These demands are passed on to the memory manager to meet specific performance, space and pause requirements. Therefore, garbage collectors have a wide design space as different techniques are appropriate depending on the system requirements.

Some systems must complete a task in the shortest time possible. Pauses are not important because there is little or no interactions with users or the physical environment. Stop-the-world garbage collectors are ideal choices for these batch processing systems as the overheads from read and write barriers are eliminated.

Long collection pauses are not desirable in soft real-time systems. Their requirements are to minimize collection pauses in the average-case, but the occasional long pause can be tolerated. Examples include mobile devices and information displays where failure to meet timing deadlines does not have serious consequences. The incremental collectors described in Section 2.3 are an optimal choice for these systems because the work increments can be spread out so that long pauses are avoided. In addition, generational collection techniques are used to reduce the performance cost of memory management as the complete memory space does not need to be repeatedly processed. There are many proposals for soft real-time collectors in the literature. In practice, most implementations of modern languages rely on soft real-time collectors, such as the Java HotSpot Virtual Machine and CPython [129, 143].

Predictability is of paramount importance in hard real-time systems. For example, a medical device, like a cardiac pacemaker, missing a timing deadline may result in the death of a patient. Hard real-time systems must be shown to not violate timing requirements because of garbage collection. So the run-time of any operation, such as loads, stores and memory allocations, must be bound by a small constant in the worst-case. These systems cannot rely on average or expected-case costs [123]. Also, hard real-time systems are often safety-critical and cannot suffer from unreliable behavior due to memory issues like fragmentation. Therefore, some form of compaction is required unless objects are allocated within fixed-size memory blocks only [55].

Existing hard real-time garbage collectors are incremental or concurrent and are implemented in software. However, unexpected collection pauses are not acceptable, so hard real-time systems are very sensitive to how the collection work is scheduled alongside the user's program [79]. In the remainder of this chapter, we survey the literature on software-implemented, hard real-time garbage collection and explore the two main scheduling approaches: *work-based* and *time-based*.

3.2 Work-Based Real-Time Garbage Collection

Work-based garbage collectors introduce a collection work 'tax' in connection with memory allocations. These collectors are incremental to ensure that the program's threads perform a variable amount of collection work depending how much memory they allocate. Also, the amount

of tax is generally configurable; for example, it may depend on the priority of the thread requesting the allocation.

Showing that work-based collectors do not violate timing constraints is relatively simple as the collection and allocation rate are strongly related. The system's allocation rate is extracted from the real-time program by using static analysis methods. The collection rate is estimated using a *timing model* of the collector; the model estimates the worst-case run-time of collection operations, like the duration of a mark-sweep cycle in a tracing collector, for a given program. From this and the configurable tax parameter, the amount of collection work performed by a thread when allocating is calculated to guarantee that the user's program is not paused unexpectedly. Therefore, all operations are bound by a constant depending on the configurable tax parameter.

3.2.1 Baker's Garbage Collector for LISP

Baker pioneered the work-based technique with his incremental copying collector for LISP. The collector uses a read barrier to ensure that the program only sees *to-space* references after a flip [32]. This prevents *from-space* references from being inserted into *to-space*. The read barrier checks every pointer loaded from memory. No action is taken if the pointer references a *to-space* location, although the check requires executing a test and a branch instead of a simple load. Loaded pointers to *from-space* locations are updated to reference *to-space*: a copy of the object is created in *to-space*, if it does not exist already, and the pointer is updated to reference the new copy. As a result, loads from memory may result in costly copying operations.

A flip initiates a collection cycle. This entails copying all objects directly reachable from the roots and forwarding the root pointers. The flip is an atomic operation: it must be run to completion before executing other operations. In addition, Baker's algorithm performs collection work when allocating, so the user's program is paused temporarily. The collection work increments are atomic to prevent race conditions when there are multiple threads. Therefore, implementations of software collectors, like Baker's, often disable interrupts when executing collection work increments or performing read barriers, but this causes event handling delays until ongoing collection operations are performed.

The collector manages the function call stack differently from other dynamically allocated objects to avoid performance penalties. The stack pointer is saved during the flip to indicate which locations the collector needs to scan. Then the stack is processed incrementally from top to bottom; discovered objects are copied and pointers updated to reference *to-space* locations. However, the algorithm's correctness relies on some modifications to stack operations that decrease performance. For example, the read barrier must be enforced when loading from the stack. Also, the collector's pointer into the stack must be updated when the stack is popped to avoid tracing unreachable locations.

The collector's memory requirements are high as it is based on copying. In theory, the

minimum memory requirements are twice the usable memory size. But in practice, the memory requirements are much higher and depend on the configurable tax parameter that trades space requirements for performance and responsiveness. The space overheads decrease as the user's program pays a higher tax per allocation. A high tax parameter also increases the worst-case run-time of allocations and interrupt latency because allocations are atomic.

3.2.2 Brooks' Garbage Collector for LISP

Brooks' proposal is a change to Baker's algorithm to improve performance [42]. Brooks observed that loads are more frequent that stores, so he modified Baker's collector to rely on a write barrier. That is, the collector marks the pointers that are stored into memory. This ensures that the program never introduces *from-space* pointers to *to-space* objects that have already been scanned. But the write barrier no longer prevents the program from seeing *to-space* references only, so a mechanism to always access the most up-to-date copy of an object is needed to guarantee correctness. Brooks solves this problem by forcing an indirection on every memory access using an address stored in a metadata word, known as a *header*, at the beginning of each object. The efficiency of this approach is demonstrated by comparing the code size of Brooks' indirection with Baker's need for additional test and branch instructions when accessing memory.

The timing properties of Brooks' collector are largely similar to Baker's. The function call stack is processed incrementally although the roots are scanned atomically during a flip. Brooks' collector must also scan the roots atomically at the end of the collection cycle instead of only at the start to ensure that pointers to *from-space* do not remain in the roots. The root scan operation at the end of the collection cycle is potentially run more than once until all live objects are marked and scanned. As a result, processing the roots causes significant overheads and multiple pauses in the worst-case.

3.2.3 The Treadmill

Baker proposed a further algorithm, the Treadmill, to eliminate the cost of copying from his original proposal [31]. The Treadmill is a mark-sweep collector that divides the memory space into equally-sized memory blocks. Each block has two pointers that are used to organize the full memory space into a circular doubly-linked list. The collector labels the memory blocks according to their status using the tri-color abstraction: white, gray and black. Objects can also be labeled as 'free'. Four pointers into the list are used to keep track of the block status. The run-time of collection operations is bound by a small constant because changing a block's label, such as when allocating, only involves simple pointer updates.

The Treadmill largely relies on the same principles as Baker's original copying collector. However, the Treadmill is not suitable for systems that allocate objects of arbitrary sizes. In addition, it suffers from internal fragmentation and has per block memory overheads corresponding to the list pointers.

3.2.4 Yuasa's Garbage Collector for LISP

Yuasa published a mark-sweep real-time collector that uses a write barrier to maintain the snapshot-at-the-beginning invariant [195]. The write barrier loads the location that is about to be overwritten during a store operation. If the loaded value is a pointer, it is processed for marking; discovered objects are marked and pushed into a stack to be scanned later. The advantage is that write barriers are executed infrequently compared to read barriers. However, Yuasa's write barrier requires a load before performing the store, so reads take a longer time. Also, Yuasa's collector incurs space overheads due to the marking stack.

The algorithm is non-copying to eliminate performance overheads. External fragmentation is not an issue as objects are allocated within fixed-size memory blocks. It is difficult to use the collector in systems allocating objects of arbitrary size. However, this was not a concern for Yuasa because his collector was originally intended for LISP where most objects are list cells with two components. In addition, objects are scanned atomically during allocations as in Baker's algorithm, but this is not a problem for Yuasa's collector either because LISP's cell objects are small and can be scanned quickly.

Yuasa's algorithm pauses the program at the beginning of a collection cycle to scan the roots. The root set consists of the processor's registers and the function call stack. But the run-time of root scanning is potentially unbounded because the stack is dynamic in nature and often very large. Yuasa solves this problem by using an alternative collection stack where the contents of the function call stack are copied for later processing. Yuasa states that this copy operation can be done at the beginning of the collection cycle in the background using Direct Memory Access (DMA). However, this incurs a performance impact and memory overheads to store the stack with the copied pointers. Also, many embedded processors cannot operate concurrently with the DMA, so the user's program would be paused while the function call stack is copied.

3.2.5 Garbage Collection for the Jamaica Virtual Machine

Siebert proposed a work-based mark-sweep collector that aims to mitigate the performance drawbacks of previous algorithms [159–164]. The collector was designed for the Jamaica Virtual Machine, a Java implementation intended for real-time systems. It allocates objects within 32 byte memory blocks, although the size is configurable. This eliminates the need to compact, but trades external for internal fragmentation. Multiple memory blocks can be allocated to accommodate objects larger than the block size. This has clear performance drawbacks when accessing large arrays and objects. Siebert mitigates these issues by organizing the blocks from arrays into tree data structures and other objects into linked lists.

The collector uses a write barrier to maintain the invariant that black objects never contain pointers to white objects. The write barrier marks pointers to white objects that are being stored into black objects. The marked objects are also added onto a list for later processing. The collector must scan each memory block atomically during marking, so choosing a large block size causes

long program pauses. Race conditions between the user's program and collector are prevented because thread switching occurs at specific synchronization points in the program. It is only necessary that the invariant is true at the synchronization points rather than between them. The code between synchronization points runs without interruption and performs collection work during allocations. The drawback is that identifying the synchronization points requires programmer intervention.

An innovation in Siebert's collector simplifies root marking. The collector has a single root pointer and every object referenced from the registers or the function call stack must also exist in memory. This ensures that program pauses during root marking are kept very short. But there is a performance cost because the program must constantly ensure that the references in the roots are backed up in memory.

3.2.6 Blelloch and Cheng's Multi-Core Garbage Collector

Blelloch and Cheng invented a copying collector with provable time and space bounds [39, 50]. The collector uses Baker's work-based technique to schedule work, but the algorithm is parallel and runs concurrently with the user's program across multiple cores. Compared to Baker's collector, Blelloch and Cheng's algorithm has significantly higher memory and performance overheads due to the complexity of maintaining consistency across all cores. It is also unsuitable for embedded systems that are often single-core rather than multi-core.

The collector is snapshot-at-the-beginning and uses a modified version of Nettle and O'Toole's write barrier [122]. This requires the collector to maintain two copies of every object in memory with the program only seeing *from-space* copies. The write barrier copies the object referenced by the pointer being overwritten if a copy does not exist already. It also updates all copies of the object being accessed. But the pointer written must also be marked, if the object accessed already has a copy, to ensure that *from-space* references are not introduced into *to-space*. The write barrier is very expensive, so Blelloch and Cheng proposed to only record the store in a log during the write barrier. The log is later processed during allocation operations. However, this requires additional storage space for the log and further performance overheads.

A flip and the termination of a collection cycle are complex operations in Blelloch and Cheng's collector because all processes must be stopped and the roots processed atomically, so program pauses are introduced. In addition, Blelloch and Cheng place some restrictions to facilitate the real-time analysis. First, collection work increments, such as root scanning and copying, are atomic and pause the program. Second, the system does not support nested interrupts. And third, allocations must be followed by the explicit initialization of every field within the new object before a further allocation is performed.

Blelloch and Cheng's collector scans the function call stack incrementally. The stack is divided into fixed size *stacklets* that must be scanned atomically (see Section 8.2.3.3). This facilitates bounding the time that the collector spends marking the roots. However, additional code is needed

to operate the stack; this causes overheads.

3.2.7 Ritzau's Reference Counting Garbage Collector

Ritzau developed a reference counting collector for real-time systems [140, 141]. The collector allocates objects within fixed-size blocks to avoid external fragmentation. Each block has a header with space for a pointer and the reference count. The blocks are initially organized in a free list that can be used to fulfil allocation requests. The collector uses *deferred reference counting* to reclaim dead objects. Blocks whose reference count reaches 0 are added to a *to-be-free* list instead of being reclaimed immediately. The to-be-free list is only processed when there is not enough memory in the free list to fulfil an allocation. In this sense, Ritzau's collector is work-based as the program must perform collection work proportional to the allocation size.

The advantage of Ritzau's collector is that reference counting does not require scanning the roots, so marking pauses are eliminated. However, the collector is incapable of reclaiming data structures forming cycles and it is not clear how this problem is addressed without affecting the collector's timing properties. There are also high memory overheads as each block's header is 12-16 bytes in size. Objects smaller than the block size will cause internal fragmentation. Multiple blocks are used to accommodate objects larger than the block size. These large allocations will incur further memory overheads due to the unused headers for each object. Large objects must be organized using linked lists or trees as proposed by Siebert [162], and as such they incur performance penalties.

3.3 Problems with Work-Based Real-Time Garbage Collection

Real-time programs must be carefully analyzed to determine the *Worst-Case Execution Time (WCET)* of every execution path. The worst-case corresponds to the longest time taken to execute a fragment of code regardless of how frequent the worst-case scenario occurs. Two properties are desirable to facilitate the worst-case analysis. First, the time taken to perform the operations, such as loads and adds, along every execution path must be bound by a small constant in the worst-case. Second, the worst-case and average-case execution time should ideally be similar. It might still be possible to meet real-time constraints without the second property. However, the system's resources are likely to be underutilized to ensure that deadlines are met in the worst-case that may rarely (if ever) happen.

Work-based garbage collectors have been shown to fulfil the first property. Baker's, Siebert's and other work-based collectors are supplemented by a formal analysis showing that store, load and allocation operations can be bound by a small constant. But these collectors do not meet the second property, so they are impractical. This is because the collection work performed in connection to memory operations increases the worst-case time and memory bounds in unrealistic ways [28, 55, 79].

Consider Baker's copying algorithm to illustrate the problem. The program performs collection operations as tax for memory allocated. The worst-case duration of an allocation will occur during and shortly after a flip when many objects are discovered and copied. Programs do not perform as much collection work when allocating at other times. So there is an asymmetry in the run-time of operations that incur collection work [55]. However, the execution time analysis must assume the worst-case for every allocation and memory access. This results in excessive processing and memory resources being reserved even though they are not needed. A related issue is caused by the uneven spread of collection work. In Baker's collector, the run-time of read barriers and allocations shortly after the flip approaches the theoretical worst-case run-time. The user's program will be overloaded with collection work during this time if it performs several allocations or memory accesses. Therefore, a fair share of the processor is not guaranteed for the user's program shortly after the flip. Time-based real-time garbage collection, discussed in Section 3.4, partially addresses these deficiencies.

3.4 Time-Based Real-Time Garbage Collection

Most modern real-time collectors are time-based. In this approach, collection work is scheduled in a separate execution thread and considered as another task in the traditional scheduling analysis for the real-time system. The analysis uses the program's allocation and collection rates to statically estimate the system's memory requirements and the amount of time that the collection thread must be run to guarantee that deadlines are always met. Similar to the work-based approach, the allocation rate is extracted from the real-time program using static analysis methods and the collection rate is estimated using the collector's timing model.

Time-based garbage collection successfully addresses the drawbacks of work-based collection. But the scheduling analysis of the real-time system is more complex when using time-based collectors because the collection thread needs to be scheduled. This section describes time-based collectors in the literature.

3.4.1 Henriksson's Low Priority Garbage Collection

Henriksson published one of the earliest works on time-based garbage collection. He proposed a copying collector based on Bengtsson's [33] and extensively analyzed how it could be executed in a separate thread [79, 142]. Henriksson's proposal schedules the collection thread using the *slack stealing* technique. The collection thread is assigned a low priority and is scheduled when high priority, hard real-time threads are not running. The collector only runs until enough work has been performed to guarantee that high priority threads will not be paused. Any spare processor capacity not used by high priority or collection threads is used to execute low priority, soft real-time threads. However, the low priority threads are taxed for memory allocations with the usual work-based strategy.

The collector relies on a Brooks-style write barrier to ensure that pointers to *from-space* locations are not written into *to-space* objects. *Lazy evacuation* is also used to mitigate the overheads of write barriers in high priority threads. The idea is that space for a newly discovered object is only reserved during a write barrier; the bulk of the copying work is deferred to the garbage collection thread. In addition, every memory access must follow the usual Brooks address indirection using a pointer in the object's header. Interrupts must be disabled when performing pointer operations as the write barrier is atomic. Other operations, such as procedure calling and object copying, are also atomic. This causes jitter as interrupt handling is potentially delayed.

Henriksson's object headers incur high memory overheads of up to 16 bytes per allocated object. Each header has space to accommodate the forwarding pointers and collection information such as references to type maps. The thread function call stacks are considered part of the root pointers and are scanned incrementally. Each call stack is actually a data structure with two independent stacks. The first stack contains the regular call frames created by the program. The second stack only has references to addresses that contain pointers in the first stack. This increases the programs memory and performance overheads as pushing and popping values to the stack actually involves operations in two data structures.

3.4.2 Metronome

The Metronome is a time-based real-time collector proposed by Bacon, Cheng and Rajan [27, 28]. It is a mark-compact snapshot-at-the-beginning algorithm that aims to reduce copying. The algorithm divides the memory space into pages of configurable size. Each page is further divided into blocks of equal size. Allocations are served from the page with enough free space and block size that most closely matches the requested memory size. Sparsely allocated pages are collected during a *defragment* stage by copying the live objects to more occupied pages. This reduces the amount of copying required.

The collector and program threads are interleaved using a fixed schedule. This causes long interrupt latencies as the collector cannot be preempted arbitrarily. In addition, the program thread's procedure call stacks are part of the roots and cannot be scanned incrementally. Therefore, the Metronome requires the stack size to be tightly bound so that it can be scanned atomically within a collection work increment. The Metronome uses a read barrier to maintain correctness and Bacon, Cheng and Rajan suggest that compiler optimizations can be used to mitigate its performance overheads. However, Detlefs observes that these optimizations require interrupts to be disabled for a potentially long time, so interrupt latencies are increased [55].

An analysis of the Metronome's real-time properties features in Bacon, Cheng and Rajan's work. This analysis requires program parameters such as the allocation rate, the size of live data, the number of pointers in memory and some scheduling information. However, their formulation is not rigorous as the worst-case run-time bound for the defragment stage relies on empirical observations rather than analytical bounds. It is not difficult to see how programs will cause

much higher worst-case run-times for collection operations.

3.4.3 Kim et al's Copying Garbage Collector

Kim et al observed that Henriksson's approach to scheduling collection work yields high memory overheads to avoid unexpected pauses [91–93]. So Kim et al schedule the garbage collection thread as an aperiodic task. Through experimentation, they find that their real-time collector reduces memory overheads by 14-38% compared to Henriksson's approach. Kim et al also present an analysis of the collector's real-time properties. They supplement their work with a proposed strategy to statically estimate the program's live size which is an essential parameter for scheduling real-time garbage collectors.

Kim et al's algorithm is largely similar to Henriksson's as both are copying collectors relying on Brooks-style write barriers and lazy evacuation. However, Kim et al's write barriers only log stores in an *update entry* instead of lazily evacuating newly discovered objects. The *update entry* is checked at the end of the collection cycle and all pending objects are evacuated. This logging technique reduces the write barrier overhead for the program, but increases the amount of collection work and storage space requirements. It is also unclear how Kim et al's algorithm handles *update entry* overflows.

Kim et al's algorithm requires the *from-space* memory to be zeroed during the flip at the end of the collection cycle. Hardware is used to ensure that the memory is zeroed in constant time so that the flip is atomic. Other details of the collector are unclear, such as root and stack scanning.

3.4.4 Chang's Hybrid Garbage Collector

Chang proposed a hybrid real-time garbage collector aiming to reduce memory and performance overheads compared to previous time-based collectors [46–48]. It combines Ritzau's reference counting technique with basic mark-sweep [141]. The hybrid design is motivated by Hampton's observation that data structures that do not form cycles account for 44-100% of the total memory usage with an average of 81% [77]. Chang expects that most objects will be reclaimed immediately after they become garbage using reference counting and only objects forming cycles will be reclaimed by the mark-sweep collector. Therefore, most garbage can be recycled very quickly and less memory is needed to ensure that the user's program is never paused.

The hybrid collector allocates objects within fixed-size blocks as it does not compact, so it suffers from internal fragmentation. Every block has a word that is used to group them in linked lists. Objects are allocated using one or more blocks, but the first block contains a header that is three words in size. The header has two reference counts: the first counts the references from the roots while the second counts references from other objects. The collector relies on two types of write barriers to ensure that white objects are not referenced from black objects. The *write-barrier-for-objects* only adjusts the object counts when references are added or removed from other objects. In contrast, *write-barrier-for-roots* adjusts the object's root count when a reference

is added or removed from the roots. The write-barrier-for-roots eliminates the need to scan the root set because references from the roots are tracked by counts, so the collector is simplified. However, the roots are normally modified very frequently with pointers constantly being added and removed, so we expect that maintaining the root reference count incurs high performance overheads.

Chang's collector is scheduled using the dual-priority approach. There are two collection tasks to schedule: *reclaiming* and *tracing*. The reclaiming task is run at high priority and reclaims memory blocks in the *to-be-free* and *white-list-buffer* lists. The to-be-free list contains memory blocks from garbage objects found using reference counting. The white-list-buffer list contains garbage objects found by the tracing collector. The tracing task is periodic and runs the tracing collector at low priority. Chang's expectation is that most garbage objects are found via reference counting, so the reclaiming task is run more often than the tracing task.

Compared to incremental mark-sweep, we expect the hybrid collector to incur significant memory and performance overheads when the majority of objects are linked in data structures forming cycles. In this case, the inherent costs of reference counting are redundant. Also, reclaiming garbage objects forming cycles takes longer in Chang's collector because the tracing task runs at low priority. Therefore, using the hybrid collector can be counterproductive in some systems. Another disadvantage of Chang's collector is that its real-time analysis requires measuring the amount memory from objects forming cycles, but no techniques are proposed to estimate this and it is unclear whether the programmer can provide the information.

3.4.5 Garbage Collection for Safety Critical Java

The Real-Time Specification for Java (RTSJ) attempts to work around the limitations of garbage collectors for real-time systems [40]. RTSJ's approach is to create a new type of thread with explicitly managed scoped memory for dynamic allocations. The scoped memory is not garbage collected; objects are deallocated in bulk by reclaiming the full scope. But RTSJ imposes restrictions on the programmer to enforce the safe use of memory. Schoeberl points out that these restrictions are too strict and proposes a time-based garbage collector to regain the simplicity of Java in real-time systems [147].

Schoeberl's time-based collector is run in its own thread [147, 148, 151]. It is a copying snapshot-at-the-beginning collector that uses a write barrier and handles to avoid having to update pointers after copying. Each handle is associated with a location in memory that contains the object's address, size and other administrative information. Schoeberl's algorithm performs an extra sweep stage at the end of copying that iterates over every handle and reclaims those that correspond to dead objects. There is also a *clear* stage that zeroes the *from-space* memory in preparation for the next collection cycle. However, Schoeberl did not provide a timing model of his collector as required by real-time systems.

The majority of the algorithm is implemented in software, so operations like the write barrier

need to be protected by locks to prevent race conditions between program and collector. However, Schoeberl and Puffitsch optimize the copying operation using hardware (see Section 4.1.4) [150]. The implementation of the copy accelerator is microcoded and takes up to 27 clock cycles to copy each word of data.

Schoeberl's collector is intended to support the Safety Critical Java (SCJ) standard [103]. Programs compliant with SCJ are composed of an *initialization* followed by a *mission* stage. The initialization stage is assumed to not be hard real-time, so it is collected in a stop-the-world fashion. The mission stage is hard real-time, but it is assumed that the collector only runs when the thread procedure call stacks are empty. Therefore, the root scanning operation is greatly simplified because the stacks do not need to be processed. In addition, Schoeberl observes that some objects allocated during the initialization stage are never reclaimed. So Schoeberl partitions the memory into *immortal* and *garbage collector* spaces where initialization and mission objects are allocated respectively. Partitioning improves performance because the immortal space is only scanned in search for pointers; it is never compacted. These simplifications to the collector are beneficial for SCJ, but it is not clear whether the collector can be easily ported to support other programming languages and standards like Python.

3.5 Problems with Existing Real-Time Garbage Collectors

Work-based and time-based garbage collectors require the system to be supplied with extra memory to avoid unexpected pauses, but the requirements are often impractical. Previous studies indicate that existing real-time collectors increase memory requirements by factors of 1.6-8 [28, 50, 87, 92]. Also, the collectors are implemented in software, so the user's program must regularly pause for the processor to execute collection operations. Therefore, real-time collectors only guarantee as little as 40-50% of the processor's time for application work [28].

Another problem with existing real-time collectors is that their hard real-time analysis formulations are often incomplete, like Schoeberl's, or flawed, like the Metronome's. Also, the proposed static timing analysis techniques for garbage collection rely on information extracted from the program, like the allocation rate, to estimate the system's memory requirements or create a real-time schedule. But there is little research into automated tools that programmers can use to facilitate this task. As a result, existing real-time garbage collectors are unusable in practice.

3.6 Tax-and-Spend: An Alternative Scheduling Approach

The work-based and time-based scheduling approaches only target a limited subset of applications and environments. For example, Metronome is intended for single-core or small multi-core systems while Henriksson's slack stealing collector is suitable for periodic applications and does not work well under high load. So Auerbach et al proposed the *tax-and-spend* scheduling approach

that combines the work-based and time-based strategies [25]. The idea is that tax-and-spend leverages the benefits of both work-based and time-based collectors and is sufficiently flexible to be used in many different applications and environments.

Tax-and-spend has two mechanisms to schedule collection work. First, there are dedicated garbage collection threads that are interleaved with program execution, for example, during any available slack or idle time. However, this concurrent collection work may not be sufficient to prevent long pauses. So the second scheduling strategy is to tax the program threads with collection work in a manner suitable to achieve that thread's utilization target.

Auerbach et al observed that tax-and-spend provides 3 times shorter latencies and better processor utilization when compared to Metronome. But tax-and-spend requires the collector to support both incremental and concurrent operation which increases implementation complexity. Also, tax-and-spend is intended for large multi-core systems as opposed to embedded devices. These multi-core systems are normally soft real-time, instead of hard real-time, because they rely on features, like caching and virtual memory, that have highly unpredictable latency. For these reasons, we do not consider tax-and-spend further in this thesis.

3.7 Real-Time Garbage Collectors for Multi-Core Systems

The garbage collectors described in this chapter are suitable for single-core and small multi-core systems. But these collectors do not scale well to large multi-core systems. Specifically, it is difficult to ensure that real-time collectors do not incur high overheads and long pauses when compacting to prevent fragmentation in multi-core systems. Pizlo et al proposed a lock-free concurrent garbage collector called Stopless to address this problem [132]. Stopless uses an intermediate *wide* copy of an object being compacted that stores the object's contents along with status flags for every word. A collection barrier on every access uses the status flags to ensures that the user's program reads or writes the correct memory location using *Compare-And-Swap (CAS)* atomic operations.

In further work, Pizlo et al proposed two further collectors, Chicken and Clover, based on Stopless that eliminate the need for the wide copy at the expense of longer worst-case performance [133]. Chicken eagerly assumes that the user's program will not access an object being compacted and simply aborts the copy operation if this does occur. So Chicken's read and write barriers are simpler compared to Stopless', but the copy aborts may cause fragmentation. McCloskey et al independently proposed a solution very similar to Chicken called Staccato [110]. Clover is similar to Stopless, but uses a randomly generated value α to signal that a field of an object has been copied. Clover's write barrier detects when α is stored to memory and pauses the user's program until compacting finishes. However, the likelihood of these pauses occurring is very low if α is selected appropriately, so Clover mostly runs lock-free and without long pauses.

The concurrent garbage collectors described in this section are intended for soft real-time

systems as observed by Pizlo et al [132, 133]. This is because multi-core systems usually rely on caches and virtual memory. Also, the collectors often cannot guarantee progress, for example, because object copies are aborted, and their hard real-time properties are not formally analyzed.

3.8 Summary

The design of hard real-time garbage collectors has not changed significantly since Baker's original proposal [32]. The differences between early work-based and modern time-based approaches mostly lie in how collection work is scheduled to balance memory requirements, performance overheads and pauses.

As seen in the previous sections, work-based approaches tax the program with collection work in exchange for memory allocations. These collectors focus on bounding the worst-case run-time of all operations that incur collection work with a small constant. But the average-case and worst-case run-times of these operations differ substantially, so the performance and memory overheads due to garbage collection are often infeasible. Time-based collectors address the limitations of work-based collectors at the expense of increased complexity in the scheduling analysis of the hard real-time system.

The main features of the work-based and time-based collectors surveyed in this chapter are summarized in Table 3.1. The collectors are mostly implemented in software and eventually interfere with the program's execution to run read or write barriers, trace objects, adjust reference counts or copy objects. These operations are often difficult to perform incrementally and may require interrupts to be disabled, so the system's responsiveness to events is impaired. In addition, efficiently implementing function call stack scanning is often complex due to the stack's dynamic nature. As a result, existing hard real-time garbage collectors are difficult to use and incur high run-time and memory overheads.

	Basic Algorithm	Scheduling	Barrier	Object Copying	Incremental Root Scanning	Discussed in Section
Baker	Copying	Work-based	Read	Incremental	No	3.2.1
Brooks	Copying	Work-based	Write	Incremental	No	3.2.2
Treadmill	Mark-sweep	Work-based	Read	Non-copying	No	3.2.3
Yuasa	Mark-sweep	Work-based	Write	Non-copying	No	3.2.4
Siebert	Mark-sweep	Work-based	Write	Non-copying	Single root	3.2.5
Blelloch and Cheng	Copying	Work-based	Write	Atomic	No	3.2.6
Ritzau	Reference counting	Work-based	—	Non-copying	—	3.2.7
Henriksson	Copying	Time-based[a]	Write	Atomic	Yes	3.4.1
Metronome	Mark-compact	Time-based	Read	Atomic	No	3.4.2
Kim et al	Copying	Time-based	Write	Unclear	Yes	3.4.3
Chang	Hybrid[b]	Time-based	Write	Non-copying	Yes	3.4.4
Schoeberl	Copying	Time-based	Write	Incremental [c]	Unclear	3.4.5

Table 3.1: Comparison of real-time garbage collection algorithms.

[a] The collector is time-based, but low priority threads are taxed with collection work.

[b] Reference counting and mark-sweep.

[c] Uses hardware to incrementally copy objects in the background.

CHAPTER 4

HARDWARE GARBAGE COLLECTION

Garbage collection has long been a source of overheads. Previous studies found that software collectors account for up to 40% of a program's run-time [43, 67]. This occurs because software garbage collectors do not run efficiently on conventional computer architectures. Collectors rely on simple memory operations that are better suited to dedicated hardware. For example, researchers observed that tracing is up to 9 times faster when implemented using dedicated hardware [104]. As a result, hardware optimizations for garbage collection have been studied for decades.

Early hardware garbage collectors were motivated by the high overheads of managed languages like LISP [117, 118] and Smalltalk [179, 187]. For example, Smalltalk programs were found to run 5 to 20 times slower than equivalent C programs [188]. Thus, special-purpose computers that relied on hardware garbage collection were built in the 1980s. However, these collectors were not intended for real-time systems. In addition, the special-purpose machines were not commercially successful because the applications of their architectures were limited and their development cost was too high [145]. In this chapter we explain and compare more recent hardware garbage collection proposals from the literature. We classify the proposals in two groups, *hardware-assisted* and *hardware-implemented*, according to the extent that the collection algorithm is realized in hardware.

4.1 Hardware-Assisted Garbage Collection

Hardware-assisted garbage collection optimizes only part of the algorithm using dedicated hardware. For example, Maas et al implement a hardware unit to accelerate tracing [104]. The specific hardware-assistance varies substantially across proposals, but in general, the idea is to accelerate the collection algorithm without compromising the machine's general-purpose functions or incurring high hardware overheads. The disadvantage is that optimization opportunities using

hardware are lost because most of the collector is implemented in software. In the remainder of this section, we discuss proposals for hardware-assisted garbage collection from the literature.

4.1.1 Pauseless

Azul Systems developed the Pauseless garbage collector for their enterprise multi-core system [52]. Pauseless is both parallel and concurrent; one or more cores run the collector while other cores execute the user's program. As a result, Azul's system is ideal for large enterprise applications on virtual machines, such as Java. But the system is only soft real-time and it does not appear to support weakly typed languages, such as C.

Pauseless implements a collector based on mark-compact mostly in software. A collection cycle has three stages. First, the memory is traced and live objects are marked. Marking is followed by a relocation stage that copies marked objects from sparsely populated pages to other pages; the physical storage for the sparse pages is reclaimed. Virtual memory protection is used to ensure that the user's program does not access pages currently being compacted. But hardware support is required to implement virtual memory efficiently and such features are rarely available in small embedded processors. A third stage, called *remap*, updates pointers to reference the new location of copied objects.

Azul's system uses hardware to accelerate Pauseless's read barrier. There is a barrier instruction that generates fast garbage collection traps, but the read barrier itself is implemented in software. During marking, the read barrier marks pointers being loaded into the roots. Loads during remap are also trapped to update pointers to old addresses before these are loaded into the roots. Barrier instructions generating a trap effectively pause the program while the barrier completes. Traps occur more frequently and take longer to perform at the start of the collection cycle because most unmarked pointers are discovered early, so the program may not progress its work much during some time intervals. This clustering effect of read barrier traps is undesirable in real-time systems because it is difficult to guarantee that a fair share of the processor will always be available for the program.

Azul's system uses *checkpoints*; points where the collector cannot proceed until all threads perform some action. The checkpoints require program threads to pause and run garbage collection operations. For example, a checkpoint at the start of the collection cycle enables marking the root set. But this can take a potentially long time as the root set includes the registers, stacks and globals. Another checkpoint at the start of the relocation stage is used to update all root references to live objects in the sparse pages that will be compacted. Therefore, Pauseless introduces program pauses that account for up to 16% of the time over a 2 second interval.

4.1.2 Joao et al's Hybrid Garbage Collector

Joao et al proposed a hybrid garbage collector for server machines [86]. Through experimentation, they observed that software collectors account for 15-55% of run-time overheads. These occur

because the user's program is often paused waiting for the collector to reclaim memory, so substantially increasing the memory size reduces the overheads. Joao et al's approach is conceptually similar to Chang's collector described in Section 3.4.4. They rely on reference counting to reclaim and repurpose most of the memory as soon as it becomes garbage while a backup tracing collector is used to identify garbage data structures forming cycles.

Reference counting has high run-time overheads due to the need to constantly update the counts. Therefore, Joao et al include hardware that coalesces multiple reference count updates while an object is cached. The counts are only updated in memory when an object is evicted from the caches; as a result, the performance cost is mitigated. Objects found to be dead using reference counting are added to a free list that is used to quickly satisfy allocations without obtaining further space from the memory system. But embedded devices do not always have a cache. Also, Joao et al's scheme has substantial cache memory overheads: 66.5 KB for a 64-bit system with 64 KB L1 D-cache and 4 MB L2 cache. These overheads are needed to store the collector's metadata, like the reference counts, in the cache.

The majority of the collector is implemented in software except the reference counting updates. So the collector introduces pauses for operations such as root scanning. The duration and frequency of these pauses vary depending on the software collector; the reference counting hardware imposes few restrictions in this regard. However, the user's program must execute special instructions instead of normal loads and stores to let the hardware know when pointers are copied or overwritten. This makes it difficult to use Joao et al's collector with weakly typed programming languages. The need for special instructions also makes it difficult to efficiently implement basic functions, such as copying a block of memory, because the software must be aware of the type of the words copied and overwritten. Finally, it is difficult to analyze the real-time properties of hybrid collectors, as discussed in Section 3.4.4, as there are no techniques to automatically estimate the amount of space taken up by objects forming cycles.

4.1.3 Maas et al's Mark-Sweep Accelerator

Maas et al proposed a hardware accelerator to mitigate the performance overheads of tracing garbage collectors in servers [104]. They observe that mark-sweep collectors spend 75% of time in the marking stage while only 15% performing read and write barriers. Therefore, Maas et al designed a hardware component that sits alongside the memory controller and performs the mark and sweep stages.

The accelerator has two main units: traversal and reclamation. The traversal unit implements the regular tracing algorithm in a pipelined fashion. Pointers and data are segregated into partitions within objects to facilitate marking, but this also makes Maas et al's collector unsuitable for weakly typed languages. The reclamation unit consists of a set of *block sweepers* that perform the sweep operation in parallel across multiple memory blocks. The collector does not compact memory, so it is vulnerable to fragmentation.

Maas et al only evaluate a basic implementation of their hardware accelerator alongside a RISC V processor running Linux. Compared to a software garbage collector, their accelerator performs mark and sweep 4.2 and 1.9 times faster respectively. However, the implementation is only stop-the-world and does not account for the challenges of concurrent collectors that cause considerable overheads. This is clearly unsuitable for real-time systems although Maas et al provide ideas on how they expect the collector to work concurrently. In addition, the collector is designed to operate alongside caches, virtual memory and complex memory controllers that are not available in small embedded processors.

4.1.4 Schoeberl and Puffitsch's Object Copying Accelerator

Schoeberl and Puffitsch use hardware to accelerate object copying and memory accesses via handles [150] in the Java Optimized Processor (JOP) [146]. Their accelerator operates in a Direct Memory Access (DMA) fashion and is designed for Schoeberl's real-time collector (see Section 3.4.5). The software initiates the copy operation by providing source and destination addresses along with the object's size, then the accelerator performs the copy in the background independently from the processor. Schoeberl and Puffitsch's design also reduces pause times due to read and write barriers. A memory access from the processor to an object currently being copied interrupts the accelerator. The memory access is blocked for 12 clock cycles before the hardware resolves it. The delay occurs because copying can only be interrupted at word boundaries, so the accelerator effectively steals memory cycles from the processor. Once interrupted by the user's program, the software must manually restart the copy operation which causes further delays and pauses.

The software collector must poll the accelerator to check when a copy operation terminates. Normally, data is copied at a rate of 5 clock cycles per word after the software starts the copy operation for an object. But Schoeberl and Puffitsch implemented and evaluated a simplified version of their accelerator that must be triggered for every individual word to copy. The accelerator must be triggered so frequently that it only operates while the software collector is running. So most of the performance benefits are eliminated because the accelerator cannot run independently in the background. In addition, copies are significantly slower at a rate of 27 clock cycles per word.

4.2 Hardware-Implemented Garbage collection

Garbage collectors are hardware-implemented if most of the algorithm is implemented using dedicated hardware. For example, Gruian and Salcic proposed a mark-compact collector fully implemented in hardware [74]; no part of the algorithm is realized in software. Hardware-implemented collectors enable performance and memory optimization opportunities that are difficult to match using software. In addition, the close integration between processor and garbage

collector in hardware facilitates eliminating pauses and overheads due to coordination between the two components. However, there are a number of perceived disadvantages [86]:

- Hardware-implemented collectors may limit the machine's general-purpose capabilities as it occurred with early LISP and Smalltalk computers. We consider that this perception arises because hardware-implemented collectors have traditionally been used alongside esoteric computer architectures that are often too specialized, such as object-oriented [111] or high-level language computer architectures [117, 118, 179, 187]. However, there is little research attempting to use hardware-implemented collectors with conventional RISC processors and no evidence that this is inviable.

- The cost of developing and verifying hardware-implemented collectors can be high because they incur major changes to the processor or memory architecture [86], but this is not always true. Hardware-implemented collectors can be realized as self-contained, add-on components that are connected to the processor via standard interfaces [124, 145, 165]. These garbage collectors do not require changing existing processor architectures or microarchitectures. Also, hardware-implemented collectors are not necessarily intended for large-scale systems, such as servers. They can instead be designed for smaller embedded systems that are less costly to develop and verify.

- Programs have widely different behaviors, so the memory management algorithm is expected to adjust to the software's needs. Hardware-implemented collectors are considered too inflexible to adjust to these changing requirements. This may be accurate in the context of large-scale systems. However, embedded systems have significantly more limited hardware and simpler memory hierarchies that do not permit as wide of a memory management design space compared to large-scale systems. In addition, hardware-implemented collectors achieve performance that is difficult to match with software.

We discuss hardware-implemented collectors from the literature in the remainder of this section.

4.2.1 The Garbage Collected Memory Module

Nilsen and Schmidt's Garbage Collected Memory Module (GCMM) is one of the first real-time hardware collectors [124, 145]. The module is a self-contained, add-on memory component that can be connected to the processor using standard interfaces. GCMM consists of a private microprocessor that runs a copying collector in the background alongside the main processor executing the user's program. The collector's semispaces are stored in separate memory banks within GCMM and memory cycles are allocated to the collector or the program by an arbitrer. This arrangement has prohibitive hardware costs for embedded systems due to the need for a dedicated processor for the garbage collector.

GCMM introduces pauses during program execution to obtain the roots and type information from the main processor. These pauses last up to 500 clock cycles in the worst case. In addition, read barriers take up to 6 memory cycles to complete. In general, a system with GCMM is slower than a system without it by 0-30%. These overheads are caused by cache flushing when GCMM copies objects. Another factor that contributes to the delays is that GCMM dynamically allocates function call stack frames which greatly increases the demand for free memory, especially on recursive programs.

4.2.2 Active Memory Processor

The Active Memory Processor (AMP) is a self-contained, add-on memory component similar to GCMM [165]. AMP implements a reference counting and mark-sweep collector using a private microprocessor and bitmaps. Most garbage memory is normally reclaimed using reference counting while the mark-sweep collector is run infrequently to reclaim data structures forming cycles. Therefore, AMP reduces memory requirements by 77% as most dead objects are reclaimed without delay by the reference counting collector.

An AMP module manages 4 MB of memory on behalf of the program. Multiple modules can be grouped to form larger systems. External fragmentation occurs because live memory is not compacted. AMP modules receive commands from the main processor to allocate memory, but it is not clear how other operations, such as read/write barriers and root marking, are performed. Collection operations and allocations are implemented efficiently using bitmaps. Each memory block is associated with a set of bits that are fed into and-gate and or-gate trees to compute information such as where a block of size n can be allocated. The bitmaps are too large to implement using dedicated hardware, so instead they are organized using software and cached within the AMP for fast access. However, the cache makes it difficult to analyze the system's real-time properties and a dedicated microprocessor for garbage collection incurs high overheads for embedded devices.

4.2.3 Meyer's Copying Garbage Collector

Meyer proposed an on-chip garbage collection coprocessor that uses semispace copying [112]. The collector is tightly coupled with the processor to reduce the coordination overheads between the two components. Thus, Meyer's collector pauses the processor for 300 clock cycles when marking the roots (instead of 500 with Nielsen and Schmidt's GCMM) and read barriers incur overheads of 50-100 clock cycles. In further work, Meyer moves the hardware implementation of the read barrier from the coprocessor to the processor and reduces its overheads to 5-50 cycles [113]. Meyer also estimates that his collector has memory overheads in a 3-5 factor to achieve real-time behavior, but no formal real-time analysis is presented to corroborate this.

The collector is designed for a novel object-oriented architecture previously presented by the same author [111]. Object headers and the organization of data and pointers within objects are

both specified by the architecture. This simplifies collection operations, such as distinguishing pointers from data, but makes the system less flexible and complicates supporting weakly typed languages which are widely used in embedded systems.

Meyer implements the collector as a microprogram. But compared to using dedicated hardware state machines, microprogrammed devices are often slower as they must load microcode from memory. The collector also uses locks to prevent the processor from accessing memory locations that are loaded in the collector's registers. These locks introduce pauses that are undesirable in real-time systems, even if the duration of the pause is bound. Finally, the collector uses a Baker-style read barrier that marks pointers being loaded into the roots. However, the barrier is triggered more frequently at the beginning of a collection cycle, so during this time the program cannot progress its work much. This problem is similar to the issue with the Pauseless collector (see Section 4.1.1) which makes it difficult to schedule real-time programs on these systems.

4.2.4 Stanchina and Meyer's Mark-Compact Garbage Collector

Stanchina and Meyer modified Meyer's system (see Section 4.2.3) to use mark-compact instead of copying [167]. The new system relies on temporary handles to redirect memory accesses while compacting, so loads and stores require two memory operations in the worst-case to be completed. The collector is still implemented as a microprogrammed hardware coprocessor, but the change reduces memory overheads by factors of 3-6.

A collection cycle has three stages. Reachable objects are marked during the marking stage and pointers are replaced by handle references. Objects are copied during the compact stage as usual. *Cleanup* is the third stage of the collection cycle during which handle references are replaced by object addresses. However, it is hard to estimate the number of references that are updated during cleanup, so the collector's real-time behavior is more difficult to model. In addition, the collector pauses the program twice during a mark-compact cycle, instead of once in the copying collector, as the roots have to be scanned during marking and cleanup.

4.2.5 Gruian and Salcic's Mark-Compact Garbage Collector

Gruian and Salcic proposed a concurrent mark-compact garbage collector implemented in hardware [74]. The collector is deeply integrated with the Java Optimized Processor (JOP) [146]. It relies on handles to eliminate the need to update the addresses of relocated objects. Each handle is associated with a location in memory that contains the object's mark flag, its address in memory and a pointer to a data structure with type information. So loads and stores from the processor require two memory operations; first the object's address is loaded from the handle and then the object is accessed.

There are high memory overheads for embedded systems due to object metadata. Each allocated object consumes two words of handle space and two words of header. The type information enables the collector to exactly distinguish pointers from integers in heap memory, but this

strategy cannot be used to garbage collect weakly typed languages, like C, where programs can cast arbitrary pointers to integers and vice versa. In addition, Gruian and Salcic's collector marks the function call stack conservatively as type information is not available in this case. Therefore, the exact set of live objects cannot be identified statically, so it is difficult to analyze real-time programs as discussed in Chapter 2.

The program occasionally communicates with the collector by executing commands. For example, the root set must be provided via a command to start a garbage collection cycle. So the user's program must regularly pause to perform these operations. Pauses also occur when executing read and write barriers. The program must acquire a lock on an object before performing a memory access in case that object is being compacted. Gruian and Salcic mitigate these pauses by interrupting the collector when it holds a lock on an object that the processor is attempting to access. Once interrupted, the collector must restart the copy from the beginning of the object. Thus, the collector is prevented from progressing if the processor repeatedly interrupts copying. This makes it unclear if Gruian and Salcic's collector is suitable for use in real-time systems and the locks still incur 2-7 clock cycles of overhead per memory access.

4.2.6 Garbage Collection for Reconfigurable Hardware

Developers are increasingly using reconfigurable hardware, in the form of Field-Programmable Gate Arrays (FPGA), to achieve greater performance. These devices are traditionally programmed with hardware description languages (Verilog and VHDL) which are difficult to use. So recent research has focused on enabling high-level languages, like C# and Java, to program FPGAs [26, 71]. But dynamic memory management is seldom used in FPGAs regardless of programming language, so Bacon, Cheng and Shukla proposed a hardware garbage collector fully implemented in reconfigurable logic for this specific use case [29].

The garbage collector implements a mark-sweep snapshot-at-the-beginning algorithm similar to Yuasa's. The heap memory is split into a set of *miniheaps*. Each miniheap stores objects of the same type and is implemented using one or more RAM components from Xilinx FPGAs know as Block RAM (BRAM) [192]. The collector is exact since each object field is stored in a different BRAM and only the fields containing pointers are scanned during marking. In addition, the BRAMs are truly dual-ported such that up to two memory accesses (read or write) can be performed in the same clock cycle. The collector accesses memory using a dedicated port while the application uses the other port, so both can operate simultaneously in every clock cycle. As a result, the collector never pauses as long as the allocation rate does not exceed the collection rate.

Bacon, Cheng and Shukla show that the collector is hard real-time. However, the number of miniheaps and the object types must be known statically before the hardware is synthesized. The size of object fields does not always map well to BRAMs which are of fixed size, so there can be fragmentation. Also, dual-ported memories are uncommon in embedded systems since they incur high overheads as will be discussed in Chapter 9. In summary, Bacon, Cheng and Shukla's

garbage collector is suitable for reconfigurable hardware applications, but it is difficult to use in general-purpose embedded systems due to its limitations and reliance on FPGA features.

4.3 Summary

Hardware has been used in the pursuit to optimize the performance of garbage collectors. Previous proposals can be broadly classified as either hardware-assisted or hardware-implemented depending on whether a small part or the majority of the collector is in hardware respectively. Hardware-assisted collectors are often flexible as the majority of the algorithm is in software and can be adapted depending on the application's needs. In contrast, hardware-implemented collectors offer optimization opportunities using hardware that are simply not available to software collectors. However, these collectors may limit the machine's general-purpose capabilities. We consider that this perception arises because hardware-implemented collectors have traditionally been used alongside specialized or esoteric computer architectures instead of RISC machines.

The main features of the hardware-assisted and hardware-implemented collectors surveyed in this chapter are summarized in Table 4.1. The majority of the proposals in the literature are intended for large computer systems that have different needs and aims compared to small embedded devices. Therefore, the collectors rely on features that are rarely available in the embedded context, such as deep cache hierarchies and virtual memory. This also explains why only one of the hardware collectors, i.e. Schoeberl and Puffitsch's, is intended for hard real-time systems and has been incorporated in a formal timing analysis; the other collectors can only be considered soft real-time at best.

The majority of hardware collectors surveyed use rigid techniques to distinguish pointers from data words. These cannot be used with weakly typed programming languages, such as C and C++. Our main motivation is also to enable the use of garbage collected languages efficiently. But we focus on embedded systems where the vast majority of existing software is written in C or C++. Therefore, hardware garbage collectors must enable compatibility with these older technologies to at least some extent. In addition, garbage collectors, whether in hardware or software, must be integrated into a complete processing system with I/O ports, interrupts, exceptions, etc. However, these practical considerations are rarely discussed in the hardware garbage collection literature.

Finally, most of the collectors reviewed introduce pauses despite using hardware. The pauses in hardware-assisted collectors occur because part of the collector is still implemented in software, so the user's program must be blocked to run collection operations or to interface with the hardware. In contrast, the pauses in hardware-implemented collectors are related to root marking and read/write barriers. They are due to the integration between collector and processor that often relies on locks or arbitration mechanisms which are inefficient. As a result, existing hardware garbage collectors incur performance overheads and are difficult to use in real-time systems.

	Basic Algorithm	**Acceleration Type**	**Barrier**	**Target System**	**Discussed in Section**
Pauseless	Mark-compact	Assisted (barrier)	Hybrid[a]	Server	4.1.1
Joao et al	Hybrid[b]	Assisted (count update)	Software	Server	4.1.2
Maas et al	Mark-sweep	Assisted (marking and sweeping)	Stop-the-world	Server	4.1.3
Schoeberl and Puffitsch	Copying	Assisted (object copying)	Software	Embedded	4.1.4
Nilsen and Schmidt	Copying	Implemented	Hardware	Server	4.2.1
Active Memory Processor	Hybrid[c]	Implemented	Unclear	Unclear	4.2.2
Meyer	Copying	Implemented	Hardware	Embedded	4.2.3
Stanchina and Meyer	Mark-compact	Implemented	Hardware	Embedded	4.2.4
Gruian and Salcic	Mark-compact	Implemented	Hardware	Embedded	4.2.5
Bacon, Cheng and Shukla	Mark-sweep	Implemented	Hardware	FPGA	4.2.6

Table 4.1: Comparison of hardware garbage collection algorithms.

[a] Read barrier is accelerated using an ISA instruction, but the barrier functionality itself is implemented in software.
[b] Reference counting and tracing.
[c] Reference counting and mark-sweep.

Part II

Integrated Hardware Garbage Collection

CHAPTER 5

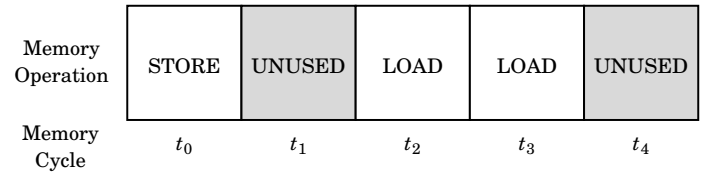
DESIGNING AN INTEGRATED HARDWARE GARBAGE COLLECTOR

Garbage collection has three limitations that must be overcome before it can be used in real-time embedded systems. First, the collector must have low run-time overheads. Second, it must have low memory overheads as embedded devices are often resource-constrained. And third, long and unpredictable pauses must be avoided to show that the collector allows the system to meet real-time requirements. Existing garbage collectors address one or two of these limitations at the expense of the others.

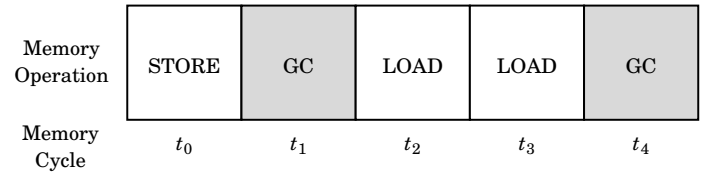
Hardware garbage collectors excel at reducing performance and memory overheads. These collectors have accelerators that run key operations more efficiently than software, such as read and write barriers. The collection hardware is often tightly integrated with the processor to mitigate coordination overheads between the two components. In addition, collection operations are performed in the background by dedicated hardware, so the processor is freed up to run the user's program. However, existing hardware collectors are not hard real-time and their timing properties are difficult to analyze formally.

Some specialized software garbage collectors reviewed in Chapter 3 are hard real-time. These algorithms are carefully designed so that the run-time of operations, such as loads, stores and allocations, are bound by a small constant. In addition, the collectors are supplemented by a formal real-time analysis to ensure that a share of the processor's time is reserved for the program. So the system's timing deadlines are always met, even in the worst-case, because unpredictable pauses are eliminated. But these hard real-time garbage collectors implemented in software incur high performance and memory overheads, are difficult to use and have undesirable side effects like increasing event handling latencies.

In this chapter, we present the high-level design of a garbage collector suitable for embedded systems. Concrete implementation details of this design are discussed later in Chapter 9. The



(a) Some memory cycles are unused in processors without the IHGC.



(b) Processors with the IHGC perform garbage collection whenever the memory is not used for instruction execution.

Figure 5.1: In conventional systems without the IHGC, many memory cycles remain *UNUSED* as the processor does not need them for instruction execution (*LOAD* and *STORE*). The IHGC takes advantage of these *UNUSED* memory cycles, that would otherwise be wasted, to perform garbage collection operations (*GC*) in the background independently from the processor. We call this *interleaving*.

collector was originally proposed by David May [108] and later adapted by Ed Nutting [127]. We further refine the design in this thesis and discuss the most important design choices. We also consider how techniques from hard real-time and hardware garbage collection research can be combined to strike a balance between performance, memory and pause requirements.

5.1 System Overview

Embedded processors typically use a fraction of all available memory cycles to fetch instructions and execute loads or stores. But as we will discuss in Chapter 7, about 20-30% of memory cycles remain unused. We present an Integrated Hardware Garbage Collector (IHGC) for hard real-time embedded systems that takes advantage of these spare memory cycles, that would otherwise be wasted, to run collection operations, which are mostly dependent on accessing memory. Thus, our design automatically *interleaves* memory accesses from the processor with collection operations, as shown in Figure 5.1. Garbage collection operations are performed only when the processor does not use the memory for instruction execution, so the IHGC does not steal memory cycles from the processor or incurs performance penalties for the user's program.

The IHGC implements a tracing algorithm fully in the hardware as a small state. It runs in the background independently from the processor although both components are tightly coupled as shown in Figure 5.2. The IHGC has read-only access to the register file to obtain the root pointers for marking. The processor has access to the collector's state to efficiently perform

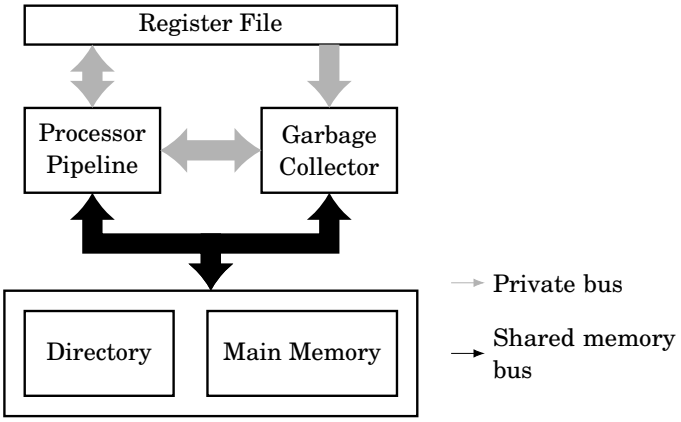


Figure 5.2: Overview of an IHGC system.

memory accesses in the hardware in a similar fashion to Meyer's proposal [113]. The IHGC relies on the use of address indirection through handles. A special memory that we call a *directory* stores metadata for each handle, while the main memory holds the program's data. The collector operates concurrently with the processor and access to both memories is provided via a shared memory bus. This ensures that the user's program is not paused by the collector when fetching instructions and executing loads or stores. Pauses only occur when the allocation rate exceeds the collection rate (exhausting all the free memory), but even these pauses can be eliminated by using static real-time analysis methods, which we will discuss in Chapter 6.

5.2 Pointer and Data Types

The IHGC uses *exact* garbage collection as it distinguishes pointers from data using type information. Every word in memory and the registers has a 1-bit *tag* that indicates whether it contains a pointer or data value. Without the type information, the collector would be *conservative* by assuming that a pointer is any word in memory whose value corresponds to the address of an object. As explained in Section 2.6, conservative collectors suffer from pointer aliasing problems, so they do not work well alongside compaction and their real-time behavior is difficult to analyze statically.

Programs running in a system equipped with the IHGC cannot directly manipulate tags. Pointers can only be obtained by allocating memory using the processor's `newm` instruction or copying an existing pointer. The IHGC type tags are realized as extra tag bits, so they incur low memory overheads as discussed in Section 2.6. This mechanism has four advantages:

1. The collector is fast and simple as there is no need to encode type information in type maps that are parsed either in hardware or with software assistance.

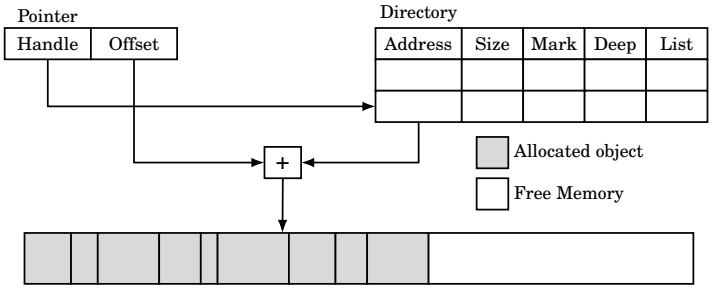


Figure 5.3: Address resolution for a memory access from the processor.

2. The IHGC is more flexible than collectors using pointer and data partitions within objects because the partition's encoding is not hard-coded in the hardware.

3. The IHGC maintains compatibility with standard integer types. The alternative is to steal a bit from the processor's natural word size, e.g. encode the type in bit 0 in a 32-bit machine, but the system foregoes compatibility with standard integer types because the range of integers that can be represented in a word is reduced.

4. The IHGC captures type information at run-time using the tag bits as the program allocates memory and links its data structures using pointers. Therefore, our collector is mostly compatible with weakly typed languages, such as C and C++. Some minor caveats are discussed in Chapter 8.

Type tags also prevent programs from constructing pointers to arbitrary memory locations. So tags guarantee the integrity of pointers and have important implications for security and reliability. For example, process isolation can be easily enforced because a process cannot access arbitrary objects that belong to another process if it does not hold valid pointers referencing them. In summary, the tags have benefits that are hard to achieve without hardware support.

5.3 Directory

In addition to having a tag, pointers in the IHGC are conceptually split into *handle* and *offset*. The handle uniquely identifies the referenced object and is an index into a fast directory memory. The offset is the byte index within the object that the pointer refers to. For each object, the directory has a metadata record that contains the object's *size* and its *base address* in physical memory together with maintenance information for the garbage collector (see Section 5.4). In the simplest case, accessing a word of memory requires the processor to load the object's base address from the directory using the pointer handle. Then, this is added to the pointer offset to resolve the physical address as shown in Figure 5.3. Finally, the memory access can proceed.

Programs running on the IHGC system never operate directly on physical memory addresses. The IHGC effectively enforces an indirection through the directory using handles. This is fully

implemented in hardware and is transparent from the programmer's point of view. There are three main benefits with this scheme:

1. The address indirection reduces garbage collection compaction work. Without this, collectors are required to update pointers that refer to relocated objects because their physical memory address has changed. By using indirection through handles, the collector only needs to update the object's physical memory address in the directory. This simplifies the collection algorithm, which in turn facilitates the worst-case run-time analysis needed for real-time systems.

2. The metadata in the directory can be leveraged to implement hardware safety checks in parallel with memory accesses. For example, out-of-bounds accesses are caught without run-time or code overheads by comparing the pointer offset with the object's size. Previous studies show that supporting similar checks in software for C programs usually incurs run-time overheads above 10% and code size overheads of 20-90% [97, 121, 139].

3. Storing the object metadata in a single location, such as the IHGC's directory, helps reduce duplication and memory overheads. This metadata is essential for safety operations like bounds checks. The alternative to maintaining it in a central location is to associate metadata with every pointer in memory. This kind of pointer is known as a *fat pointer* because the space requirements for each pointer increase by a factor of 2 to 4 depending on the implementation [73, 191]. As a result, *fat pointers* incur memory overheads of 8-200%, which is a significant cost for constrained embedded devices [97, 139].

The use of the handle indirection also has two perceived disadvantages. First, the directory incurs a fixed memory overhead to store the object metadata, but this is comparable or lower than the overhead introduced by explicit memory managers and other real-time garbage collectors as will be discussed in Section 7.4.4. And second, it is often considered that indirection through handles imposes high performance overheads because the directory must be read before accessing the memory. But implementing the indirection in hardware using the techniques explained in Chapter 9 eliminates any performance overhead. As a result, the advantages of the handle indirection outweigh the benefits for the IHGC.

5.4 Garbage Collector

The IHGC is a tracing mark-compact collector. Its operation relies on three metadata items per object that are stored in the directory: the *list* value that holds a pointer handle and is used to chain directory records into linked lists, the *mark* flag which signals that the object is live during a collection and a *deep* flag that is set if a pointer has been stored into the object. In addition, the first word in main memory of every allocated object is a *header* that contains the object's handle.

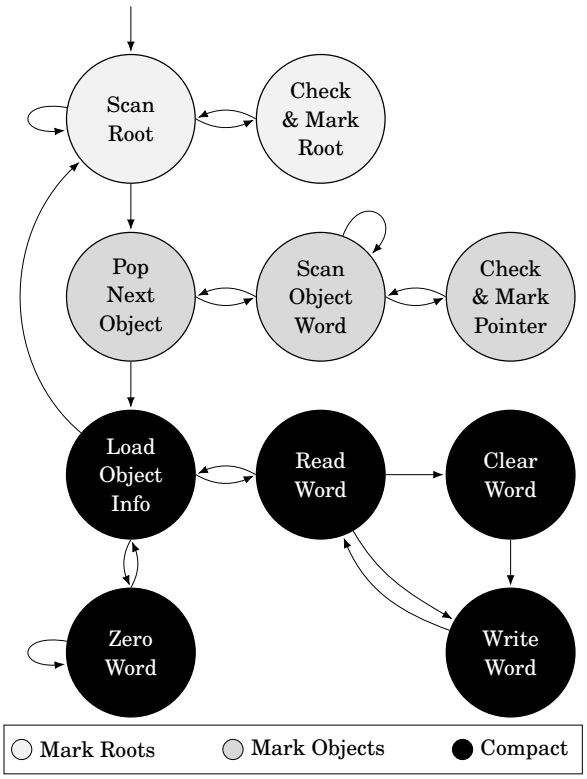


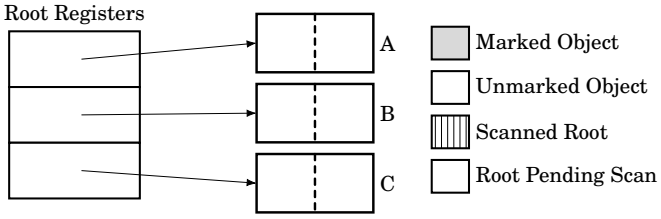
Figure 5.4: Simplified view of the IHGC state machine. Each state transition takes at most one memory cycle to complete. State transitions are performed concurrently with the execution of the user's program by using interleaving.

The garbage collector is implemented using the small hardware state machine shown in Figure 5.4. Each state transition is carefully designed so that it can be completed in a single memory cycle. This ensures that state transitions can be easily interleaved with memory accesses for instruction execution without pausing or stealing memory cycles from the processor as occurs with existing hardware collectors [150]. The complete specification for the IHGC's state machine is included in Appendix A.

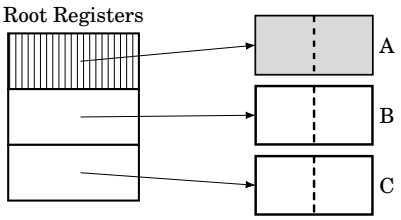
The remaining of this section describes in detail the stages in the IHGC's cycle.

5.4.1 Mark Roots

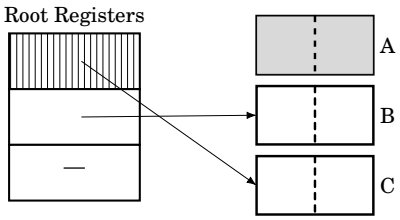
A collection cycle always starts by marking the roots. The `Scan Root` state inspects every element in the register file to extract pointers to live objects. For every root pointer found, the collector loads the handle's metadata from the directory and transitions to the `Check & Mark Root` state. If not already marked, the handle is marked as live by setting the mark flag in its directory record. Also, the collector inspects the handle's deep flag to determine whether the object may contain pointers. If the deep flag is set, then the handle is added to the *next* list so that it can



(a) Initially, none of the roots have been scanned by the collector and all objects are unmarked.



(b) The first root register is scanned and a pointer to *A* is found; the object is marked.



(c) The pointer to C is 'hidden' from the collector in an already scanned register.

Figure 5.5: Sequence of events causing the processor to 'hide' pointers from a concurrent garbage collector by moving them to already scanned roots. Objects referenced by the hidden pointers remain unmarked and are incorrectly reclaimed. The IHGC uses *shadow registers* to address this problem.

be scanned in search of unmarked pointers at a later stage. The deep flag greatly reduces the amount of work for the collector because objects that are guaranteed to not contain pointers, such as arrays of integers, will never be scanned during marking.

As marking proceeds, a *livesize* counter is kept of the total amount of space that will be used by objects that have been marked and that will be retained. This is used to optimize the compacting process. So at the end of the mark roots stage, the *livesize* records the aggregated size of all marked objects directly reachable from the roots and any objects marked as a side effect of load instructions (see Section 5.6). In addition, the *next* list contains the handles of marked objects labeled as deep.

The IHGC scans one root per memory cycle only and it operates concurrently with the processor. It is therefore possible that reachable objects remain unmarked at the end of the marking stage. This is because the processor might 'hide' pointers by moving them between the registers. For example, the system in Figure 5.5(a) has three roots and initially all objects are unmarked. The collector scans the first root and marks object *A* as shown in Figure 5.5(b). The user's program then moves the pointer to *C* to the first register and overwrites the third register before the collector finishes marking the roots as illustrated in Figure 5.5(c). The first root is not scanned again and object *C* remains unmarked even though it is reachable. To solve this problem, a copy of every register is made before marking starts. The copies are the roots that the IHGC scans and are never overwritten during mark roots. It is important that the root set is

small to ensure that all roots can be copied without pausing the user's program or substantially increasing hardware requirements. As a result, the IHGC's root set consists of the processor registers only. Other data structures, such as the function call stack and global variables, are treated as dynamically allocated objects.

The roots can be efficiently copied in hardware using *shadow registers*. The register file is equipped with two copies of every register: a main copy that the user's program reads and writes and a shadow. Before marking starts, the content of the main registers is written into their corresponding shadows in parallel. The shadow registers are read by the IHGC during the marking stage while the main registers are accessed by the user's program as normal. This reduces the need for coordination between processor and collector, so pauses when marking the roots are eliminated.

5.4.2 Mark Objects

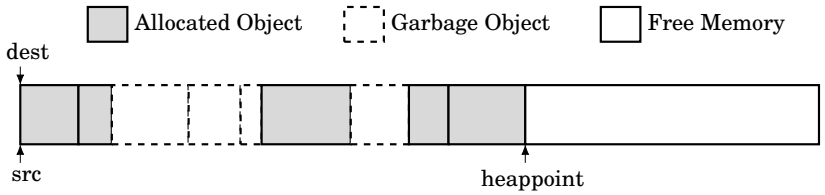
The collector starts the mark objects stage (shown in Figure 5.4) by transitioning to the `Pop Next Object` state after all the processor's shadow registers have been scanned. At this state, the collector transitions to `Scan Object Word` if the *next* list is not empty. During this transition, the handle at the front of the *next* list is popped and the address and size of its corresponding object are loaded from the directory. Marked objects whose handle is on the *next* list, i.e. deep objects, are scanned one word at a time in the `Scan Object Word` state. Each word from an object being scanned is loaded from memory; the pointers found are checked and marked at `Check & Mark Pointer` in the same fashion as `Check & Mark Root` described previously.

Marking completes when the *next* list is empty and all reachable objects are marked and scanned. So the collector begins the compact stage by transitioning to the `Load Object Info` state.

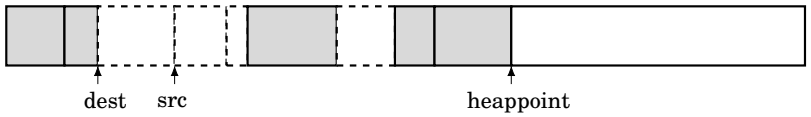
5.4.3 Compact

During the compact stage, the collector accesses all memory locations up to the *heappoint*; the highest allocated address in memory. Marked objects are retained and copied towards the beginning of the memory while garbage objects are reclaimed. Reclaimed memory locations are also zeroed before joining the free memory to ensure that they does not contain pointers when the memory is reallocated (see Section 5.5).

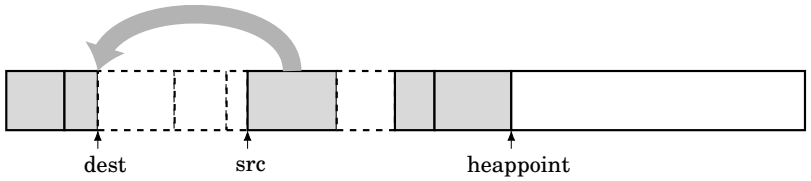
The header word at the beginning of each object in memory indicates the object's handle. The collector also uses two registers to copy objects as shown in Figure 5.6: *src* and *dest* are pointers to the object's old and new base addresses respectively. Initially, both registers are set to the lowest memory address; the header of the first object in memory. Each transition of the `Load Object Info` state uses the header to locate the object's metadata in the directory. At this point, the collector performs either of three operations:



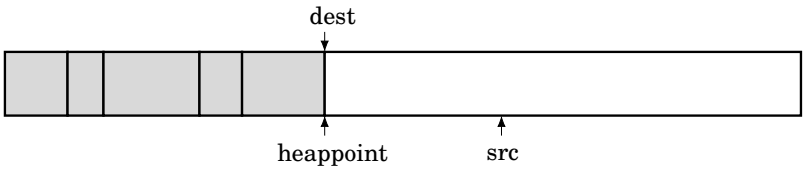
(a) Memory and IHGC pointers at the beginning of the compact stage.



(b) Marked objects that do not need to be relocated are skipped. Garbage objects are reclaimed and overwritten with marked objects or zeroed.



(c) Some marked objects are copied from their previous address at *src* to a new *dest* location.



(d) Allocated objects are compacted at one end of memory at the end of the collection cycle. The *heappoint* register is changed to reference the highest allocated address in memory.

Figure 5.6: Operation of the IHGC compact stage.

1. The object at *src* is marked and $src = dest$ as shown in Figure 5.6(a). The object is retained and its mark flag is cleared in the directory in preparation for the next collection cycle. The *src* and *dest* pointers are both increased by the object's size.

2. The object at *src* is unmarked, so it is reclaimed by adding its handle to the front of a *free* list of handles. The *src* pointer is increased by the garbage object's size, but *dest* remains unchanged as shown in Figure 5.6(b). Memory from reclaimed objects is either overwritten later with live objects or the collector immediately starts zeroing it by transitioning to `Zero Word`. The collector decides whether zeroing is required by comparing the object's address with the *livesize* as explained below.

3. The object at *src* is marked, but $src \neq dest$ as shown in Figure 5.6(c). The object is retained, its mark flag cleared and copying from *src* to *dest* starts immediately by transitioning to `Read Word`.

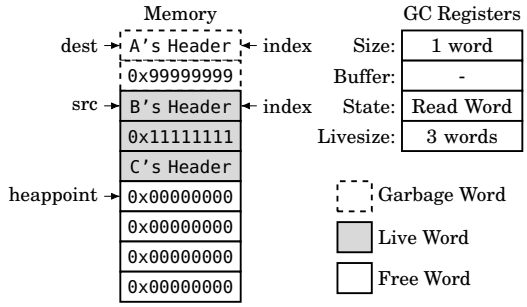
The compact stage terminates when *src* = *heappoint*. The *heappoint* is set to the *dest*, the highest allocated address in memory, as shown in Figure 5.6(d). The IHGC is now ready to start a new collection cycle.

A live object is copied from *src* to *dest* one word at a time and its old locations, i.e. at *src*, are zeroed if needed. Copying for an object starts at the `Read Word` state. Initially, the *index* register is set to the first word to copy and a *size* register indicates the object's size in words. For example, the system shown in Figure 5.7(a) has a garbage object A followed by two live objects B and C. The IHGC is preparing to overwrite A's locations with B's contents, so *index* is set to 0 and *size* is 1, i.e. A's size without including the header. At `Read Word`, the collector loads the next word to be copied, i.e at address *src* + *index*, into a *buffer* register and transitions to either of two states depending on whether zeroing is required:

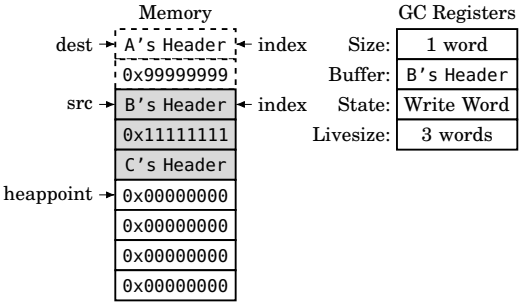
1. If *src* + *index* < *livesize*, the word's old location does not need to be zeroed because it will be overwritten later when other objects are compacted, so the collector transitions to `Write Word`. In the example, B's old header will be overwritten with C's contents, so the collector state is `Write Word` in Figure 5.7(b).

2. If *src* + *index* ≥ *livesize*, the word's old location will not be overwritten with live objects as the compact stage proceeds, so the collector transitions to `Clear Word` to zero the word at *src* + *index* before moving on to `Write Word` (see Figure 5.7(d)).

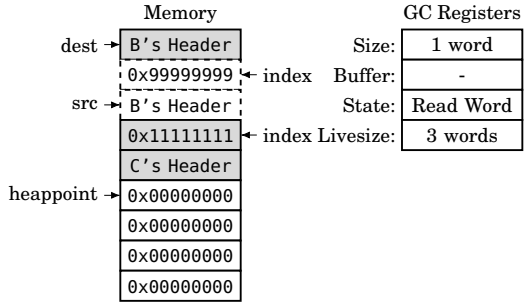
The collector transitions from `Write Word` to `Read Word`: it stores the word at *buffer* into the new memory location, i.e. *dest* + *index*, and increments *index* as shown in Figure 5.7(d). The object is fully copied when the state is `Read Word` and *index* = *size* + 1 as in Figure 5.7(f), so the collector transitions to `Load Object Info` to process the next object in memory.



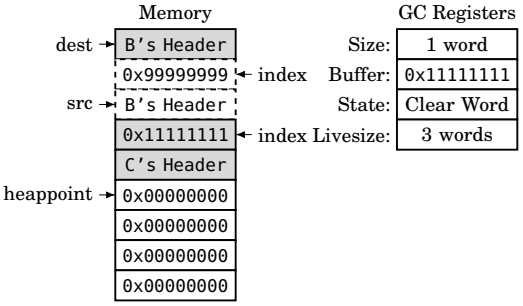
(a) Initially, *size* indicates *B*'s size and *index* is 0 while *src* and *dest* contain *B*'s old and new base addresses.



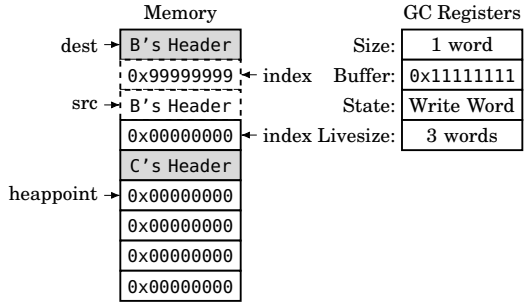
(b) The word at *src* + *index* is not zeroed, so load it into *buffer* and transition to `Write Word`.



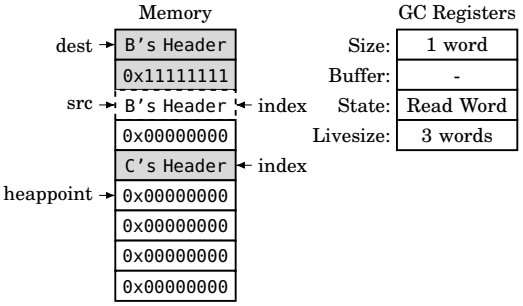
(c) The first word is copied and the *index* is incremented.



(d) The word at *src* + *index* needs zeroing, so load it into *buffer* and transition to `Clear Word`.

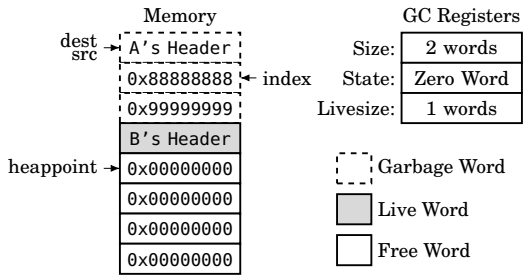


(e) The word at *src* + *index* is zeroed and the collector transitions to `Write Word`.

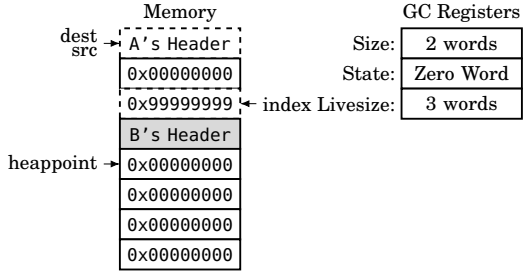


(f) The second word is copied and the *index* is incremented.

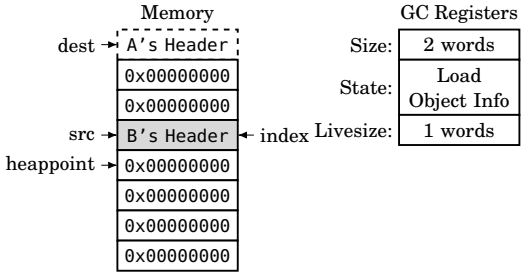
Figure 5.7: An example system with one garbage object *A* stored at the start of memory followed by two live objects *B* and *C*. During the compact stage, the IHGC overwrites *A* with *B*'s contents and zeroes the old locations from *B* that will not be overwritten with *C*'s contents.



(a) Initially, *size* indicates *A*'s size, *src* its base address and *index* is the index of the first word to zero.



(b) Each state transition from `Zero Word` zeroes the word of memory at *src* + *index* and increments *index*.



(c) The collector transitions to `Load Object Info` as no further locations within the former *A* need to be zeroed.

Figure 5.8: An example system with one garbage object *A* stored at the start of memory followed by a live object *B*. The collector zeroes the last two words of *A* using the `Zero Word` state as they are not overwritten with live objects.

It is possible that memory locations from garbage objects are not overwritten by live objects during the compact stage. The collector zeroes these locations by transitioning to the `Zero Word` state from `Load Object Info` when an unmarked object is found. Initially, *src* and *size* contain the base address and size respectively of the garbage object while *index* indicates the index of the first word that needs to be zeroed. For example, the system in Figure 5.8(a) shows a garbage object *A* stored at the start of memory followed by a marked object *B*. *A*'s first word does not need to be zeroed as it will be overwritten with *B*'s contents, so *index* = 1. Each subsequent state transition from `Zero Word` zeroes the word at *src* + *index* and increments *index* (see Figure 5.8(b)). No further locations within the garbage object need to be zeroed when *index* = *size* + 1, so the collector transitions back to `Load Object Info` as illustrated in Figure 5.8(c).

5.5 Memory Allocation

Programs allocate objects using the processor's `newm` instruction. Objects allocated during a collection cycle have their mark flags set immediately; otherwise, they may not be retained, for example, if the allocation occurs when the collector is at the compact stage. Memory allocations

are completed in one memory cycle if there is enough memory and handles to satisfy the request. Allocations are simple because free memory is always clustered at one end of the memory space. Therefore, the system only needs to pop a handle from the *free* list, increment the *heappoint* by the requested amount of storage space and initialize the object's header. Newly allocated objects are guaranteed to be initialized to zero as a result of compacting. In addition to preventing the leak of sensitive information, zeroing is essential to the IHGC's correct operation because it clears the pointer tags. Otherwise, newly allocated objects might contain pointers with handles that have already been reclaimed.

Allocations cause pauses when there is insufficient space or handles to satisfy a request. The user's program is paused until enough resources are reclaimed to fulfil the allocation. These pauses may last up to two collection cycles in the worst-case (see Section 6.1). However, we can ensure that even these pauses are completely eliminated in real-time programs by using the analysis presented in Chapter 6.

5.6 Marking On Load and Memory Access Redirection

The IHGC operates concurrently with the processor, so it must prevent reachable objects from remaining unmarked during the mark stage. The IHGC's design is conceptually similar to Baker's read barrier [32]: pointers loaded into the root registers are processed for marking. So references to unmarked objects are never written into already marked and scanned objects, i.e. shaded black. The IHGC implements this coordination mechanism fully in hardware within the processor's pipeline. Pointers read from memory when executing load instructions are immediately written back to the register file without stalling subsequent instructions. The pointer is also processed for marking within the pipeline, but this is performed in the background outside the load instruction's critical path. The IHGC's state machine is stopped for the duration of the load instruction and while marking is performed, so traps, locks or interrupts common in other hardware collectors are not required [52, 74, 112, 113, 150, 167]. As a result, the coordination between the IHGC and the processor while marking is efficient and never pauses the user's program.

During the compact stage, it is possible that the processor accesses an object that is currently being compacted. The memory access must be redirected to the correct location or register where the word is stored; otherwise, the contents of the object and the registers could become inconsistent. Logic in the pipeline addresses this problem by performing memory access redirections. To access a word of memory, the processor loads the object's metadata from the directory using the pointer handle. Simultaneously, the index to access within the object is computed by adding 1 to the pointer offset; this accounts for the header word at the beginning of the object. Then the processor uses the collector's state to find the correct location to access. There are four possible resolutions for the memory access redirection as illustrated in Figure 5.9:

1. The collector has not copied the word accessed yet ($offset + 1 > index$). So the location to

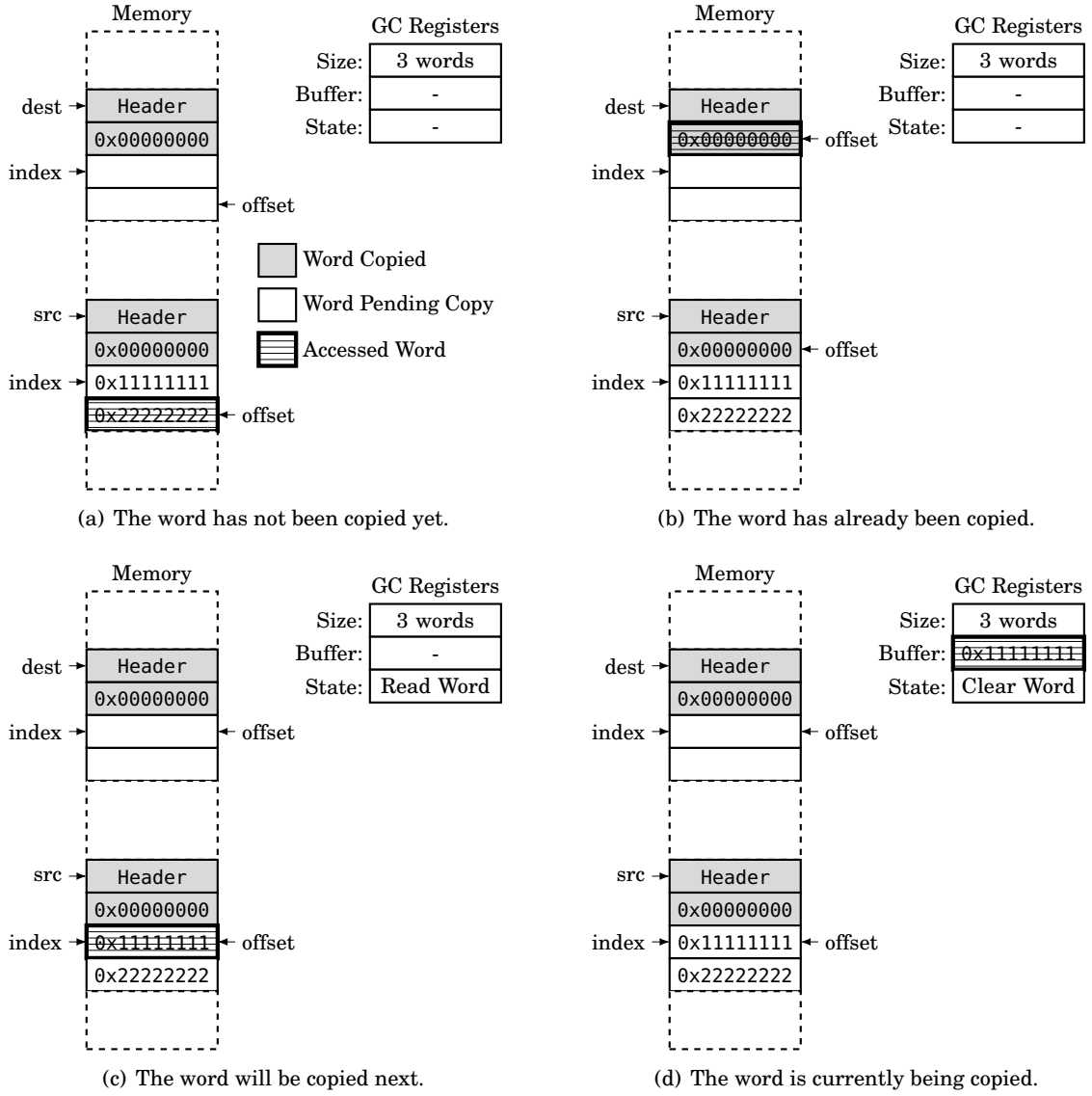


Figure 5.9: There are four possibilities when redirecting a memory access from the user's program to a word in an object being compacted. The *offset* is the pointer's byte offset to access within the object. *src*, *dest*, and *index* are the IHGC's registers used during the compact stage as described in Section 5.4.3.

access is *src* + *offset* + 1; the object's old address must be used as the base (see Figure 5.9(a)).

2. The collector has already copied the word accessed (*offset* + 1 < *index*). So the location to access is *dest* + *offset* + 1; the object's new address must be used as the base (see Figure 5.9(b)).

3. The word accessed will be copied next. This occurs when *offset* + 1 = *index* and the collector's state is neither `Clear Word` nor `Write Word` i.e. the copy word operation has not started. So the location to access is still *src* + *offset* + 1; the object's old address must be used as the base (see Figure 5.9(c)).

4. The word accessed is currently being copied (*offset* + 1 = *index*) and the collector's state is either `Clear Word` or `Write Word`. So the memory access must be performed on the collector's *buffer* register which temporarily holds the value being copied (see Figure 5.9(d)).

The clear benefits of the IHGC is that memory access redirection and marking on load are fully transparent from the program's point of view and do not incur any code size overheads. Additionally, implementing the mechanism in an embedded system requires adding little extra logic to the pipeline as discussed in Chapter 9. Pauses or performance penalties are not introduced, so real-time analysis with the IHGC is simplified. In contrast, existing collectors normally use traps, locks or other synchronization mechanisms to enforce an ordering on collection operations and the program's execution; these usually cause pipeline stalls and are described as read and write *barriers*.

5.7 Alternative IHGC Designs

Garbage collectors have an extremely wide design space, so we explored many other alternatives with various tradeoffs. We describe and consider the advantages of some of these designs for the sake of completeness and for future research.

5.7.1 The Collection Algorithm

The choice of collection algorithm has important implications in terms of overheads and real-time properties as discussed in Section 2.1.3. A clear alternative to mark-compact is reference counting. It has the advantage that objects can be reclaimed immediately after they are found dead, but it cannot reclaim data structures forming cycles. In addition, garbage objects may trigger a long chain of counter update operations. Chang used reference counting in combination with mark-sweep to address some of these problems in a real-time system [46–48]. However, his collector suffers from fragmentation and its real-time analysis requires difficult to get information about the objects forming cycles.

Mark-sweep is a simpler alternative to mark-compact which could substantially reduce the complexity of hardware implementations. For example, the IHGC's memory access redirection and directory would not be needed. But real-time programs do not tolerate problems like fragmentation. Therefore, objects must be allocated within fixed-size blocks as in the Treadmill [31] or Siebert's collector [159–164]. Unfortunately, these collectors place restrictions on the programmer and cause overheads.

Copying algorithms do not suffer from fragmentation and have been widely used in real-time systems. These collectors require a single pass and their worst-case run-time is proportional to the amount of live memory. In contrast, mark-compact collectors require at least two passes and their worst-case run-time is proportional to the memory size. Therefore, the run-time of copying collectors is easier to analyze compared to mark-compact. Unfortunately, copying collectors have higher memory overheads than mark-compact in the average-case, although not the worst-case as shown by Schoeberl [147]. Furthermore, read and write barriers in copying algorithms often require multiple memory accesses because loads and stores must be redirected between semispaces. This results in increased complexity and overheads making such collectors unattractive for hardware implementations. For these reasons, we have chosen mark-compact instead of reference counting, mark-sweep or copying.

5.7.2 Mark Roots

The mechanism to efficiently copy the roots in the IHGC can be difficult to realize in hardware as the number of registers increases. So we considered alternative root marking designs that do not require a copy. The simplest solution is to atomically mark the roots as is implemented in many existing garbage collectors [52, 74, 124, 145, 167, 195]. The obvious drawback is that the user's program must be paused for the full duration of the root marking operation. For this reason, previous collectors implement techniques to bound the duration of pauses when marking the roots, for example, by enforcing read or write barriers when accessing the function call stack [113]. But despite being short, the pauses can complicate the system's real-time analysis.

Siebert's solution for the root marking problem also eliminates the need to copy the root registers [159–164]. Siebert's collector has a single root pointer and every pointer in the registers must also exist in memory. Scanning the single root is simple in hardware and does not incur program pauses, but there are two main disadvantages with this approach. First, the user's program needs to cooperate with the collector to ensure that all references in the roots are correctly backed up in memory. The copies could be, at least partly, maintained by the compiler, but this still places tedious and error-prone memory management burdens on the programmer that we are hoping to eliminate using garbage collection. Second, there is a performance cost because the program must constantly ensure that the references are backed up in memory.

We considered marking pointers when the user's program moves them between registers during root marking. This eliminates the need to make a copy of the complete root set before mark-

ing because the program cannot 'hide' pointers from the collector as explained in Section 5.4.1. However, marking on register-to-register move instructions reduces the spare memory cycles for collection operations. This can delay the completion of a collection cycle and cause pauses if memory is not reclaimed sufficiently fast. Another problem occurs when integrating the IHGC with existing computer architectures. Instruction sets, such as ARMv7-M and RISC V, do not distinguish between pointers and data words. Therefore, arithmetic, logic and other instructions are used indiscriminately regardless of the operand's type. These instructions occasionally receive pointer operands, so they move pointers between the registers. This complicates real-time analysis as it is unclear which instructions, that do not normally perform memory operations, consume memory cycles as a result of marking. The memory cycles are no longer available to perform collection operations, so it is harder to estimate tight upper bounds on the collector's run-time.

Compared with the alternative root marking strategies, the IHGC's required copy does not incur any pauses. More importantly, the IHGC's root scan operation is incremental and requires minimum interaction with the processor, so the collector's real-time analysis is simplified. Also, most modern RISC architectures, such as ARMv7-M and RISC V, have a small register set that does not exceed 16 or 32 registers. So efficiently copying all registers using a strategy like shadow registers incurs minor hardware overheads. For these reasons, we decided against using an alternative root marking technique.

5.7.3 Marking On Store

The IHGC marks pointers when they are loaded into the registers to prevent reachable objects from remaining unmarked. The alternative approach is to use write insertion barriers such as Brooks'. Instead of marking on load, the collector marks pointers as they are stored into memory. This prevents references to unmarked objects from being written into already marked and scanned objects. Before ending the marking stage, the collector must also repeatedly scan the roots until no unmarked objects are discovered. In the worst-case, the collector scans the roots once for every reachable object, but this is extremely unlikely. Unfortunately, estimating a tighter upper bound necessary for real-time analysis is difficult.

Another strategy considered is Yuasa's write barrier that enforces a snapshot-at-the-beginning invariant. As described in Section 3.2.4, Yuasa's write barrier loads the location that is about to be overwritten during a store operation. If the loaded value is a pointer, it is processed for marking. However, this operation complicates the processor pipeline's hardware because a store actually requires two memory accesses. The write barrier also causes stalls when executing a continuous sequence of memory access instructions. This is because the run-time of stores would almost double, even when there are no dependencies between the instructions.

The IHGC's mark on load strategy can be implemented without causing pipeline stalls or pausing the user's program. It also simplifies real-time analysis which is one of the main

objectives of this thesis. Therefore, we decided against using the alternative strategies outlined in this section.

5.8 Summary

We presented an Integrated Hardware Garbage Collector (IHGC) that runs in the background reclaiming memory independently from the processor. The IHGC implements a mark-compact algorithm fully in hardware that operates whenever the processor is not accessing memory. But it never pauses the program as its operation is split at the granularity of a memory cycle. The IHGC uses extra tag bits to exactly distinguish pointers from data, so its real-time behavior can be formally analyzed. In addition, memory access indirection using handles simplifies the collector's operation and efficiently maintains the system's integrity using hardware within the processor. As a result, the IHGC incurs low performance and memory overheads and it can be shown to meet hard real-time requirements.

The main choices in the IHGC's design were explained in this chapter. Collection algorithms, such as reference counting and copying, were compared to the IHGC's mark-compact and discarded due to issues with hardware complexity, ease of real-time analysis or performance overheads. Alternative marking strategies were also analyzed and discarded due to their numerous drawbacks.

CHAPTER 6

HARD REAL-TIME ANALYSIS WITH THE IHGC

A novel Integrated Hardware Garbage Collector (IHGC) was presented in Chapter 5. However, it has not yet been shown how unexpected IHGC pauses can be prevented to ensure that hard real-time programs never miss their deadlines due to garbage collection. We address the issue in this chapter by proposing a static analysis technique that guarantees the complete absence of pauses. The analysis characterizes the user's program and derives the worst-case memory requirements to ensure that the system's hard real-time constraints are met.

Previous research has demonstrated how software garbage collectors can be scheduled concurrently with hard real-time programs [28, 47, 79, 92, 147]. But the IHGC is implemented in hardware and has features, such as interleaving, that cannot be accurately modeled using existing analysis frameworks. Therefore, we introduce a new static analysis technique to provide hard real-time guarantees using our hardware-implemented garbage collector.

6.1 Pauses in the IHGC

The IHGC never pauses the user's program when the processor executes load or store instructions. Pauses only occur if memory is allocated faster than it can be reclaimed. In this case, the collector cannot 'keep up' with the rate of allocation. Eventually, the system will reach an out-of-memory condition as there is not enough free space to satisfy an allocation request. When an out-of-memory condition arises, the execution of the `newm` instruction (and consequently the program) is paused. These pauses last until the collector reclaims enough memory to fulfil the allocation. In the worst-case, the program is paused for the duration of two collection cycles because if the last remaining pointer to an object is eliminated after it has been marked, then that garbage object survives the current collection cycle and is only reclaimed in the next cycle. These long pauses

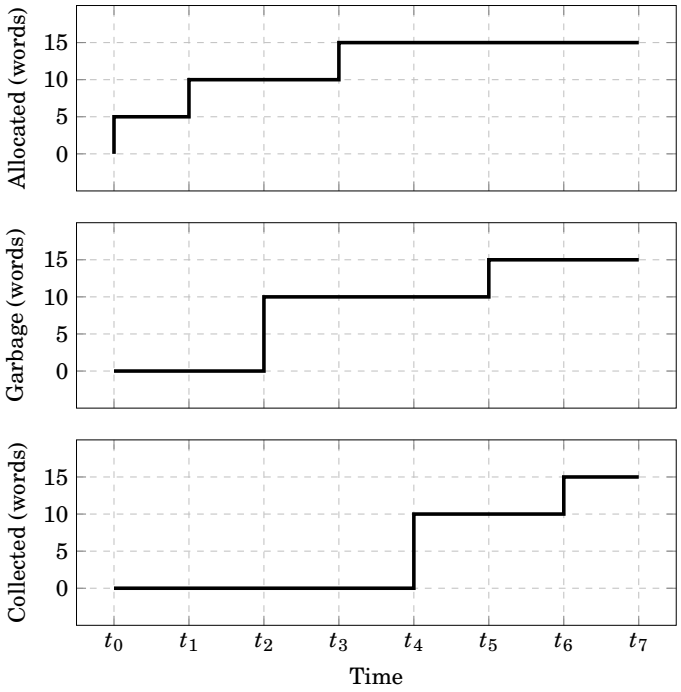


Figure 6.1: Timeline of memory allocated, garbage generated and memory reclaimed for a simple system using a mark-compact garbage collector.

during allocation make real-time analysis infeasible, so our aim is to guarantee that they do not happen unexpectedly, or better yet, that the pauses are completely avoided.

The pauses are caused by the operation of the IHGC's underlying mark-compact algorithm. The collector must first trace the full set of live objects before reclaiming and compacting. So memory from multiple garbage objects joins the free memory pool at the end of the collection cycle all together, as opposed to being reclaimed individually, immediately. For example, in the timeline of a system shown in Figure 6.1, two objects are allocated at times t_0 and t_1. Both objects are made garbage at t_2, but they are only reclaimed at t_4 when the collector completes a cycle. This implies that the memory collected is less than the garbage memory shortly after an object dies. The garbage memory only converges with that reclaimed as collection cycles are completed. In the worst-case, the convergence delay lasts up to two full collection cycles. This reclamation delay is illustrated in Figure 6.1 as the 10 words allocated by t_1 are only collected at t_4 even though the memory was garbage after t_2.

To avoid pauses, we must ensure that the system will not reach an out-of-memory condition before the memory collected converges with the garbage generated. We achieve this by provisioning the system with enough memory so that allocations can be fulfilled without pauses while the collector reclaims the next batch of garbage objects. For instance, the system in Figure 6.1 requires at least 15 words of memory to avoid pausing even though the program's live size is 10 words. This is because the allocation at t_3 happens before the garbage generated at t_2 is

reclaimed. Thus, our analysis problem is reduced to finding a safe worst-case memory size which guarantees that out-of-memory conditions will never occur.

6.2 Analysis Overview

The IHGC interleaves collection operations with memory accesses from the processor in order to take advantage of unused memory cycles. Interleaving ensures that the program is not paused if the system is provisioned with enough memory to satisfy allocations while garbage is reclaimed. So our analysis must determine the allocation rate along with the number of unused memory cycles available to the collector during the execution of a given real-time program. From this, we must then calculate the worst-case memory bound to guarantee that the collector has enough time to reclaim garbage while operating during the spare memory cycles only. Such a bound exists if the collection rate is greater than the allocation rate. Otherwise, the analysis fails to find a memory bound and unexpected pauses can occur.

Interleaving and our static analysis guarantee that collection pauses never occur. There are no collection operations to schedule in the software and the processor can instead be fully utilized to execute the user's program. This eliminates the need to consider the IHGC's operation as another software process to be scheduled, in contrast to most modern time-based collectors [28, 79, 92]. As a result, real-time analysis with the IHGC is simpler than with existing software collectors.

In the remainder of this chapter, we use our understanding of the problem and knowledge about the IHGC from Chapter 5 to derive the equations of our real-time analysis.

6.3 Worst-Case Memory Requirement

Our analysis only considers systems that are in *steady state*. That is, the program generates as much garbage as it allocates memory when measured over a long interval of time. The goal is to estimate the minimum memory size that prevents pauses when executing allocation instructions. There must be enough space to accommodate the live and garbage objects and satisfy new allocations during the time interval that the IHGC takes to reclaim garbage.

The variables used in our formulation are outlined in Table 6.1. The program has an amount of live memory $n + r$. n is the space scanned for marking, but never relocated while compacting. Objects that contribute to n are never reclaimed during the program's lifetime, such as global variables and code. The IHGC does not treat these objects differently, but we can ensure that they are never relocated by grouping them towards the beginning of memory. For example, the system can place the code object at address 0 at reset; assuming that the code object is always live, it will never be relocated if the IHGC copies live objects towards 0 while compacting. In contrast, r is the amount of live memory that can be relocated, such as temporary buffers. Also, the program allocates at most w words of memory during a collection cycle. This memory will be

Variable	Description
n	Memory scanned during marking, but never relocated.
r	Live memory that is relocated.
w	Memory allocated in the time interval that it takes to complete a collection cycle.
l	Live memory at the end of a collection cycle.
m	Amount of memory needed to eliminate program pauses.
t_{gc}	Time taken to complete a collection cycle.
t_f	Memory cycles available to the collector in an execution period.
a	Memory allocated during a collection cycle.

Table 6.1: Variables involved in our worst-case memory formulation.

considered live and retained because newly allocated objects have their mark flags set. So the maximum amount of live memory l at the end of a collection cycle i is

$$l = n + r + w \tag{6.1}$$

We expect that at most w words of memory will be allocated once again during the next collection cycle $i+1$. In addition, the memory that is garbage at the start of $i+1$ will only be collected when the cycle is completed. Therefore, the worst-case amount of memory m needed by the system to eliminate pauses during collection cycle $i+1$ is

$$m = l + w \tag{6.2}$$

Substituting Equation 6.1 into Equation 6.2 to remove l we obtain

$$m = n + r + 2w \tag{6.3}$$

The required memory m includes twice the space allocated during a collection cycle w. This is because the memory reclaimed only converges with the garbage generated after two collection cycles in the worst-case as explained in Section 6.1.

Our reasoning about w is similar to the one presented by Robertz and Henriksson [142]. We assume that the program allocates at most a words of memory during each execution of its periodic task. Also, the program does not access memory for t_f memory cycles in each period. So the worst-case w is given by the space a allocated in as many t_f intervals as necessary to complete a collection cycle of length t_{gc} or

$$w = a \left\lceil \frac{t_{gc}}{t_f} \right\rceil \tag{6.4}$$

Replacing the ceiling function in Equation 6.4 by a stronger condition and substituting for w in Equation 6.3 we get

$$m = n + r + 2a\left(\frac{t_{gc}}{t_f} + 1\right) \tag{6.5}$$

To resolve m, we need suitable estimates for t_{gc}, a and t_f. This is explained in Section 6.4 and Section 6.5.

6.4 Timing Model for the IHGC

We divide the IHGC's operation in four parts that are aggregated to estimate t_{gc} as follows

$$t_{gc} = t_{init} + t_{roots} + t_{mark} + t_{compact} \tag{6.6}$$

We individually analyze each component of t_{gc} in detail.

6.4.1 Initialization and Termination (t_{init})

Each garbage collection cycle has a constant-time overhead t_{init} associated with initialization and termination. During initialization, the IHGC's hardware state machine is set up to start a new collection cycle. When terminating, the IHGC checks failure conditions. For example, an exception is raised if the IHGC detects that there is not enough physical memory to satisfy a pending allocation request.

6.4.2 Mark Roots (t_{roots})

Compared to most collectors, root marking is a simple operation in the IHGC as the root set consists of the processor's registers only (including any instruction and stack pointer registers). Larger data structures, like global variables and the function call stack, are not considered as part of the roots and are instead marked and compacted like any other heap object. In addition, every IHGC work increment to scan the roots has a run-time of exactly one memory cycle as explained in Section 5.4.1. Thus, root marking does not pause the user's program.

During root marking, the IHGC scans the processor's register file in search for pointers that need to be marked. The mark operation is time-constant once a register is found to contain a pointer. First, the collector uses the pointer handle to load the object's mark and deep flags from the directory. If the mark flag is set, then the object is already marked and no action is taken for it. If the mark flag is unset, then it is toggled. The object's handle is also added to the *next* list if its deep flag is set; this ensures that the object is scanned later in search for pointers to other live objects.

It is hard to estimate the number of registers that contain a pointer during root marking. So we obtain a safe worst-case estimate by assuming that every register contains a pointer. This assumption still yields a reasonable bound because the time to mark a register is very short. Also, modern RISC architectures have few registers, usually no more than 32, so the root set is small. Scanning and marking a pointer takes t_{r_1} memory cycles for each register. There is also a small, constant overhead t_{r_2} associated with transitioning to the next part of the collection cycle. If the processor has k registers, we get

$$t_{roots} = kt_{r_1} + t_{r_2} \tag{6.7}$$

6.4.3 Mark Objects (t_{mark})

The collector processes the *next* list until it is empty. At each iteration, the IHGC pops a handle from *next* and loads the metadata of the corresponding object from the directory. These operations take t_{m_1} memory cycles per scanned object to be performed. Then, every word of memory from that object is loaded and scanned one at a time in search for pointers at a cost of t_{m_2} memory cycles per word. Also, the pointers found are marked with the same mechanism used in root marking; the operation is time-constant and takes t_{m_3} memory cycles.

The duration of the mark stage is proportional to the number and size of live objects, but it is also extremely dependent on the program's behavior. For example, marking takes longer as the number of pointers in memory increases. Therefore, our analysis relies on the program parameters in Table 6.2 to improve our estimate of the collector's run-time. We use this information as follows:

- The collector processes at most d deep and live objects for marking instead of all live objects. Similarly, only s words of memory from deep objects are actually scanned during marking. This is because the IHGC relies on the deep flag stored in the directory to avoid scanning objects that do not contain pointers.
- The cost t_{m_3} is only paid for each pointer in memory that is processed. So this overhead is incurred at most p times.

Including the time t_{m_4} required to initialize and terminate the mark objects stage of the collection cycle we get

$$t_{mark} = dt_{m_1} + st_{m_2} + pt_{m_3} + t_{m_4} \tag{6.8}$$

The program parameters in Table 6.2 are not essential for our analysis. It is ideal to have this information to compute tight run-time estimates of the collection cycle. But worst-case values can be used instead if the information is not available.

Parameter	Description
p	Number of pointers in live memory.
d	Number of live objects that are also deep.
s	Number of words allocated for live objects that are also deep.
c	Number of live objects that are never relocated.
z	Minimum allocation size (in words).

Table 6.2: Program parameters used to refine the estimated run-time of a collection cycle (t_{gc}).

6.4.4 Compact ($t_{compact}$)

The collector relocates live objects at one end of the memory space and simultaneously zeroes garbage memory. Compacting relies on the mark information in the directory and the header word in memory that contains an object's handle. The collector uses *src* and *dest* pointers and sets them to the lowest memory address at the beginning of the compact part.

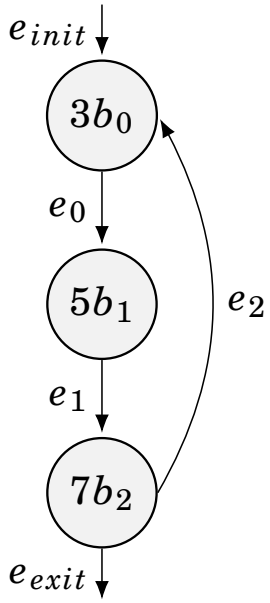
The IHGC must inspect every live and garbage object while compacting. It spends up to t_{c_1} memory cycles per object performing the following setup. The object's metadata is loaded using the handle in the header word at *src*. If the mark flag is set, then the object is live and, if necessary, the collector starts copying it to the address referenced by *dest*. Otherwise, the object is garbage and its handle is added to the *free* list. The collector starts zeroing the unmarked object if it will not be overwritten by relocated live memory. Unfortunately, it is difficult to estimate the number of garbage objects reclaimed in a collection cycle. So we derive a safe upper bound on the total number of objects processed during compact using the information in Table 6.2. There are c objects that are live, each of at least z words in size, using n words of memory. In addition, there are up to $m-n$ words of memory containing at most $\frac{m-n}{z}$ objects. Therefore, there are up to $c+\frac{m-n}{z}$ objects that the collector must process during compact, but not necessarily relocate.

We must also account for the time required to zero and copy memory. Once again, it is difficult to tightly estimate the total run-time of these operations as we do not have detailed information about garbage objects. But we know that it takes longer to copy a word of memory than to zero it. Also, the amount of zeroing work decreases as copying increases, so the worst-case run-time happens when copying work is maximized. This occurs when the collector must retain and relocate almost the full memory m, or $r+n+2w$ according to Equation 6.3. However, n words are never relocated, so $r+2w$ words of memory are copied during compact in the worst-case.

Copying a word of memory takes t_{c_2} memory cycles. Including the time t_{c_3} taken to terminate the compact operation we have

$$t_{compact} = \left(c + \frac{m-n}{z}\right) t_{c_1} + (r+2w) t_{c_2} + t_{c_3} \tag{6.9}$$

Simplifying using Equation 6.3 we get



(a) Control Flow Graph (CFG).

```
1  // Start and end constraints
2  e_init = e_exit = 1
3
4  // Other constraints
5  b_0 = e_init + e_2 = e_0
6  b_1 = e_0 = e_1
7  b_2 = e_1 = e_2 + e_exit
8
9  // Loop bound constraints
10 0 < b_0 ≤ 10
11
12 // Objective function
13 3b_0 + 5b_1 + 7b_2
14
```

(b) Integer Linear Programming (ILP).

Figure 6.2: Control Flow Graph (CFG) and its corresponding Integer Linear Programming (ILP) problem for a simple program with a loop.

$$t_{compact} = \left(c + \frac{r+2w}{z}\right) t_{c_1} + (r+2w) t_{c_2} + t_{c_3} \tag{6.10}$$

6.5 Static Program Analysis

The real-time garbage collection analysis presented in Section 6.3 and Section 6.4 relies on parameters given by the program's behavior. In this section, we describe how existing real-time program analysis techniques can be adapted to estimate some of the required information. We also cite the relevant literature proposing techniques to estimate parameters for which this thesis does not provide an analysis.

6.5.1 Memory Allocated (a) and Spare Memory Cycles (t_f)

The equations in Section 6.3 rely on two important parameters given by the behavior of the program: the allocated memory a and the number of spare memory cycles t_f available to the collector. We estimate these values using an automated tool that relies on techniques similar to those used for conventional Worst-Case Execution Time (WCET) analysis [186].

Our automated tool starts its static analysis by constructing a *Control Flow Graph (CFG)* of

the compiled program supplied as an input.[1] A CFG is a directed graph that encodes all possible paths that a program might execute. The nodes of a CFG represent blocks of code without branches or branch destinations, while the edges are branches as shown in Figure 6.2(a). Each node is associated with a coefficient that represents the cost of executing it. For example, the code from block b_0 allocates 3 words if the CFG in Figure 6.2(a) was used to calculate the amount of memory allocated by a program a. Alternatively to calculate t_f, the coefficient would indicate that 3 memory cycles are available to the IHGC when executing block b_0.

The second part of the static analysis uses *Implicit Path Enumeration (IPET)* to calculate a bound for the resource usage of the program [100]. IPET derives an *Integer Linear Programming (ILP)* problem by combining the flow information and coefficients encoded in the CFG [41]. The ILP is an optimization problem consisting of an *objective function* and a set of constraints over the variables used in that objective function. For example, the constraints in Figure 6.2(b) indicate that block b_0 in the loop will be executed at least once and at most 10 times. Also, the objective function shows that blocks b_0, b_1 and b_2 have 3, 5 and 7 spare memory cycles if this ILP was constructed to estimate t_f.

The ILP's objective function and most constraints are constructed with a simple graph traversal of the CFG. Each block has a term that corresponds to its coefficient times a unique variable name; the objective function is a summation of these terms for all blocks as shown in Figure 6.2(b). The majority of constraints relate the number of times that the edges are followed with how often a block is executed. For example, the solution to the objective function in Figure 6.2(b) must satisfy the constraint $b_1 = e_0 = e_1$ meaning that the program must follow the input edge e_0 and output edge e_1 as many times as block b_1 is executed. Loop bounds are the only constraints that are difficult to construct as the information is not readily available in the CFG, so these bounds are either input manually by the programmer or automatically estimated using a tool like SWEET [61].

Finally, our analysis tool solves the objective function in the generated ILP formulation using the open-source tool lp_solve [34]. A solution for a generated ILP either maximizes or minimizes the objective function while satisfying the associated constraints. For example, the IHGC's real-time analysis requires estimating the maximum value for a because it produces the worst-case memory requirements (m) to not pause the user's program according to Equation 6.5.

6.5.1.1 Cost Models

The coefficients from the CFG are given by a *cost model*. These models capture the amount (or cost) of resources, such as time or memory, consumed by each operation in a specific system. Cost models are often difficult to construct because they require in-depth knowledge of the system's microarchitecture.

[1] Our automated tool is based on software originally intended for static analysis of energy consumption [72]. The source code of our tool was made available under an open-source license [10].

Instruction	Clock Cycles	
	Run-time	**Memory**
`str r7, [r0, r1]`	2	2
`add r1, r1, #4`	1	0
`mov r2, #100`	1	0
`cmp r1, r2`	1	0
Total	5	2

Table 6.3: Run-time and number of clock cycles spent in memory accesses for a block of code. This, and instruction fetch information is used by our cost model to estimate t_f.

Our static program analysis tool requires two cost models. The first estimates a by associating each CFG node with the worst-case amount of memory allocated by its corresponding block of code. This technique is similar to that presented in previous research [137], but alternative analysis methods have also been proposed [105]. Our tool easily identifies blocks that allocate memory because they contain the `newm` instruction.

The cost model for t_f associates each CFG node with the number of unused memory cycles for program execution or instruction fetches. The unused cycles of a block of code are given by the difference between the run-time and the memory cycles used by that block of code. For example, the total run-time of the instructions shown in Table 6.3 is 5 clock cycles. The code uses 2 of those cycles to access memory while storing a word of data (`str`). Assuming that two instructions are retrieved per fetch, the processor also performs memory accesses during 2 clock cycles as it needs to fetch instructions. So the cost coefficient for the block of code in Table 6.3 is $5-2-2=1$ when calculating t_f.

Estimates for a and t_f are calculated using separate CFGs with different cost models as explained above. In addition, the ILP formulations are solved independently with lp_solve by maximizing a and minimizing t_f as this produces safe worst-case bounds for m according to Equation 6.5. However, this may also cause overestimates because in some cases the execution path with worst-case a does not cause a worst-case t_f. For example, the worst-case a in the program in Figure 6.1 occurs when the condition is true, but the worst-case t_f occurs when the condition is false.

6.5.2 Live Size (n, r, d, s, c)

Live size information is vital for most real-time garbage collection analysis techniques (including ours) [28, 79, 92]. Empirical estimations are not acceptable because there is no guarantee that the values correspond to the program's safe worst-case requirements. Therefore, various techniques have been proposed in the literature to statically estimate a program's live size with different degrees of success [5, 92, 131, 180]. But this is an inherently difficult problem

```
1  if (i == 0) {
2      // Allocate memory and perform a computation
3      // that requires few memory operations
4      p = malloc(1000);
5      for (i = 0, sum = 0; i < 1000; i++)
6          sum += i;
7  } else {
8      // Execute many operations that require
9      // memory, such as copying
10     memcpy(p, q, 1000);
11 }
```

Listing 6.1: Program that causes an overestimate for m.

as it is undecidable.

The approach in this thesis is to assume that the programmer is aware of their program's memory consumption. This is generally true in real-time and critical systems design because programmers must choose the appropriate memory size, and other hardware requirements, for their applications. The task of identifying the program's live size can be facilitated by conventional software development tools like compilers that report object sizes. There are also specialized tools, such as AbsInt's StackAnalyzer [3], to statically estimate the function call stack size.

6.5.3 Number of Pointers (p)

The number of pointers that the IHGC processes during marking substantially impacts the duration of a collection cycle. So it is important to statically estimate the total number of pointers in live objects in order to calculate tight worst-case memory requirements. Compilers for strongly typed languages, such as Java or C#, can easily determine the number of pointers in an object. Compilers for weakly typed languages, such as C++, can also determine the number of pointers in an object as long as the program is carefully written to only store pointers in words of pointer type. Programmers can use this compile-time information, along with the live size, to estimate an upper bound on the number of pointers that will be processed during a collection cycle.

6.6 Summary

We presented a real-time static analysis technique for the Integrated Hardware Garbage Collector (IHGC). Our analysis estimates the worst-case memory size requirements to guarantee that the IHGC will never pause the user's program. The technique relies on the IHGC's ability to interleave its operations with memory accesses from the processor to complete its work. As a result, the program is never paused if the system is provisioned with enough space to satisfy allocations while memory is reclaimed. The processor is fully utilized to execute the user's program without the high performance overheads incurred by software real-time garbage collectors. Also, the

IHGC's operation does not need to be considered in the process scheduling analysis of a real-time system. This is another advantage over existing time-based real-time garbage collectors implemented in software as their analysis is more complex.

Our technique uses information that is extracted via static program analysis. This is because the IHGC's timing model depends on the program's behavior. Therefore, we developed an automated static analysis tool that uses techniques from conventional real-time analysis to estimate essential program parameters. We also indicated other techniques from the literature or strategies to obtain the remaining program parameters that the analysis relies on, such as the pointer density.

The IHGC offers reliability and run-time predictability features that are extremely difficult to achieve in software. In addition, the analysis presented is a step towards enabling the use of modern garbage collected programming languages for hard real-time embedded systems.

CHAPTER 7

EXPERIMENTAL EVALUATION

In this chapter we evaluate the IHGC's design and our hard real-time analysis technique presented in Chapter 5 and Chapter 6 respectively. A simulator of the IHGC alongside a processor implementing an existing ISA is used extensively to empirically evaluate the collector's performance and compare it with a conventional system that does not have the IHGC. We also apply our real-time analysis to practical programs and verify that pauses are indeed eliminated.

Specific considerations for the implementation of a complete garbage collected system in hardware are not discussed in this chapter. Refer to Chapter 9 for a detailed discussion on this subject.

7.1 Evaluation Platform

Innovations in computer architecture are normally evaluated in well-established simulation platforms that have been validated and peer reviewed by the research community. Example simulators include Sniper [44], ZSim [144] and gem5 [35]. However, these tools are designed for evaluating large scale, high performance systems rather than the embedded devices targeted in this thesis. In fact, Sniper and ZSim only simulate the x86 ISA which is rarely, if ever, used in small embedded devices nowadays. We are not aware of tools for embedded systems, so we developed an open-source simulator in C++ to evaluate the performance of 32-bit embedded processors with and without the IHGC [8].

We simulated the 3-stage pipeline of an ARM Cortex-M0 processor. This processor is typically used in the hard real-time applications that we are targeting because its timing behavior is predictable. However, it is important to highlight that the principles behind the IHGC can be used in conjunction with other architectures, such as MIPS and RISC V, and our choice was driven by the need to find benchmark programs to evaluate our platform.

Our simulator is timing-accurate as each instruction's cycle count is implemented according to the latencies stated in the ARM Cortex-M0 Technical Reference Manual [19]. The directory is assumed to be a memory component independent from the main memory. It is sufficiently fast to perform an address lookup followed by an address calculation, i.e. an addition, within a single clock cycle. In contrast, the main memory takes a full clock cycle to operate. This assumption is reasonable in the context of embedded systems as discussed in Chapter 9. A fast directory memory ensures that the latency of store instructions remains unchanged in processors with and without the IHGC. In contrast, the short processor pipeline causes occasional stalls when pointers are marked during a load instruction; this is accounted for by the simulator. But these delays can be easily eliminated at the expense of minimal extra hardware in the pipeline as discussed in Section 5.6.

The IHGC's tag bits are modeled as extra tag bits in our simulator as described in Section 2.6. We assume that the system is fitted with physical registers and memory that are wider than the processor's native word size i.e. 33-bit words since the Cortex-M0 is a 32-bit processor. Every access to a word in memory or the registers operates on 33 bits: one bit is reserved for the tag and invisible from the program while the remaining 32 bits hold the regular pointer or data value. This facilitates compiling and running existing software on our simulated platform using off-the-shelf compilers because standard integer types are supported. Furthermore, relying on a wider word length in embedded devices is an acceptable design choice because memories are usually on-chip; interfacing with external memory devices, that normally have a word size which is a multiple of 8 bits, is not a major concern.

7.2 Benchmarks

Benchmark suites, such as DaCapo and the SPEC Java benchmarks [36, 168], have been widely used in both academia and industry to evaluate the performance of garbage collectors. These suites are composed of programs typically found in large server and high performance systems. But the profile of these programs is not representative of real-time or embedded applications. The benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC) are perhaps the most widely used and accepted to evaluate embedded systems [60]. However, the EEMBC benchmarks are only small kernels, instead of complete applications, that do not stress the memory manager. In fact, there are no benchmark suites tailored to evaluating garbage collectors in embedded systems. This is because automatic memory management, and dynamic memory in general, is often avoided as existing memory management algorithms incur high overheads and are mostly infeasible for real-time applications. As a result, it is difficult to find programs to evaluate novel garbage collectors for embedded systems like the IHGC.

Since there are no benchmark suites that fit our needs, we adapted widely available open-source software to evaluate the IHGC. The programs are grouped into three benchmark suites:

BEEBS, MicroPython scripts and large C programs. The first is a subset of the Bristol/Embecosm Benchmark Suite (BEEBS) tailored to measure the performance and energy consumption of embedded devices [130]. BEEBS is mostly composed of small C programs that run simple numeric operations and classic algorithms like quicksort. These benchmarks rarely use dynamic memory, so they measure the IHGC's overheads on memory access instructions in isolation from memory management operations.

For the second benchmark suite, we ported MicroPython to run on our simulated system. MicroPython is a Python interpreter for embedded devices [115] that we set up to execute scripts from the Python Benchmark Suite [66]. We made minor modifications to the original scripts to only use the subset of Python supported by MicroPython. The interpreter already included a software collector, so the porting work was limited to replacing this functionality with the IHGC.[1]

The third benchmark suite is composed of large C programs found in real applications. These benchmarks were partly developed by us and include:

1. Three programs from the Timing Analysis on Code-Level (TACLe) benchmarks that are used for real-time analysis research [62]. These programs were modified slightly to use dynamic memory instead of static allocations and to avoid floating-point arithmetic.

2. Three programs that communicate over a simulated network using the TCP and UDP protocols, and in one instance the data is secured with the TLS protocol. We used the popular open-source software lwIP [59] and Mbed TLS [22] to implement these benchmarks.

3. One program that benchmarks the construction of embedded graphical user interfaces, such as those found in printers and home appliances. We developed this software using the open-source library LittlevGL [95].

7.3 Compiler and Toolchain

All benchmarks were compiled with LLVM version 7 [102] and linked with GCC for ARM Embedded Processors version 7.3.1 [21] while having -O2 and other optimizations enabled. However, compiling with this toolchain was problematic because the generated code does not differentiate between pointer and regular value types. So programs routinely execute instructions that may compromise the IHGC's safety and correctness. For example, executing a bitwise-or instruction with two pointer operands may result in a new pointer being created to a memory location that the process is not permitted to access. Therefore, the semantics of many arithmetic and bitwise instructions from the ARMv6-M ISA [20], the instruction set implemented by the ARM Cortex-M0, were originally changed to error when the operands are pointers. Unfortunately,

[1]The porting code for MicroPython along with the modified scripts from the Python Benchmark Suite are open-source [7].

this was incompatible with the code generated by LLVM, so the restrictions were eliminated from the simulator for the sake of this evaluation.

The IHGC does not treat the program call stack preferentially. The stack is marked and scanned as if it were any other allocated object without considering that stale frames previously popped off the stack are not needed by the program. So memory requirements may increase beyond the program's actual needs because the IHGC considers as live the garbage objects referenced by pointers stored below the stack pointer. This is a consequence of allocating the function call stack as a contiguous block of memory in the ARM architecture. Allocating stack frames individually upon entering each function avoids this problem. This approach eliminates the need to predefine stack sizes, so it improves reliability as stack overflows are eliminated. Also, efficiently supporting individual stack frame allocations only requires minor extensions to the architecture.

The issues discussed in this section are not limitations of our system. The problems can be addressed by introducing minor changes to the ISA or modifying the compiler's backend to select the appropriate instructions when generating code for an IHGC system. This is revisited and solutions are outlined in greater detail in Chapter 8.

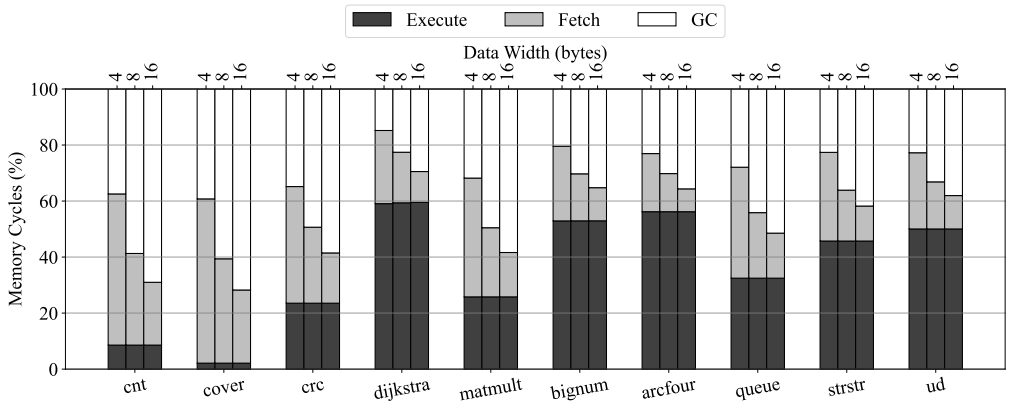
7.4 Measuring Performance

In this section, we present and discuss our experimental results. Most of the benchmarks considered are not real-time programs, so the static analysis technique presented in Chapter 6 was not used and pauses can occur. The aim is to understand the run-time performance and memory usage of our collector as well as empirically measure pause times.

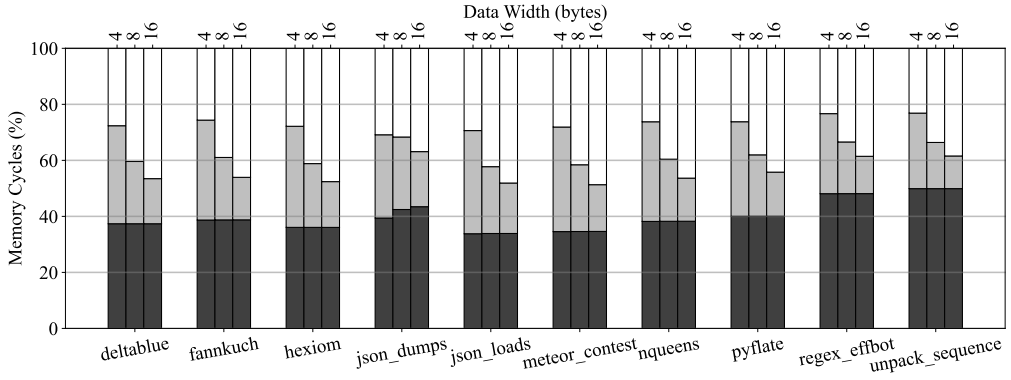
7.4.1 Characterizing Memory Cycles

The IHGC takes advantage of spare memory cycles to operate as described in Section 5.4. This implies that the system's performance degrades if not enough memory cycles are available for collection operations. In this case, the system effectively allocates memory at a higher rate than the collector can reclaim it, so an out-of-memory condition will eventually arise and pause the program. Therefore, it is important to understand the impact of the main two consumers of memory cycles when executing a program: memory access instructions and instruction fetches.

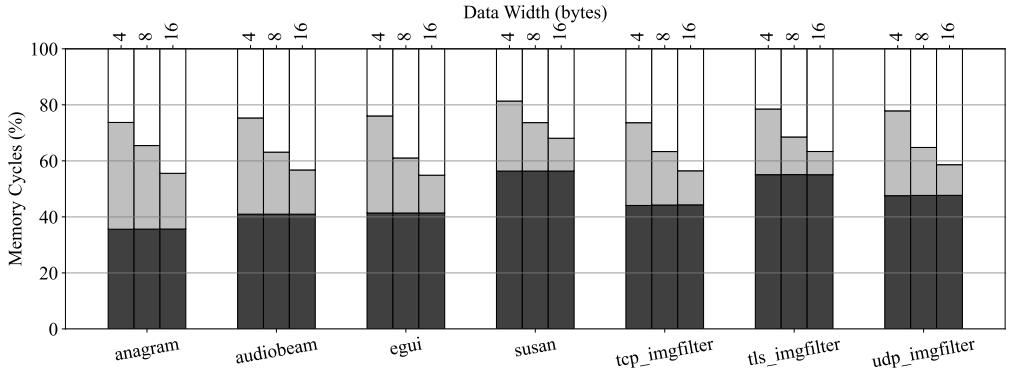
The collector does not operate during the short amount of time it takes the processor to execute memory access instructions, such as loads and stores. As a result, programs that frequently access memory decrease the spare memory cycles usable for collection operations. Figure 7.1 shows that our benchmarks spend about 5-60% of memory cycles executing memory access instructions in a system with the IHGC. It is also clear that the majority of these benchmarks use about 35-50% of memory cycles to execute memory instructions; specifically, this is the case for the larger and more realistic programs in Figure 7.1(b) and Figure 7.1(c). However, the number of



(a) BEEBS.



(b) MicroPython.



(c) Large programs.

Figure 7.1: Distribution of memory cycles in the IHGC system as the amount of data loaded per instruction fetch increases. The data width is the size of the hardware memory bus and in these experiments corresponds to the number of bytes loaded per fetch operation. The program binary is the same for all experiments as the software does not take advantage of the larger data widths to optimize performance; memory accesses for instruction execution are limited to 4 bytes. The bars labeled 'GC' correspond to the proportion of memory cycles available to the IHGC, but not all these memory cycles are necessarily used.

memory instructions executed can change significantly depending on the program. For example, benchmarks that mostly run numeric operations, such as *cnt* and *cover*, only use 5-10% of memory cycles for memory access instructions. In contrast, *dijkstra* and *susan* devote almost 60% of memory cycles for the same purpose. As we shall see in Section 7.4.2 and Section 7.4.3, this leaves enough spare cycles to achieve few pauses and similar or better performance compared to a system without the IHGC.

The spare memory cycles also decrease if the processor fetches instructions too often. The fetch rate of a system is mostly dependent on the instruction set encoding and the width of the hardware data bus connecting the processor to the main memory. For example, fetches in our simulated platform are expected to consume 50% of all memory cycles because the ARM Thumb instructions from the ARM Cortex-M0 processor are mostly encoded in 16 bits and the data width is 4 bytes, so the processor normally loads two instructions per fetch. In general, our experiments indicate that about 30-40% of memory cycles are dedicated to instruction fetches when the system loads 4 bytes per fetch as shown in Figure 7.1. This is less than the expected 50% because the processor is pipelined, so fetches are often performed while the pipeline is waiting for the result of a load or store instruction. The IHGC could not have used the memory cycle regardless of these fetches, so they do not decrease the number of memory cycles available to the collector.

In our experiments, the larger data widths are only used to increase the number of instructions fetched simultaneously. The program cannot take advantage of this feature to optimize run-time performance as up to 4 bytes only can be loaded or stored using a memory access instruction. Therefore, we would expect the amount of memory cycles used to execute memory instructions to remain constant regardless of the data width because the program binary used in all experiments is the same. But Figure 7.1 shows that this is not the case for programs like *json_dumps* and *tls_imgfilter*. The discrepancy occurs because larger data widths increase the collection rate affecting the program in two ways. First, the marking stage is performed more often, so memory cycles are taken away from the IHGC because the processor marks on load more frequently. And second, the pointer handles change when the collection rate increases, so programs that conditionally execute code based on pointer comparisons have slightly different run-time performance. As a result, Figure 7.1 shows that the proportion of cycles used to execute memory access instructions change by up to 5% when the data width increases.

As illustrated in Figure 7.1, doubling the data loaded per fetch from 4 to 8 bytes reduces the frequency of fetches by about a third. However, a further doubling from 8 to 16 bytes reduces the memory cycles consumed for instruction fetching in a smaller proportion. This is because the likelihood of finding branches within the instructions fetched increases. So the processor will rarely be able to completely consume the 16 bytes of data before fetching once again.

In summary, 20-30% of memory cycles are available to the IHGC in the larger and realistic benchmarks in Figure 7.1(b) and Figure 7.1(c). But this proportion can change significantly depending on the type of work that the program performs as shown in Figure 7.1(a). In addition, the

proportion of memory cycles for instruction fetching can be reduced by loading more instructions simultaneously. This is a simple mechanism to easily increase the spare memory cycles for the collector without impacting the program's performance.

7.4.2 The IHGC and Software Memory Managers

Comparing the IHGC with existing software memory managers is difficult. Off-the-shelf memory managers generally do not provide the same reliability guarantees that our system offers. However, for the sake of this discussion, we compared the run-time and memory overheads of the IHGC with two explicit memory management algorithms and a software garbage collector.

7.4.2.1 Explicit Memory Management

Explicit memory management requires the programmer to manually free memory when it is no longer needed. For example, C programs typically allocate memory by calling the `malloc` function and release it by calling `free`. Failing to release memory when it becomes unused, also known as *leaking memory*, degrades performance and can even lead to complete failure because the memory manager cannot repurpose enough space to serve future allocations. The implementations for `malloc` and `free` are generally provided by the C Standard Library (or libc) that is linked with the user's program at the end of the compilation process. For our experiments, we used Newlib [138], the libc that is included in the installation of GCC for ARM embedded processors. We linked the BEEBS benchmarks with a minimal subset of Newlib and ran experiments using two explicit memory management algorithms from this libc: *dlmalloc* [98] and *nano-malloc*.

Figure 7.2 compares the IHGC's run-time with the explicit memory managers from Newlib running on a conventional processor. The benchmarks in the graph can be classified in two groups. First, the benchmarks *dijkstra*, *levenshtein* and *listsort* which are the only three programs in Figure 7.2 that rely on dynamic memory. In this case, the IHGC is 1-2 times faster than the conventional processor because collection operations in our system do not incur overheads for the programs and the cost of allocations is negligible. In contrast, a share of processing time must be dedicated to execute the explicit memory manager in the conventional system. It is also clear from Figure 7.2 that the difference between the IHGC's run-time performance and the other system changes significantly depending on the explicit memory manager used. For example, nano-malloc is designed to be a simpler algorithm than dlmalloc, so the benchmarks appear to be slightly faster when using nano-malloc. However, the tradeoff is that nano-malloc is less resilient to fragmentation.

The second group in Figure 7.2 consists of all the benchmarks that ran with comparable performance in both systems, i.e. showed a 0.8-1.0 speedup factor. None of these programs use dynamic memory, so the compiled code for both our system and the conventional processor is exactly the same. These benchmarks are effectively measuring the IHGC's run-time overheads as a result of longer latencies when loading from memory, but these can be eliminated as discussed

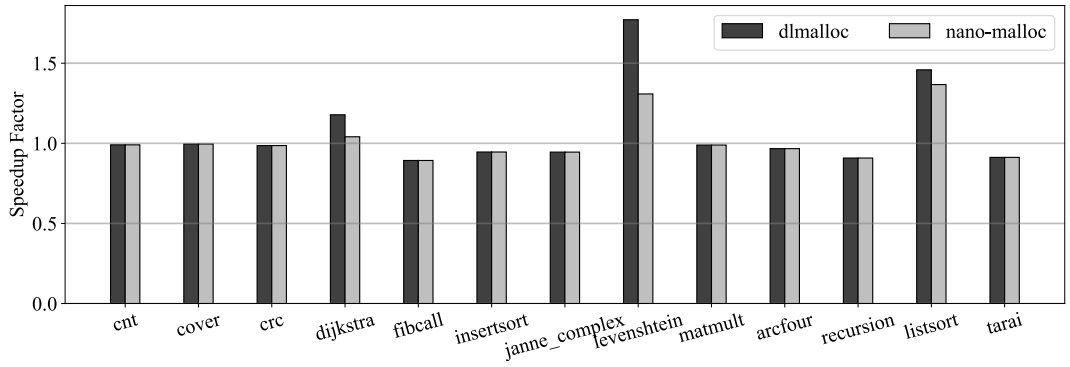


Figure 7.2: Run-time performance change of BEEBS on the IHGC system when using a conventional processor as a baseline. Separate measurements were taken when the program running on the processor without the IHGC was using two explicit memory allocators (dlmalloc and nano-malloc) from the Newlib libc.

in Section 7.1. Once again, programs with fewer memory access instructions, such as *cnt* and *cover* from Figure 7.1(a), experience very low overheads. Programs with higher rates of memory access instructions have slightly higher run-time penalties. For example, *recursion* and *tarai* are both small recursive programs that spend most of their time setting up and tearing down the function call stack frames, so about 50% of their memory cycles are spent on executing memory access instructions (most of which are loads). However, it is clear from Figure 7.2 that most programs have very modest overheads.

In practice, embedded systems often require large programs for tasks, like networking and security, that rely on dynamic memory management. Given our experimental results, we expect these programs to have comparable or better performance when running in our system compared to a conventional processor. This is because the IHGC normally operates on spare memory cycles, that would otherwise be unused, without run-time penalties for the user's program. In contrast, a traditional program written in C that relies on explicit memory management inevitably uses the processor to fetch instructions and execute `malloc` and `free` in software.

7.4.2.2 Software Garbage Collector

The memory manager that has the closest feature set to the IHGC's is a software garbage collector. These collectors are commonly employed in interpreters to support garbage collected languages like Python and Java. So we compared the run-time performance of MicroPython in processors with and without the IHGC. MicroPython is an interpreter tailored to small embedded devices for a subset of the Python 3 language. It uses a conservative mark-sweep collector operating in a stop-the-world fashion [114].

The heap memory requirements for the IHGC and the software collector are very different for the *json_dumps*, *meteor_contest* and *pyflate* benchmarks as shown in Figure 7.3. There are three

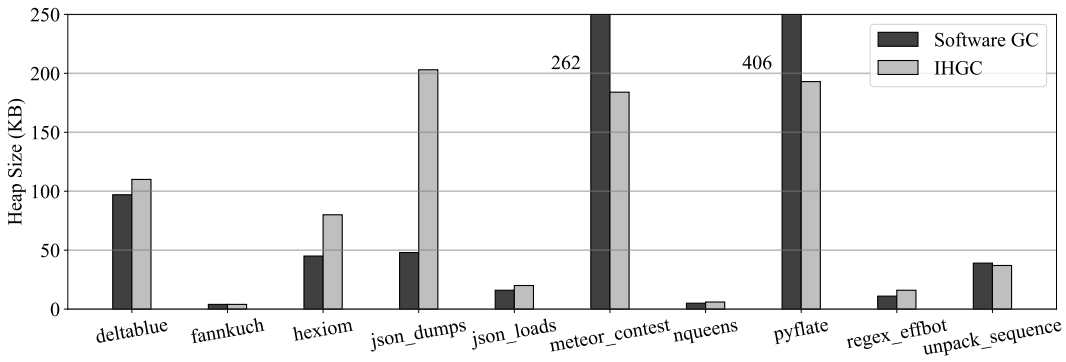


Figure 7.3: Heap memory requirements of the MicroPython benchmarks. The measured heap size corresponds to the minimum amount of memory required to store all the program's reachable objects, i.e. the minimum amount of heap memory required to run the program.

main problems causing the mismatch. First, MicroPython's software collector does not compact memory and suffers from fragmentation, so its memory requirements can be substantially higher than the IHGC's. Second, we disabled the safety checks that prevented the program from manipulating pointer handles as explained in Section 7.3. Unfortunately, in this setup our collector is no longer exact because the program can construct pointers to arbitrary objects while manipulating values. These arbitrary objects may actually be dead, but the collector does not reclaim them because it believes that there is a pointer to them. In reality, these pointers are merely data values from the program's point of view. And third, the stack is allocated as a contiguous block of memory in the IHGC, so objects below the stack pointer are often retained despite being dead. As a result, the heap memory requirements for the IHGC and the software collector are very different which makes performance comparisons difficult. For example, the *json_dumps* benchmark appears to run slower in the IHGC than with the software collector. However, this is only due to the hardware collector having to process a heap that is over four times larger than that of the software collector which clearly takes significantly longer.

It is also important to consider the amount of memory processed by the collectors when comparing the IHGC and the software collector. In MicroPython, it is guaranteed that the code and global data sections do not contain pointers that need to be traced. So the software garbage collector only needs to process the stack and the heap. In contrast, the IHGC is independent of the user's program and traces the full memory space. This is necessary in our setup because the LLVM compiler includes pointers within the code and global sections of the program. This means that our collector is processing in excess of 170 KB of memory more than the software collector. Unfortunately, the difference in setup is crucial when comparing the two systems based on programs that have relatively small heaps. For example, according to Figure 7.4 *fannkuch* and *nqueens* appear to run slower in the IHGC system when the heap is small (see Figure 7.3). This occurs because the heap is not large enough to give the hardware collector sufficient time

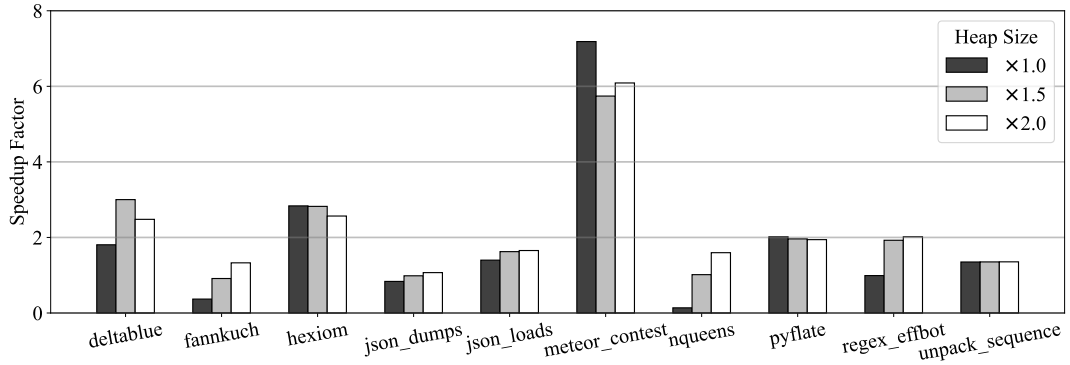


Figure 7.4: Run-time performance change of the MicroPython benchmarks on the IHGC system as the heap size increases. A conventional processor running MicroPython with its software garbage collector was used as a baseline.

to process the code object before an out-of-memory condition is reached. In fact, increasing the heap by only 50% improves our system's performance by a factor of 7.3 for *nqueens*. It is also important to highlight that this is not a limitation of our design and it is easy to prevent the collector from tracing the code object by using a toolchain that takes advantage of the IHGC's features as discussed in Chapter 8.

Despite the limitations of the comparison mentioned above, the IHGC outperforms the software collector by a factor of 1.5-7 in several of the MicroPython benchmarks as shown in Figure 7.4. This gain is thanks to our interleaving technique that executes the collector without run-time overheads. It can also be seen in Figure 7.4 that the performance changes significantly depending on the heap memory size. Increasing the heap size may result in better performance by helping the collector avoid pausing the program due to out-of-memory conditions. For example, *deltablue* is paused by the collector for about 25% of the time when the system is supplied with the minimum amount of memory needed to run the program, i.e. ×1.0 heap size in Figure 7.4. However, the performance for *deltablue* improves by a similar proportion when the heap size is increased by 50%. This change occurs because increasing the heap size gives our collector more time to reclaim garbage before the system runs out of free memory to satisfy allocation requests. Clearly, the performance may not improve when the heap size increases as is the case with *pyflate*, *hexiom* and *unpack_sequence* in Figure 7.4. This is because the collector was already fast enough to reclaim memory without pausing, so changing the heap size has no effect. Furthermore, note that the software collector experiences a similar effect in benchmarks like *hexiom* where the IHGC's performance gain decreases slightly as the heap size increases.

The data presented in Figure 7.4 was gathered when the IHGC was configured to simultaneously load 4 bytes per instruction fetch. This parameter was chosen because increasing it does not affect the run-time in the conventional processor as the software garbage collector cannot take advantage of this hardware feature. In practice, the run-time performance for some MicroPython

benchmarks improves a further 10-20% by doubling the amount of data loaded per fetch without increasing the heap size. As discussed in Section 7.4.1, this is because the collector has more spare memory cycles to operate without any overheads for the program, so the time spent paused is greatly reduced when the heap size is small relative to the size of the live data.

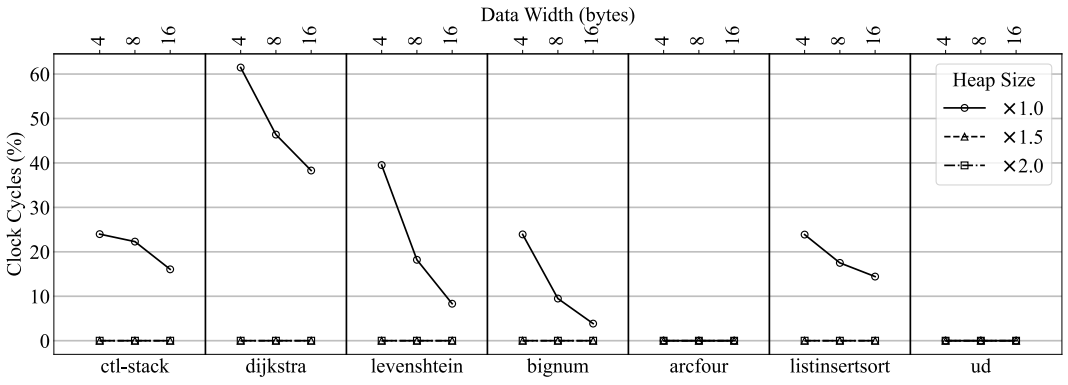
7.4.3 Pauses

Pauses are inherent to any software garbage collector. This is because the processor must eventually be used to progress collection work and the user's program regularly needs to coordinate its operation with the collector to guarantee correctness. The IHGC completely eliminates these pauses under normal operation as it runs on spare memory cycles, so the processor can be fully utilized for application work. Also, the IHGC uses hardware within the processor's pipeline to ensure correctness when programs execute load and store instructions. However, pauses can still occur if the program allocates memory faster than the collector can reclaim it. Eventually, the system will reach an out-of-memory condition because there is simply not enough free space or handles to satisfy an allocation. In this case, the program is paused until the collector reclaims enough memory to meet the immediate allocation demands.

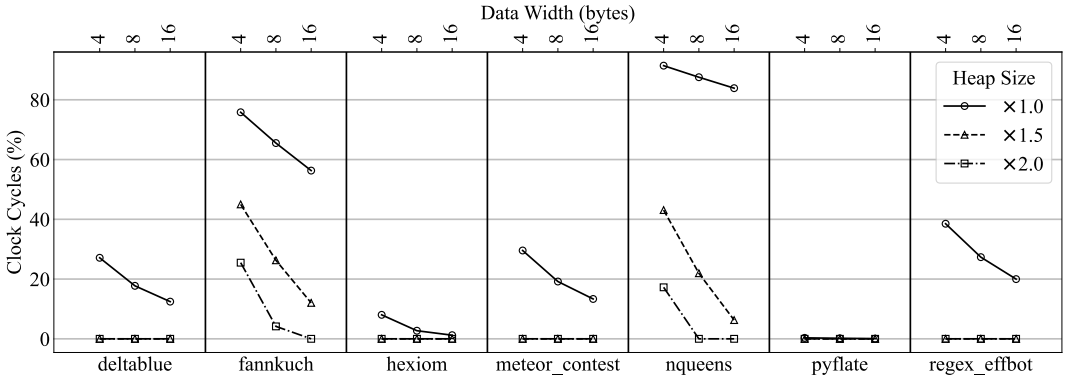
We investigated two techniques to mitigate pauses while allocating: using larger heaps and increasing the amount of data loaded per instruction fetch. Enlarging the heap beyond the minimum operating size needed by a program is perhaps the most widely used technique to reduce pauses. The idea is to supply the system with enough space so that allocations can be served immediately, so the collector has more time to complete a cycle as discussed in Chapter 6. In contrast, simultaneously fetching more instructions gives our collector more time to operate by increasing the system's spare memory cycles.

In this section, we are interested in evaluating the impact of pauses on performance. So Figure 7.5 shows the proportion of time that several of our benchmarks are paused due to collection operations as the heap size and the number of instructions fetched increases. We do not characterize the distribution of pauses in terms of MMU (as discussed in Chapter 2) because these empirical experiments do not guarantee that the IHGC is suitable for hard real-time operation. In Section 7.5, we will discuss how our proposed analysis techniques from Chapter 6 are used to guarantee that the IHGC never pauses, i.e. the MMU is 100%, so that the system is hard real-time.

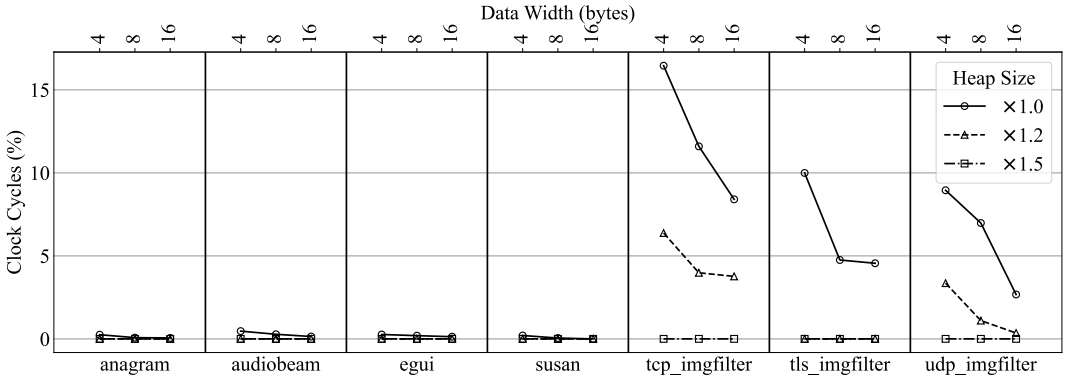
For BEEBS and the large benchmarks, increasing the heap size by a factor of 1-1.5 decreases the time spent paused dramatically as shown in Figure 7.5. In fact, only one program experiences pauses when the heap is enlarged. Doubling the amount of data loaded per instruction fetch decreases the time spent paused by up to 20% for *levenshtein* and 5% for *tcp_imgfilter* and *tls_imgfilter*. In the case of the MicroPython benchmarks, the heap size has the most dramatic effect for *fannkuch* and *nqueens*. As explained in Section 7.4.2.2, this occurs because the minimum heap size requirements for these programs are very small and the IHGC unnecessarily processes



(a) BEEBS.



(b) MicroPython.



(c) Large programs.

Figure 7.5: Proportion of time that the benchmarks are paused due to the IHGC's operations. For each benchmark, the markers correspond to measurements with different heap sizes. Also, the data width is the number of bytes loaded per fetch operation.

over 170 KB of data. Therefore, collection cycles take too long and likewise the pauses. However, the other MicroPython benchmarks show the same benefits that we observed in BEEBS and the larger benchmarks.

In summary, basic parameters in the IHGC system can be chosen by the hardware designer to achieve very low or completely eliminate program pauses. The conventional approach of increasing the heap size beyond the program's minimum operational requirements is very effective. However, the disadvantage is that memory requirements can increase significantly, which is not ideal for small embedded devices. The alternative technique of increasing the number of instructions loaded per fetch is also very effective. Our experiments indicate that this approach can reduce the time spent paused by up to 20%. It only has modest hardware costs and the technologies to support memory accesses larger than 4 bytes are widely used by the semiconductor industry. A combination of both approaches can be used in embedded devices with the IHGC to mitigate pause times.

7.4.4 Tag, Directory and Header Overheads

The IHGC has modest memory overheads due to the use of tags, directory and headers in a 32-bit embedded system. There is a 1-bit tag for every 32-bit word of memory, so tags incur a 3% memory overhead. The directory overheads are configured by the hardware designer. The directory size sets the maximum number of allocated objects in the system. The size of the physically addressable memory space also impacts the directory's size; the object's physical address needs more directory bits if there are more memory locations to address. The IHGC uses the first word of every object in memory to store a handle. In practice, the handles can always be represented in 16 bits (2 bytes) because the remaining bits in a word indicate the offset within the object. Therefore, the header is at most 16 bits in size; the remaining 16 bits from a 32-bit word can be used by programs to store object metadata, such as type information.

In summary, the overall memory overheads vary depending on the system's configuration and the size and number of objects allocated by programs. For example, the directory overheads are 6% when each directory entry requires 6 bytes and there is directory space for 1400 objects in a system with 128 KB of memory. The header overheads are less than 3% if 1400 objects are allocated and there is 128 KB of live memory. So the memory overhead per allocated object is 8 bytes due to the directory and headers in this specific case. Also including the storage requirements for the tags, the overheads are approximately 12%.

The IHGC's overheads due to tags, directory and headers are comparable to the memory requirements imposed by existing dynamic memory allocators. For example, dlmalloc requires at least one word of extra memory per allocated object [98]. Compared to existing garbage collectors, the IHGC's space overheads are substantially lower. For instance, the collectors proposed by Henriksson [79], Gruian and Salcic [74], and Stanchina and Meyer [167] incur 16 bytes of overheads per allocated object instead of the IHGC's 8 bytes. Therefore, our collector's memory

overheads are lower while offering similar features to existing algorithms from the literature.

7.5 Hard Real-Time Analysis in Practice

Many benchmark suites, including TACLe [62], BEEBS [130], SNU Real-Time [1] and Mälardalen WCET [75], have been proposed to evaluate hard real-time systems. These benchmarks are largely small kernels executing arithmetic operations or simple algorithms for sorting or stream processing. However, conventional real-time techniques cannot analyze dynamic memory allocators. So the benchmarks are written to use static memory management and actively avoid dynamic allocations. As a result, they are unsuitable to validate our hard real-time analysis technique for a garbage collected system.

The available benchmarks also reflect the scarce use of modern languages in real-time systems. Only one suite, JemBench [149], contains programs written in Java while the other benchmark suites for embedded devices are almost exclusively written in C. But the benchmarks in JemBench are mostly small programs translated from C to real-time versions of the Java standard like Safety Critical Java (SCJ). Therefore, the use of dynamic memory is avoided making the programs unsuitable to evaluate our real-time analysis for the IHGC. A related problem is that there are very few software tools, like virtual machines, compilers and interpreters, to support modern languages in hard real-time embedded systems. Multiple publications and products can be found for such software infrastructure, but the source code is unavailable or proprietary [4, 65, 134, 135].

In this section, we analyze programs from TACLe and BEEBS that demonstrate the limitations of existing benchmark suites as outlined above. To overcome some of these problems, we also developed two programs in C motivated by practical use cases that demonstrate the IHGC's hard real-time capabilities. We statically analyze these programs using the hard real-time technique from Chapter 6.

7.5.1 Real-Time Evaluation Methodology

We use the analysis techniques presented in this thesis to estimate the worst-case amount of memory required to never pause a given program. To achieve this, we construct a timing model of our simulated ARM Cortex-M0 processor and use it alongside the automated static analysis tool to estimate the program parameters necessary for the analysis, i.e. t_f and α (see Section 6.5). The ARM Cortex-M0 has very predictable timing behavior, so its timing model is simply a table indicating the time, in clock cycles, taken to execute each instruction. Our simulator is timing-accurate, so the timings for the instructions are mostly taken from the ARM Cortex-M0 Technical Reference Manual as described in Section 7.1.

In addition to the processor's timing model, the analysis from Chapter 6 requires the timings of various IHGC operations to construct a full timing model of the collector. For example, the

Benchmark	t_{gc}/t_f	a	m	**Minimum**
dijkstra	0.12	1575	15113	11963
levenshtein	4.69	33	582	206
matmult	0.40	1323	4142	1496
md5	0.02	35	9018	8948
sha	0.05	28	9121	9065

Table 7.1: Results of analyzing BEEBS and TACLe benchmarks. The amount of memory allocated per period (a) and the estimated worst-case amount of memory (m) are in words. The last column indicates the empirically measured, minimum amount of memory (in words) required to run the program.

duration of the initialization and termination operations (t_{init}) is necessary to estimate the length of a collection cycle (t_{gc}). These operations have a fixed duration in clock cycles given by our IHGC implementation in the simulator.

Calculating the worst-case memory requirements (m) for a given program involves simple algebraic manipulations once the program parameters have been extracted and the constants for the collector's timing model are available. This analysis also provides a wealth of information about the program and the IHGC. For example, the automated static analysis tool calculates the worst-case run-time of the code considered which can later be used to estimate interrupt latencies and jitter. Additionally, information about the collector's run-time can be used to find bottlenecks and even improve the design of the IHGC.

It is important to highlight that the techniques presented in this thesis are suitable for analyzing hard real-time programs only. These programs are typically designed to facilitate static analysis to guarantee that the embedded system will always meet its deadlines. Therefore, the programs considered in this section are hard real-time. In contrast, soft real-time programs do not have such strict timing constraints, so programmers normally experiment with these systems empirically to ensure that interrupt latencies and jitter are within acceptable ranges. However, unpredictable pauses and delays can still occur. In this thesis, we leave hardware garbage collection for soft real-time embedded systems as future work.

7.5.2 Real-Time Analysis Benchmarks

We analyzed programs from the TACLe and BEEBS benchmark suites (see Section 7.2). The programs analyzed are written in C and run simple numeric operations or classic algorithms. They are self-contained and suitable for embedded systems, but they do not use dynamic memory. Therefore, we modified the benchmarks to allocate memory dynamically instead of statically. We also changed the programs to periodically execute their main operation. For example, in the case of *matmult*, the program periodically allocates memory for three matrices and performs a matrix multiplication. The code executing the periodic operation was analyzed using our automated

static analysis tool to estimate a and t_f.

A summary of the analysis results is in Table 7.1. The second column shows that in *dijkstra*, *matmult*, *md5* and *sha* a full collection cycle finishes before the program's periodic operation is completed once. In this case, our estimation of m is the theoretical minimum according to Equation 6.3 and Equation 6.4 in Section 6.3:

$$m = n + r + 2a\left\lceil \frac{t_{gc}}{t_f} \right\rceil = n + r + 2a \tag{7.1}$$

In other words, the minimum amount of extra memory that our analysis predicts is twice the memory allocated during the execution of one period of the program or $2a$. For *matmult*, this corresponds to twice the size of the matrices allocated before performing the multiplication.

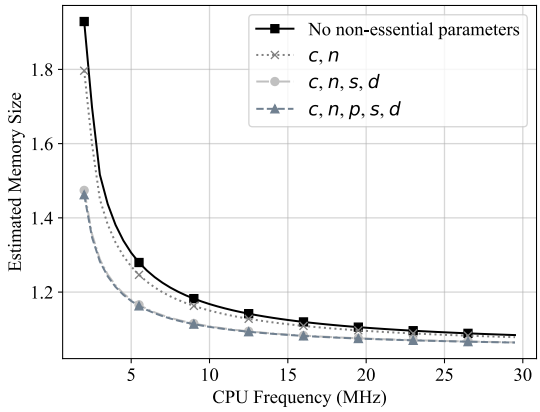
Levenshtein is an example of a program where our analysis overestimates the memory requirements. This program executes a dynamic programming algorithm that computes the string distance. The input strings given are typically very short and the task is memory intensive. Therefore, the worst-case number of unused memory cycles t_f is too small compared to the duration of a collection cycle t_{gc}. This causes our analysis technique to overestimate that the system theoretically needs more than $8a$ extra words of memory to eliminate pauses. However, we empirically measured the program's memory requirements and we never observed the extra memory needed to exceed $2a$ although this is not a guarantee that the system will never pause.

The TACLe and BEEBS programs analyzed exhibit very basic use of memory. They allocate most of the memory shortly after the beginning of each execution period. Then a (potentially long) computation is performed, and finally the memory is discarded (along with the result). This behavior is not realistic because the benchmarks were not designed to evaluate dynamic memory managers; the results shown in Table 7.1 are only included in the thesis for completeness. For this reason, we developed programs that include more complex dynamic memory usage patterns. The remaining of this section presents our analysis for such use cases.

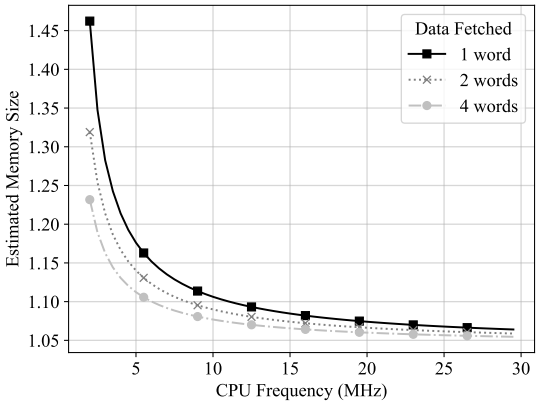
7.5.3 Case Study: Converter

We developed *converter*, a real-time program written in C, to evaluate our analysis technique. The program receives data packets from a serial device at a rate of 115200 bits per second. It must output these packets using another serial device with a data transfer rate of 9600 bits per second. The program must ensure that data is output continuously without interruption, but the input data stream can be paused.

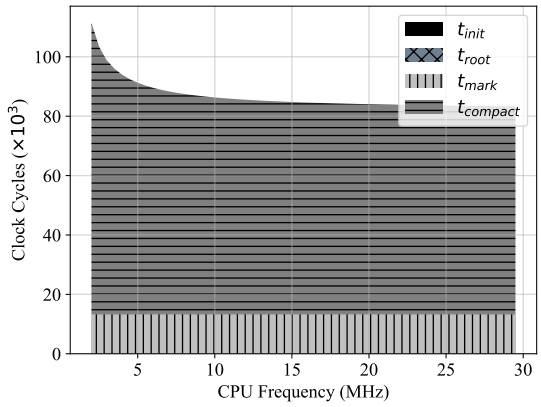
The data packets are of variable size. The minimum packet size is 64 bytes and the maximum is 1518 bytes. The program buffers up to 64 KB of incoming packet data since the input rate is higher than the output. When the buffer is full, the input stream is paused until storage space from outgoing packets can be reused. The serial devices use Direct Memory Access (DMA) to automatically move incoming and outgoing data between the memory and the serial port without



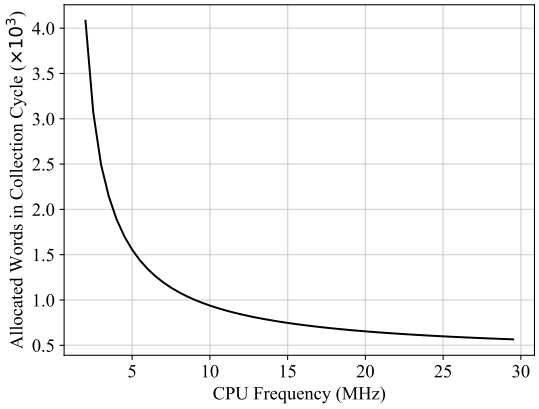
(a) Memory requirements as more information about the program is supplied to the real-time analysis.



(b) Memory requirements as the amount of data loaded per instruction fetch increases.



(c) Duration of a garbage collection cycle.



(d) Memory allocated in a collection cycle (w).

Figure 7.6: Estimated memory requirements and garbage collection cycle duration for *converter*. Higher clock speeds increase t_f and yield smaller estimates for the memory size and collection cycle duration. The memory requirements are shown as the ratio between the estimated amount of memory and the program's worst-case live size. It is assume that the duration of a memory cycle in clock cycles does not change as the clock frequency increases.

processor intervention after setup. For simplicity, we assume that the DMA does not share a port with the processor and the IHGC to access the memory, so it does not affect the number of spare memory cycles available to the collector.

The program uses polling instead of interrupts. It can determine whether the last serial operation was completed by executing instructions that check the device's status flags. As a result, the program must constantly check the serial port's status instead of being notified by an interrupt controller. This polling approach puts additional pressure on the IHGC and our analysis technique because the processor is always using memory cycles for instruction execution and is never idle.

The first step to apply our technique is to estimate the program parameters required by the IHGC's model. We estimated n, r, p, d, s, z and c by manually inspecting the code, but used the automated static analysis tool from Section 6.5 for a and t_f.

We must identify the operation that the program executes periodically to estimate a and t_f. *Converter* periodically receives a data packet via the serial device. It performs the following sequence of operations during the execution of each period:

1. Receive the fixed-size packet header containing the packet size.
2. Allocate memory for the incoming packet.
3. Receive the remaining packet data.
4. Add the packet to the output transmission queue.

The worst-case a is simply the maximum packet size. The worst-case t_f is the minimum number of memory cycles available to the collector while the program is executing the operations listed above. But receiving the data takes less time if there is less data to transfer. So the worst-case t_f occurs when receiving packets of the minimum possible size.

Using the worst-case a and t_f causes our analysis to dramatically overestimate the system's memory requirements. This is because the program will never allocate space for a packet of maximum size when it receives a packet of minimum size. Therefore, the worst-case a and t_f cannot occur simultaneously. Instead, we use every feasible combination of a and t_f pairs to calculate a list of m; the list is only as long as there are possible packet sizes. Finally, we select the maximum m from the list which gives our worst-case estimated memory requirements to guarantee that the IHGC never pauses the user's program.

The results of our analysis for *converter* are shown in Figure 7.6. For these plots, we varied the number of memory cycles available for the IHGC (t_f) by changing the clock frequency without modifying the program. Increasing the clock frequency speeds up the processor, memory and IHGC as the three components share the same clock, so the larger t_f occurs because the input and output data transfer rates remain fixed in all our experiments. In addition, the extra memory requirements are shown as the ratio between the estimated memory calculated by our analysis

$(n+r+2w)$ and the program's worst-case live size. Figure 7.6(a) shows that the best estimate produced by our real-time analysis increases memory requirements by factors of 1.06 to 1.46 for clock frequencies between 2-30 MHz. The excess memory increases further if the clock speed is dropped. For example, the estimates at 600 KHz are over 7 times larger than the size of the program's live memory. Also, our analysis fails to estimate a memory size when the clock speed drops too much, such as 500 KHz. This occurs when the allocation rate exceeds the collection rate, so theoretically the IHGC cannot 'keep up' with the program.

The real-time analysis produces the lowest memory requirement estimates when more information about the program's behavior is supplied via the c, n, p, s and d parameters. But these parameters are not essential. Figure 7.6(a) illustrates how the estimates increase when the parameters are progressively replaced by their worst-case values. For example, the line labeled c,n,s,d does not include p; implying that every word in deep objects is expected to be a pointer. This worst-case assumption increases the time that the IHGC is expected to need to complete the marking stage. The resulting memory estimate does not increase significantly because the program allocates very few deep objects.

The estimated memory requirements increase by up to 21% when the parameters s and d are not supplied. This large gap occurs because without s and d our analysis has to assume that every object in memory is deep. Therefore, the marking stage will theoretically last longer because the majority of *converter*'s allocated storage space corresponds to packet data that does not contain pointers. Lastly, replacing all the parameters c, n, p, s and d that are not essential to our real-time analysis with their worst-case values yields up to a further 7% larger memory size estimate. Supplying these parameters is a burden for the programmer. Therefore, our analysis technique eases that burden by allowing the parameters to be replaced by their worst-case values at the expense of larger estimated memory sizes.

Figure 7.6(c) shows the distribution and duration of collection cycles as the processor's clock frequency changes. We can make three important observations. First, the duration of t_{init} and t_{roots} is negligible as expected because these are short, constant-time operations that are performed in less than 5 memory cycles. Second, the duration of the IHGC's mark stage lasts 10-15% of the total collection cycle length and is not affected by the duration of a collection cycle. This is because the amount of marking work is proportional to the live size which we can closely characterize using static program analysis techniques. Finally, the IHGC spends 80-85% of its time compacting in the worst-case. Also, longer collection cycles increase the duration of compact because more memory is allocated during the cycle (w) as shown in Figure 7.6(d). A larger w also implies more work during compact because newly allocated objects must be copied, which in turn, yields longer collection cycles. We consider that future research should focus on improving the performance of compact given that it is the most time-consuming operation.

The value of t_f decreases if the processor fetches instructions too often as discussed in Section 7.4.1. Increasing the amount of data fetched simultaneously reduces the memory cycles

required by the processor for fetching, so the estimated memory size decreases as shown in Figure 7.6(b). According to our experiments, fetching 2 words simultaneously instead of 1 decreases the estimated memory size by about 11% when the processor's clock speed is 2 MHz. Also, increasing the number of words fetched from 2 to 4 decreases the estimated memory size by a further 7%. This approach is simple from the programmer's point of view as it does not require changes the program source code. However, increasing the data fetched requires a change in the hardware along with a different cost model to estimate t_f using our automated static analysis tool.

7.5.4 Case Study: Router

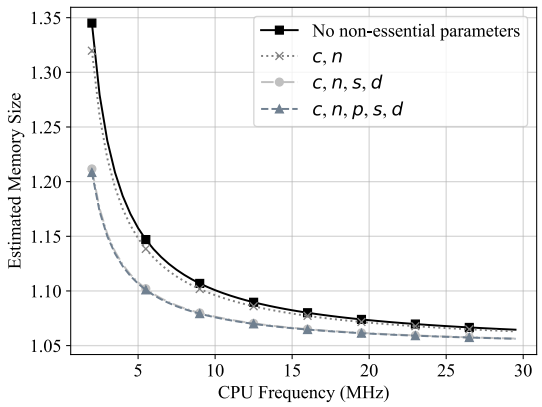
Router is another real-time program developed by us to validate our hard real-time analysis technique. It performs the functions of a simple routing device that receives packets from one input port and transmits them to either one of two ports using an address in the packet header. The input data rate differs from the output rate, so the router must buffer packets until they can be sent. The program must ensure that the input stream is never paused due to the lack of buffer space. So network packets in the internal buffers can be dropped to free up space for incoming packets.

The packet sizes and general system configuration of *router* is similar to *converter*. The packets are of variable size and contain between 64 to 1518 bytes of data. *Router*'s buffers are also 64 KB in size. The input and output ports are operated by serial devices that access the system's memory through Direct Memory Access (DMA). The program uses polling instead of interrupts.

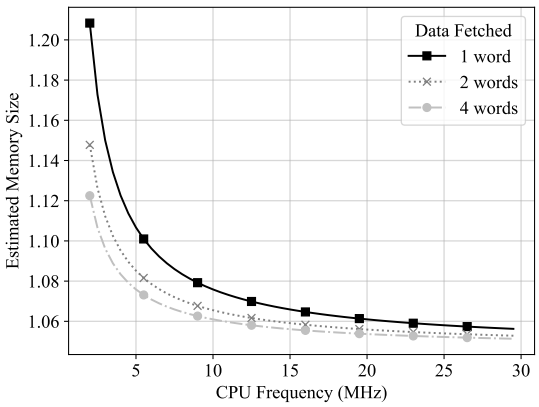
We estimated n, r, p, d, s, z and c by manually inspecting the code. To estimate a and t_f, we identified the operations that the program executes periodically and fed this information to our automated static analysis tool. *Router* performs the following sequence of operations during the execution of each period:

1. Receive the fixed-size packet header containing the packet size.
2. Drop packets from the internal buffer (if necessary) to make space for the incoming data.
3. Receive the remaining packet data.
4. Route the packet to the correct output transmission queue.

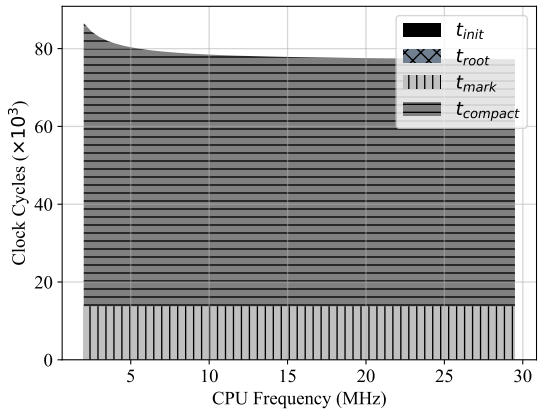
The results of our analysis for *router* are show in Figure 7.7. There is clearly a striking similarity between the shape of the curve for *converter* and *router*. This is expected because the system's memory requirements to avoid pauses increase as the number of spare cycles available to the collector decreases. However, the plots also show a key differences between the two case studies: the duration of a garbage collection cycle is shorter in *router* than *converter*. Comparing Figure 7.7(c) with Figure 7.6(c), it is clear that the difference is due to shorter compact stages in



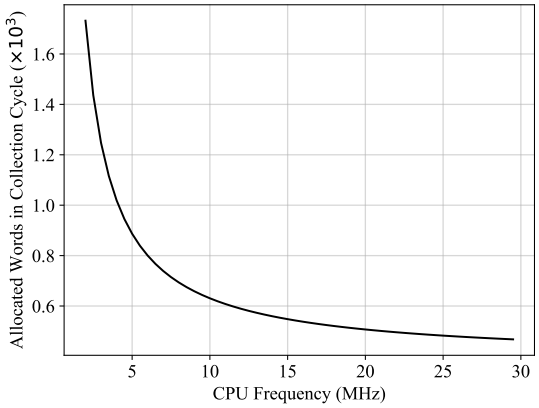
(a) Memory requirements as more information about the program is supplied to the real-time analysis.



(b) Memory requirements as the amount of data loaded per instruction fetch increase.



(c) Duration of a garbage collection cycle.



(d) Memory allocated in a collection cycle (w).

Figure 7.7: Estimated memory requirements and garbage collection cycle duration for *router*. Higher clock speeds increase t_f and yield smaller estimates for the memory size and collection cycle duration. The memory requirements are shown as the ratio between the estimated amount of memory and the program's worst-case live size. It is assumed that the duration of a memory cycle in clock cycles does not change as the clock frequency increases.

router. This occurs because the collector has more spare cycles to operate when *router* is executed compared to *converter*. As a result, there is less memory allocated during a collection cycle (w) and the worst-case memory estimates are also smaller.

7.5.5 Scaling Up the Hard Real-Time Analysis

In this thesis, we applied our real-time analysis technique to programs written in C with up to 500 lines of code. We were limited by the difficulty to find suitable benchmarking software. However, there is no evidence to suggest that the analysis would not scale to larger, more complex programs. Our hard real-time analysis relies on two pillars: the time-predictability of the IHGC and the static analysis of programs. We demonstrated the time-predictability of the IHGC in Chapter 6 when we constructed its timing model. Our static program analysis relies extensively on state-of-the-art real-time analysis techniques normally used for Worst-Case Execution Time (WCET). Therefore, our analysis is expected to scale to other programs as long as the WCET techniques can be applied to them.

Our real-time static analysis technique can also be applied to programs written using *statically typed* modern languages like Java, C# or Go. These languages require the programmer to specify the type of each variable at compile-time. Thus, our automated program analysis tool can easily use the type information to determine what operation the code performs and the amount of time or memory resources that it consumes. In contrast, the type of a variable is only determined at run-time in *dynamically typed* languages, such as Python or JavaScript, so it is difficult to statically analyze programs written in these languages. As a result, the real-time analysis techniques demonstrated in this thesis using C programs are also applicable to software written in statically, although not dynamically, typed modern languages.

7.6 Summary

Through simulation, we experimented with C programs, some of which relied on explicit memory management. We found that the performance of these programs is comparable or better in the IHGC than in a conventional processor. We also observed that when running a Python interpreter, the IHGC's run-time outperforms a software collector by factors of 1.5-7 in several of our benchmarks. Our experiments indicate that large and realistic programs mostly run without pauses when the system's heap size is increased by 20% over the minimum operational requirement. In addition, simple hardware features, such as the amount of data simultaneously loaded per instruction fetch, can be leveraged by the collector to efficiently improve performance without increasing pause times or memory requirements.

We put in practice our hard real-time analysis technique using the TACLe and BEEBS benchmark suites. The results show that our technique predicts the theoretical minimum memory requirements to run these programs without pauses. However, these programs do not use dynamic

memory in realistic settings. So we analyzed two practical programs and explored how the estimated memory requirements change depending on the spare memory cycles available to the IHGC. For example, increasing the processor's instruction fetch rate can reduce the estimated memory size by up to 11%. We demonstrated how our automated static program analysis tool was used to characterize a program's memory cycles and allocation patterns. It was also shown how additional program information, such as the number of pointers and live objects, improves our memory size estimates.

The aim of this experimentation work was to show that the IHGC, a hardware-implemented collector, can deliver real-time guarantees while keeping performance and memory overheads low compared to existing collectors. We consider that this goal was achieved, but several issues remain open:

- There are clear conflicts between some of the IHGC's features and the code generated by off-the-shelf compilers.
- It is unclear how some features of the IHGC, such as the type tags and root marking, can be efficiently realized in the hardware.
- Embedded devices require I/O and the ability to respond to interrupts and exceptions. We must provide further evidence that the IHGC can be used in systems that require these features.

The issues are addressed in Part III of this thesis.

Part III

Architecting a Garbage Collected Embedded System

CHAPTER 8

GARBAGE COLLECTION IN INSTRUCTION SET DESIGN

In Part II, we presented and evaluated the IHGC, a hardware garbage collector that is deeply integrated with the processor. Our focus so far remained in the algorithm's design, efficiency and real-time properties while other problems with integrating the collector into a practical computer system have not been discussed. For example, we found conflicts between the code generated by off-the-shelf compilers and the IHGC. We address these issues by exploring, through compiler design, the impact of the IHGC on the computer's Instruction Set Architecture (ISA).

8.1 Problem Statement

Exact collectors, like the IHGC, guarantee that objects are never reclaimed while they are reachable and that unreachable objects are eventually reclaimed. Collectors use mechanisms like read and write barriers (or mark on load in the IHGC) to ensure that these correctness properties are always met when the user's program accesses memory. But many other, non-memory operations can also violate these properties. For example, adding a value to a pointer without restrictions may result in a pointer to another object that is in the process of being reclaimed. This is clearly erroneous behavior, so reliable garbage collected systems must prevent these failures.

Modern garbage collected languages eliminate the problems outlined above by design. For example, strictly typed languages like Java and C# do not allow arbitrary operations on pointers or type conversions between pointers and integers. The code is checked statically and type errors are detected and reported by the compiler. Similarly, dynamically typed languages, such as Python and JavaScript, have runtime environments that prevent potentially dangerous pointer operations and type conversions at run-time. Therefore, the software collectors used in these systems are relieved from the responsibility of ensuring that pointer operations do not cause

failures. However, an embedded system with the IHGC cannot rely on programming language assumptions. First, runtimes often incur performance, memory and code size overheads that these small devices cannot afford, so the runtime itself or much of its checking capabilities could be dispensed with. Second, the compiler or runtime performing the type checks is not necessarily bug-free; mistakes (or malicious behavior) sometimes occur that could compromise the stability of the system. And third, the vast majority of embedded software is currently written in C/C++ or even assembly. These languages are weakly typed, so programs cannot be trusted to uphold the IHGC's correctness properties. Instead, we solve the problem by exploring changes to the interface between the IHGC and the programmer: the Instruction Set Architecture (ISA).

Changes for the ISA to work alongside the IHGC must carefully balance two conflicting goals: safety and compatibility. An ISA can leverage the IHGC's rich feature set to boost safety by enforcing strict run-time checks on operations that use pointers. For example, the ISA could prevent pointers to out-of-bounds locations from being constructed instead of simply checking out-of-bounds accesses when loading or storing to memory. These strict checks facilitate error detection and containment, but they limit compatibility with existing software and compilers that do not uphold such safety requirements. For instance, embedded C/C++ programs compiled with LLVM or GCC often perform pointer operations that would be considered unsafe by the ISA, so the software would no longer work. Therefore, we must ensure that ISA changes strike a balance between safety and compatibility with existing software, but as a minimum, guarantee that the stability of the IHGC system is never compromised regardless of software bugs (or malicious intent).

Previous hardware garbage collection research often focuses on the algorithm's design and its efficiency. But the practical considerations of integrating the collector with a computer architecture are often neglected. For example, it is unclear how arbitrary pointer operations that compromise the collector's correctness are handled or how collectors interact with interrupt handlers and I/O devices. It is our view that these challenges must be addressed to build a truly practical embedded system with hardware garbage collection. Thus, our goal in this chapter is to identify and resolve the issues arising from the integration of the IHGC with ISAs. Our main concern is to guarantee the collector's correctness and reliability, even in the presence of buggy or malicious software, while ensuring that the ISA maintains a high degree of compatibility with existing software and compilers.

8.2 Architectural Challenges

Programmers usually write code in a language that eventually gets translated into machine instructions by a compiler. The compiler is the direct user of an ISA; it encodes the assumptions and intentions of the ISA designers. So it is important that compilers can generate efficient code using an ISA. In addition, implementing a compiler is a good strategy to ensure that the

ISA is compatible with the programming languages supported by the compiler. As a result, we investigated the integration challenges between the ISA and IHGC through compiler design.

In this section, we explain the challenges in designing an ISA for the IHGC. We do so from the perspective of an existing architecture, ARMv6-M, and a new architecture, BeyondRISC, proposed by Andrés Amaya García and David May. We use code examples and assembly emitted by the LLVM version 9 compiler to illustrate the issues encountered and our proposed solutions. We also outline our experience while extending LLVM with a new backend for the BeyondRISC architecture.

8.2.1 Background

We briefly describe basic features from the architectures discussed in the remaining of this chapter.

8.2.1.1 The ARMv6-M Architecture

The ARMv6-M architecture is the microcontroller profile of the ARMv6 revision [20]. ARMv6-M supports the Thumb instruction set which consists mostly of 16-bit and a few 32-bit instructions. There are 16 registers each with 32 bits. However, the majority of instructions can only access the first eight (*r0-r7*), or *low*, registers. A small number of instructions can also access the remaining eight, or *high*, registers which include a program counter (*pc*), a link register (*lr*) and the stack pointer (*sp*). In addition, many 16-bit Thumb instructions have an implicit register operand due to the lack of space in the encoding.

Conditional execution is a feature of some Thumb branch instructions; this is also known as *predicated execution*. The branches are only taken if the bits in a status register satisfy the condition specified by the branch instruction; otherwise, the branch is not taken effectively becoming a `nop`. Most Thumb instructions modify the status register.

8.2.1.2 The BeyondRISC Architecture

BeyondRISC is a computer architecture for embedded devices that uses variable 16-bit or 32-bit instructions. The register file has 10 general-purpose 32-bit registers along with a program counter (*pc*), the stack pointer (*sp*), an environment pointer (*ep*) and a status register (*sr*). The *ep* is used to facilitate access to global data structures. The *sr* contains configuration flags, such as for floating-point arithmetic; BeyondRISC does not rely on predicated execution. Most instructions access the general-purpose registers while only a few instructions, like stack operations, can access the special-purpose registers.

BeyondRISC is an experimental RISC architecture that we developed from scratch. It was specially designed to be implemented in systems that also have the IHGC. We use BeyondRISC in this thesis to illustrate how an ISA can be designed from scratch with garbage collection in

mind. Therefore, it is an important vehicle to explain the challenges and solutions to integrating the IHGC with novel ISAs. We also use BeyondRISC to discuss the practical implications of extending existing compilers to support ISAs with garbage collection.

8.2.2 Operations on Pointer and Value Types

Exact garbage collectors must distinguish pointer from data values without fault. The IHGC achieves this by using an extra tag bit for every word in memory and the registers. Therefore, any ISA that operates alongside the IHGC must be adapted to incorporate these two types. In our experience from Chapter 7, these requirements cause recurrent problems with existing ISAs because there is no concept of types when specifying the instruction operands. For example, the ARMv6-M architecture does not mandate that the two input operands for an `add` instruction cannot both be pointers even though the operation is meaningless. Such operations violate the IHGC's correctness properties as they can generate pointers to arbitrary objects.

The ARMv6-M architecture actually includes very few instructions specifically designed for address generation. In fact, arithmetic instructions are also often used to perform pointer arithmetic on the low registers while most address generation instructions use an implicit high register as the base pointer. Therefore, the obvious solution to support the IHGC alongside the ARMv6-M architecture is to add the extra tag bit to every word in memory and the registers and to change the semantics of the existing instructions to restrict the allowed operand types. The instructions are changed as follows:

- Most data processing operations, like multiplication and byte reverse, raise an exception when any of the supplied operands is a pointer.
- Memory access operations, such as loads and stores, check that the base address operand is a pointer and the offset (if any) is a data word.
- The operations performed by addition and subtraction instructions depend on the types of the operands. For example, adding a data word to a pointer results in a pointer to the same object with potentially a different offset, even if overflow occurs, but adding two pointers gives rise to an exception.

The advantage of this approach is that the architecture does not require new instructions and compatibility with existing software is largely maintained. The drawback is that compilers need to be made aware of the semantic changes, so programs need to be recompiled to run on the modified architecture. In addition, some existing software will inevitably cease to execute correctly as it performs restricted operations on pointers. For instance, MicroPython's parser encodes the type of a word using the least significant two bits [115]. The bitwise operations shown in Listing 8.1 are often used on arbitrary words to determine whether they contain word-aligned pointers or data. Fortunately, the majority of programs will be unaffected by semantic changes

```
#define MP_PARSE_NODE_NULL               (0)

#define MP_PARSE_NODE_IS_NULL(pn)        ((pn) == MP_PARSE_NODE_NULL)
#define MP_PARSE_NODE_IS_LEAF(pn)        ((pn) & 3)
#define MP_PARSE_NODE_IS_STRUCT_KIND(pn) (... && ((pn) & 3) == 0 && ...)

if (MP_PARSE_NODE_IS_NULL(*pn)) {
    // Empty node
} else if (MP_PARSE_NODE_IS_LEAF(*pn)) {
    // Node does not contain a pointer
} else {
    // Type cast to pointer.
    mp_parse_node_struct_t *pns = (mp_parse_node_struct_t*)(*pn);
    if (MP_PARSE_NODE_STRUCT_KIND(pns) != pn_kind) {
        ...
    }
}
```

Listing 8.1: Code fragment from the MicroPython `parse.h` and `parse.c` files performing bitwise operations on arbitrary words [115]. Type conversions between pointer and integer are also executed.

to instructions because arithmetic and bitwise operations on pointers do not occur in modern languages like Java, Python or C#. Moreover, the C standard mandates that operands to bitwise and arithmetic operations, with the exception of addition and subtraction, must not be pointers and conversion between pointer and integer types is discouraged [81].

Another problem with allowing instructions to accept both pointers and data words is *conditional behavior*. That is, the instruction's operation becomes conditional upon the operand's types which increases the complexity of the hardware implementation. This is especially problematic when attempting to leverage the IHGC's data to perform safety checks alongside pointer arithmetic. An alternative to mitigate this issue is to segregate the instructions into arithmetic and address calculation at the expense of instruction encoding space. For example, an `add` performs the regular addition of two integers and an instruction `ldaw` adds a scaled offset to a base pointer; `add` and `ldaw` both raise exceptions if operands of incorrect type are supplied. Additionally, `ldaw` could raise an exception if the resulting pointer is outside the object's bounds. Segregating arithmetic and pointer arithmetic instructions in this fashion is not a new idea; existing ISAs already implement similar schemes, such as the XMOS XS1 architecture [107]. We reuse this idea in BeyondRISC to cleanly differentiate types and safeguard the IHGC's correctness properties.

8.2.2.1 Pointer and Value Types in LLVM

LLVM is a widely used general-purpose compiler framework to translate code from C, C++ and other programming languages into machine code. The compilation process has three major stages. First, the source code is parsed and translated into the LLVM Intermediate Representation (IR). This is a Single Static Assignment (SSA) assembly language that retains type information from

the original code. The IR is low-level and flexible enough to allow applying a range of optimizations during the second stage. Finally, the instruction selection stage reads in the optimized IR and emits the machine code for a specific architecture.

The instruction selection stage is implemented by a *backend* that is different for every computer architecture. For example, there are different backends for ARM, x86, RISC V and we added a new one for BeyondRISC. The LLVM backend framework takes the IR as an input and progressively lowers the SSA instructions into actual machine instructions. However, the process discards the majority of the operand type information early on. For example, the backend internally converts a string pointer in the original LLVM IR to a plain 32-bit integer when compiling code for a 32-bit machine. This causes difficulty in backends that generate code for an IHGC system because it is always unclear what instruction is appropriate for the given operands. To maximize compatibility with existing compilers and software, we mitigate these problems in both the ARMv6-M and BeyondRISC architectures by allowing a small set of arithmetic instructions to operate on both pointers and data:

Bitwise-or. Performs the bitwise-or of two words and produces an integer regardless of the input operand types.

Unsigned less than. The input operands are treated as unsigned integers and compared.

Add, subtract. The input operands are added or subtracted if they are not pointers. If one of the operands is a pointer, the integer operand is added or subtracted from the base pointer's offset; the pointer handle remains unchanged. Adding two pointers gives rise to an exception. Subtraction of two pointers is also allowed; the pointers are treated as unsigned integers and the result is a data value. Subtraction of two pointers is a commonly used operation in programming languages, like C, to compute the distance between two pointers.

Another problem with the type occurs when branching on a condition. Programming languages like C specify a `NULL` pointer as the integer value 0 cast to a pointer type. Compilers take advantage of this to optimize comparisons against the `NULL` pointer within conditional expressions. For example, the compiler can directly branch using a pointer as a boolean value as shown in Figure 8.1. The compiler can also use bitwise-or instructions on pointer operands to derive a boolean value for the branch as it occurs in Figure 8.2. To efficiently support these operations, we simply need to allow the comparison instruction (`cmp`) to operate on pointers in the ARMv6-M architecture. In contrast, BeyondRISC does not use predicated execution, so we must instead allow conditional branches using pointers as boolean values. The `NULL` value evaluates to `false` and a pointer evaluates to `true`.

```
1 unsigned int *a;
2
3 if (a) { ... }
```

(a) Conditional if-statement.

```
1 brTrue $a, .label
2
3
```

(b) Assembly.

Figure 8.1: Compiler generated machine code using a pointer as a boolean operand for a conditional branch instruction. A `NULL` pointer results in the branch not taken.

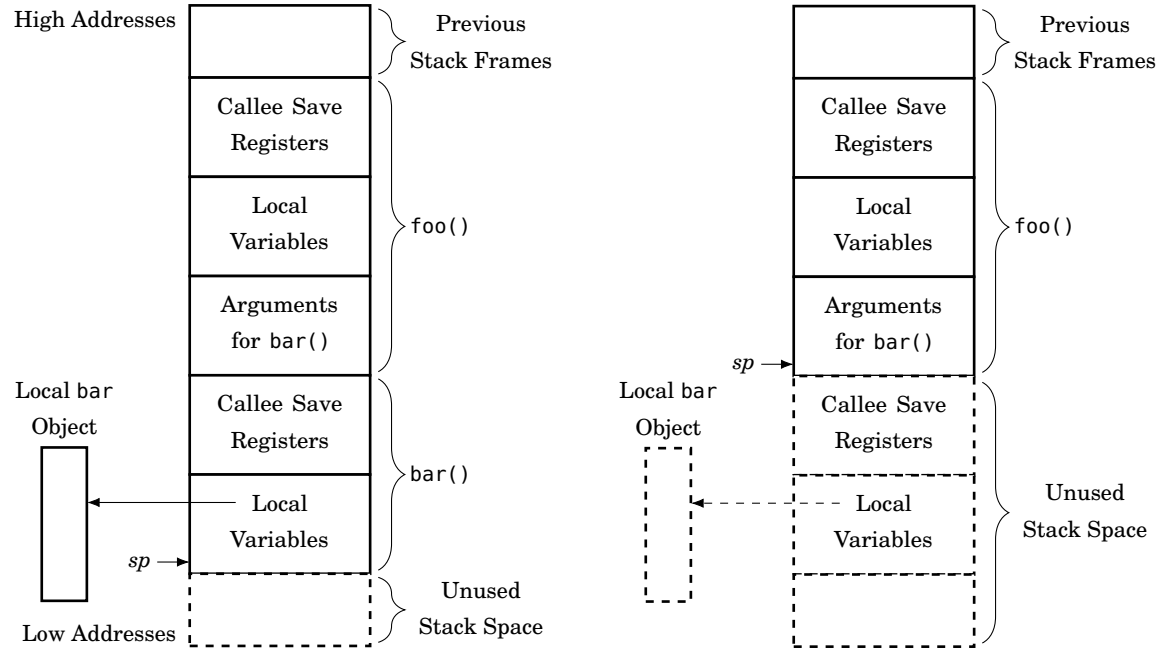
```
1 unsigned int *a, *b;
2
3 if (a == NULL && b == NULL) { ... }
```

(a) Conditional if-statement.

```
1 or      $isNull, $a, $b
2 brFalse $isNull, .label
3
```

(b) Assembly.

Figure 8.2: Compiler generated machine code using a bitwise-or instruction to produce a boolean value from pointer operands. The compiler takes advantage of `NULL` being standardized to the value zero to optimize the conditional check.



(a) Stack layout after a function call to `bar` from `foo`.

(b) Stack layout after returning from `bar` to `foo`.

Figure 8.3: Layout of a stack implementing the ARM-THUMB Procedure Call Standard (ATPCS) when `foo` calls another function `bar` with more than four arguments. The local object dynamically allocated by `bar` will not be reclaimed by the IHGC after returning to `foo`.

8.2.3 Function Call Stack

The function call stack is perhaps the most complex data structure managed by the garbage collector. There are two main problems caused by the inherently dynamic nature of the stack. First, it is difficult to distinguish pointers from data words in the stack. The contents of the stack are constantly changing as functions are called and exited, so the type of the words in the stack is also in constant flux. Multiple schemes have been proposed to address the problem, but the solutions often incur high overheads or degrade the system's real-time properties. For example, Henriksson's collector uses two stacks; the user program's stack and a supplementary stack with references to the locations in the first stack that contain pointers [79]. Pushing and popping pointers on the stack requires operating two data structures and at least double the space for each pointer. An alternative proposed by Gruian and Salcic is to conservatively scan and mark the stack [74]. But their collector is no longer exact making it difficult to analyze its real-time properties. The IHGC does not suffer from these problems because it relies on tag bits to distinguish pointers from data.

The second problem is the difficulty in determining when the stack must be scanned during marking. Many existing collectors (especially those implemented in software) consider the stack as part of the roots, but root marking is often not an incremental operation. Therefore, root scanning may introduce long pauses when the stack is large. The IHGC eliminates these pauses because the stack is not included in the root set of pointers. The stack is treated as a regular object that is marked and compacted incrementally. However, our experiments from Chapter 7 showed that this approach to managing the stack caused the IHGC to unnecessarily retain garbage objects. This problem occurred because our simulated system uses a *contiguous* stack layout. In the remaining of this section, we describe this issue and outline solutions.

8.2.3.1 Contiguous Stack

The ARMv6-M architecture relies on the ARM-THUMB Procedure Call Standard (ATPCS) [17]. The standard describes the mechanism used to implement function calls including: input arguments, return values and the layout of the function call stack. ATPCS mandates a *contiguous* stack meaning that each thread's stack is allocated as a contiguous block of memory. Also, the stack is *full descending*, so the *sp* register references the last value pushed and the stack grows from high to low addresses. Before performing a function call, the caller loads the first four arguments onto *r0-r3* and any other arguments are pushed onto the stack. The callee can modify the argument registers, but it must ensure that the values in any other *callee save registers* are unchanged when returning to the caller. So on entry to a function, the callee decrements the *sp* to create a new stack frame for local variables and saves the registers whose value needs to be restored before returning.

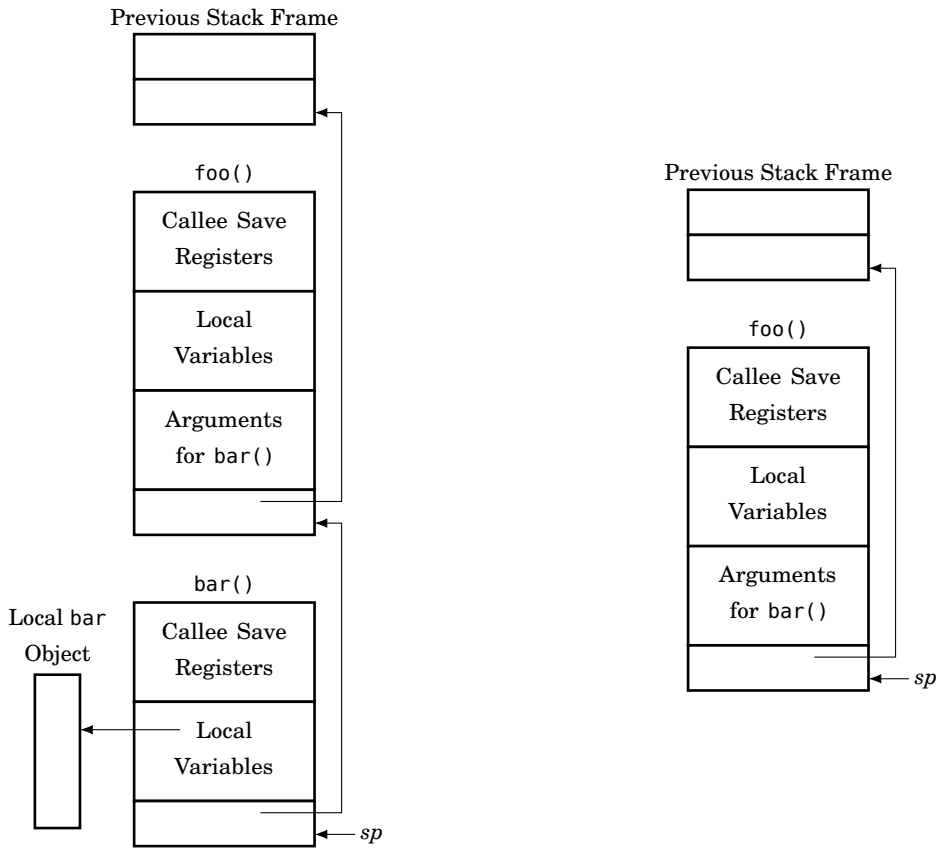
An example of an ATPCS stack after two function calls is shown in Figure 8.3(a). The stack frame at the high addresses belongs to the function `foo` and the following frame is from `bar`.

Stack space has to be reserved for argument passing because `bar` has more than four arguments. In addition, copies of the callee save registers are also pushed to the stack upon entry to `bar` so that these can be restored before returning. In principle, ATPCS and contiguous stacks can be used alongside the IHGC as long as the stack is allocated as a single object. However, this can result in programs with unintentionally higher memory requirements as observed in Chapter 7. The problem occurs because the IHGC marks and scans the stack as any other heap object. So objects referenced from stale stack frames below the *sp* will be considered as live even though they are garbage. For instance, the local object allocated by `bar` is referenced from a stack location as shown in Figure 8.3(b). The object becomes garbage when `bar` returns, but the pointer, which is now below the *sp*, is still reachable from the IHGC's point of view. Therefore, the garbage is unnecessarily retained increasing memory requirements.

Hardware garbage collectors are often used alongside contiguous function call stacks [112, 113, 167]. These collectors manage the stack differently from other heap objects to ensure that stale stack frames are not scanned during marking. For example, the collector only scans the function call stack up to the *sp* and simply ignores, i.e. does not scan, stale stack frames. But compared to the IHGC, this approach has two drawbacks. First, treating certain objects, like the stack, differently increases hardware complexity because the collector's operation becomes dependent on the object's type. And second, the memory locations in stale stack frames are not zeroed after the *sp* is adjusted. In practice, the program can easily access the old values, including pointers, in stale stack frames, so the system would reach an inconsistent state if the IHGC only marks the stack up to the *sp*. This is because objects referenced from stale stack frames would be reclaimed although technically they are still reachable. Clearly, programs that mistakenly or maliciously use the pointers stored in stale stack frames could compromise the collector. In summary, the IHGC does not manage contiguous stacks differently as this makes it difficult to efficiently guarantee the system's stability and reliability.

8.2.3.2 Linked Stack

Contiguous stacks can be operated efficiently, but large memory blocks must be preallocated for each stack although the memory may not ultimately be needed. Also, the stack overflows when the amount of data pushed onto it exceeds its capacity; this is a common source of bugs in embedded devices. For these reasons, computer systems have previously relied on *linked* stacks instead [68, 96]. With this approach, each frame is dynamically allocated individually upon entering a function. The frames are chained using pointers such that the stack forms a singly-linked list with the *sp* always referencing the top frame as shown in Figure 8.4(a). Linked stacks can be used in BeyondRISC procedure calls to ensure that the IHGC never marks and scans stale frames. This is because, as illustrated in Figure 8.4(b), stack frames become unreachable as soon as the function returns, so the IHGC can reclaim the frame object without delay along with any unused local object referenced from that stack frame only. As a result, the stack is a truly



(a) Stack layout after a function call to `bar` from `foo`.

(b) Stack layout after returning from `bar` to `foo`.

Figure 8.4: Stack layout using linked stack frames after a function `foo` calls `bar`. The stack is allocated as a disjoint list of stack frames. `bar`'s stack frame is no longer reachable when the function returns. Therefore, the collector reclaims dynamically allocated objects referenced from local variables in the callee's stack frame.

dynamic data structure that only uses as much space as necessary, can be managed as any other heap object and does not compromise the system's stability.

A common criticism of linked stacks is the performance overhead of a dynamic allocation on function entry and freeing before returning. However, the overhead of allocating an object in the IHGC is negligible as the operation normally lasts about one memory cycle only. Also, the cost of freeing the stack frame object is eliminated because the collector automatically takes care of this in the background. Therefore, we expect the performance of linked stacks on a system with the IHGC to be comparable to ATPCS's contiguous stacks.

A disadvantage of using linked stacks alongside an existing architecture with the IHGC is the need for new instructions to efficiently operate the stack. For example, the ARMv6-M architecture must be extended with instructions to quickly allocate, enter and exit a stack frame. In addition, linked stacks can cause high volumes of stack frame allocations in some programs. For instance,

```
1 int fac(int n) {
2     if (n == 0)
3         return 1;
4     else
5         return (n * fac(n - 1));
6 }
```

Listing 8.2: Code fragment from BEEBS implementing factorial recursively [130].

```
1 int fib(int i) {
2     if (i == 0) return 1;
3     if (i == 1) return 1;
4
5     return fib(i - 1) + fib(i - 2);
6 }
```

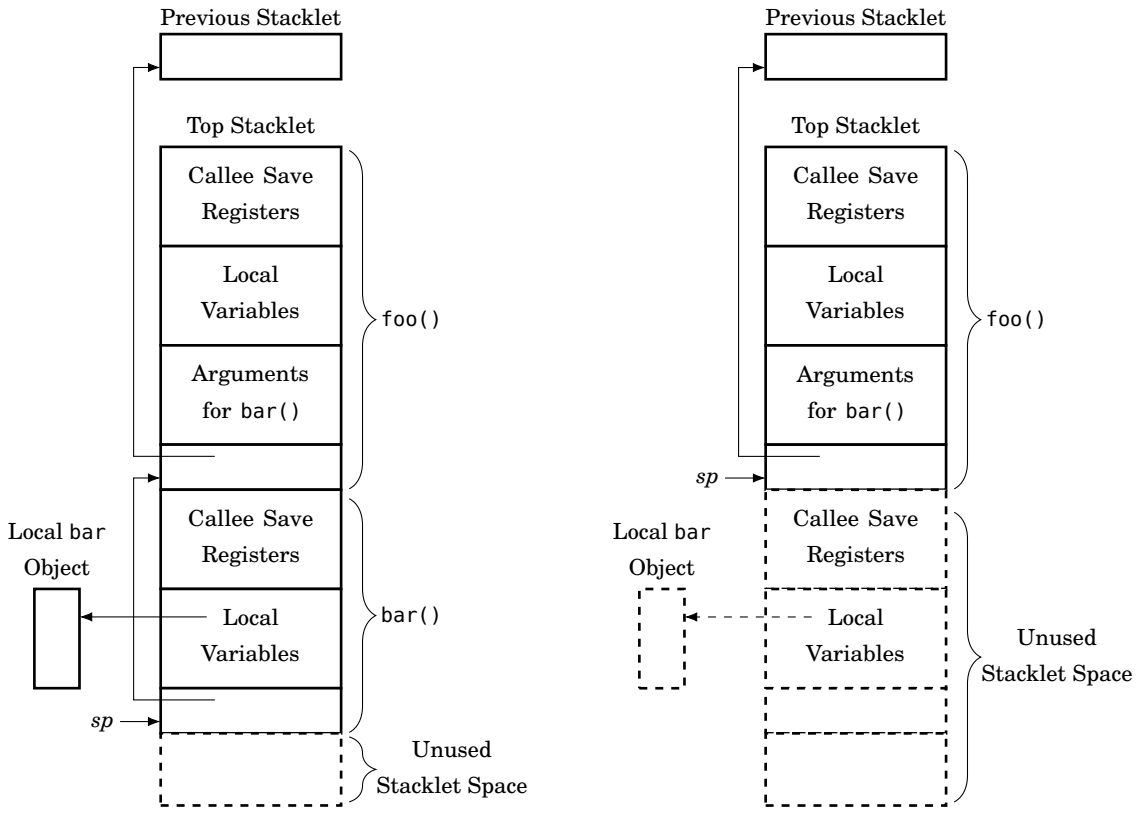
Listing 8.3: Code fragment from BEEBS implementing Fibonacci recursively [130].

the recursive functions in Listing 8.2 and Listing 8.3 from the BEEBS benchmarks allocate very deep call stacks in a short time [130]. These programs put excessive pressure on the collector and can result in occasional pauses. However, extreme recursive behavior is unusual in embedded systems because programmers can often avoid it with little effort and compilers can minimize stack frame allocations using optimizations like inlining.

8.2.3.3 Stacklets

Linked stacks are extremely flexible, but they incur high performance overheads in conventional computer architectures because the program must execute in software the time-consuming, dynamic allocation and free routines for each stack frame. This motivated researchers in parallel computing to propose implementing stacks as chains of *stacklets* [69, 70]. Each stacklet, also called a *segment* [64], is a contiguous block of memory that can accommodate several stack frames. When calling a function, the program checks whether there is sufficient space in the current top stacklet, i.e. the object referenced by the *sp*, to store the callee's stack frame. The check is simple as it only involves comparing the *sp* with the size of the top stacklet. If there is sufficient space, then the *sp* is simply decremented to make room for the callee's stack frame in the top stacklet as shown in Figure 8.5(a). Otherwise, a new stacklet is dynamically allocated to place the callee's stack frame as depicted in Figure 8.6(a). The new stacklet becomes the top stacklet and it is linked to the existing chain of stacklets using pointers.

Stacklets are a compromise between contiguous and linked stacks. Compared to contiguous stacks, stacklets prevent local garbage objects referenced by stale stack frames from remaining in memory for a long time. This is because each stacklet is relatively small, so they frequently become unreachable objects that can be reclaimed as the program returns from functions. For example, Figure 8.6(b) shows that the stacklet containing `bar`'s stack frame becomes unreachable as soon as `bar` returns since the top stacklet is popped off. Therefore, local garbage objects referenced from `bar`'s frame can also be reclaimed. Compared to linked stacks, stacklets reduce the volume of dynamic allocations and mitigate the impact of recursion because a single stacklet contains stack frames from multiple function calls. But unlike linked stacks, stacklets occasionally



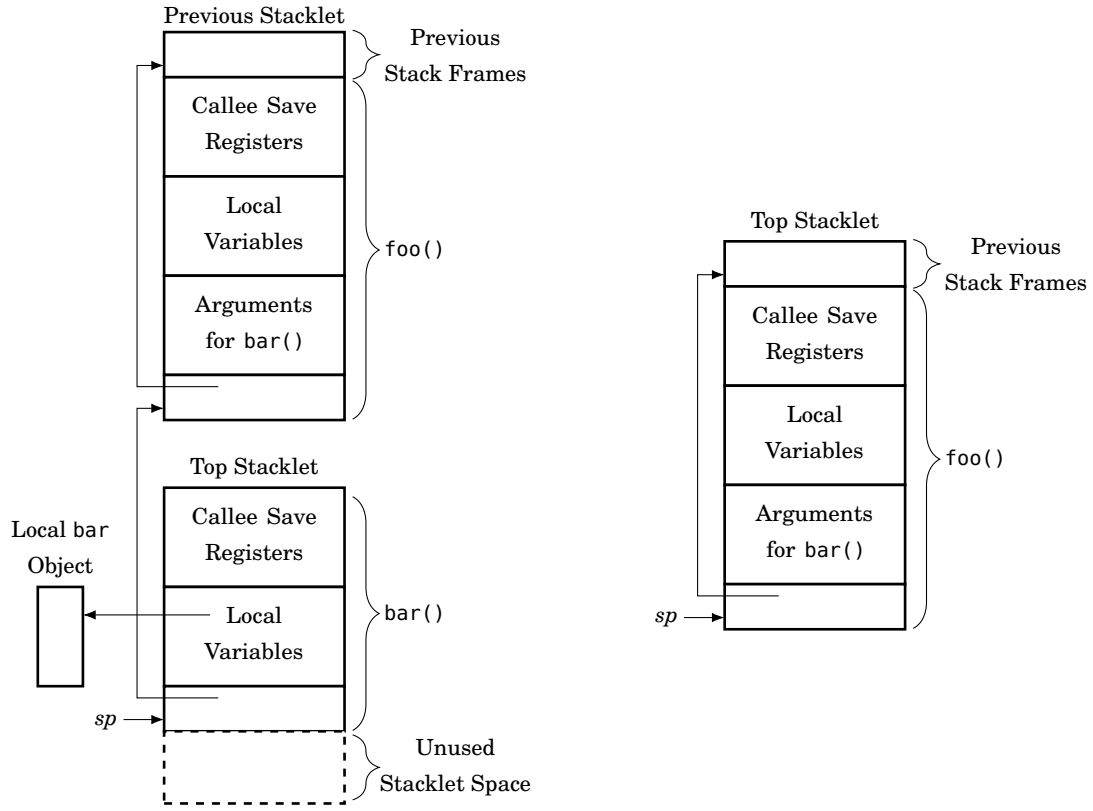
(a) Stack layout after a function call to `bar` from `foo`.

(b) Stack layout after returning from `bar` to `foo`.

Figure 8.5: Stack layout using stacklets after a function `foo` calls `bar`. The current top stacklet has enough space to accommodate `bar`'s stack frame, so a new stacklet is not allocated. The local object will not be reclaimed by the collector until the pointer in `bar`'s stale stack frame is overwritten or the current top stacklet is exited.

retain garbage for longer than necessary as unused local objects can be referenced from stale stack frames in the top stacklet. For instance, Figure 8.5(b) shows that `bar`'s stack frame, and the local object that it references, remain reachable from the IHGC's point of view when `bar` returns. This is because the top stacklet is not popped off the stack as it also contains `foo`'s stack frame, i.e. `bar`'s caller. Increasing the size of each stacklet decreases performance costs as less dynamic memory allocations are performed. However, larger stacklets also increase the likelihood that local garbage objects are unnecessarily retained for longer as stacklets are popped off less frequently.

The IHGC facilitates implementing stacklets because the size of the top stacklet is always known: it is in the directory. So an instruction can be added to ARMv6-M or BeyondRISC to easily check whether a stack frame fits within the current top stacklet or a new stacklet must be dynamically allocated. To achieve this, the instruction simply compares the size of the callee's



(a) Stack layout after a function call to `bar` from `foo`.

(b) Stack layout after returning from `bar` to `foo`.

Figure 8.6: Stack layout using stacklets after a function `foo` calls `bar`. A new stacklet is dynamically allocated at the start of `bar` because the function's stack frame does not fit within the previous top stacklet. The collector reclaims the new top stacklet and the locally referenced object when `bar` returns.

stack frame with the size of the top stacklet minus the *sp* offset. Therefore, stacklets can be efficiently implemented alongside the IHGC.

8.2.4 Exception and Interrupt Handling

The ARMv6-M architecture handles exceptions and interrupts using hardware that implements the ATPCS. When an exception or interrupt occurs, the hardware automatically pushes onto the function call stack the current context, that is, the registers that are not restored by callees before returning. These include the function argument registers (*r0-r3*), *lr* and *pc*. Execution then jumps to a preconfigured handler for the exception or interrupt. Therefore, the handler can be implemented as an ATPCS-compliant function call which performs the usual stack manipulations on entry and before returning. However, this mechanism has the same drawbacks of ATPCS. It relies on contiguous function call stacks, so garbage memory can be unnecessarily retained

for longer than expected increasing memory requirements as explained in Section 8.2.3.1. As a result, we consider architectural changes to eliminate ATPCS's reliance on contiguous stacks when entering interrupt and exception handlers.

A naive solution is to modify the hardware so that it automatically allocates an object and stores the register context in that object when entering an interrupt or exception handler. But, unless we always use the hard real-time analysis from Chapter 6, dynamic allocations can result in collection pauses, so there could be a delay in entering the handler function. This is unacceptable as interrupt requests are extremely time-critical. Allocating when an exception arises can also result in system failure. For example, an allocation failure could have given rise to the exception due to an out-of-memory condition. Therefore, relying on dynamic allocations to enter an exception or interrupt handler is not ideal.

A better alternative to eliminate ATPCS's reliance on contiguous stacks is using *banked registers* to store the register context when interrupts and exceptions occur. The processor has multiple copies of the registers, but only the copy indicated by the execution mode is visible. Thus the BeyondRISC architecture has three modes: `NORMAL`, `EXCEPTION` and `INTERRUPT`. The user's program is executed in `NORMAL` mode. The mode is changed to `EXCEPTION` when an instruction gives rise to an exception, such as division by 0. The `EXCEPTION` register set is then used to execute the exception handler. `INTERRUPT` mode is entered and its corresponding register set is used when an interrupt occurs while the processor is in either `NORMAL` or `EXCEPTION` mode.

Previous architectures implement similar approaches to interrupt and exception handling. Examples include the Atlas computer [99], earlier ARM revisions that implement Fast Interrupts (FIQ) [16, 18], and, to a limited extent, the ARMv6-M architecture which banks the *sp* [20]. The main benefit of banked registers is that exception and interrupt handling does not rely on contiguous stacks and collection pauses never occur when entering handlers as allocations are not performed at that time. The handler could still introduce pauses if it uses linked stacks or stacklets, but the hardware does not cause involuntary pauses when entering the handler. Also, entering and exiting interrupt handlers is very fast because no registers have to be saved or restored from memory. The tradeoff is that BeyondRISC cannot nest interrupt handlers which is desirable in applications where there are complex priority relationships between multiple interrupt sources. Another problem is that it is difficult to extend existing architectures, such as ARMv6-M, to use a new exception and interrupt model.

8.2.5 I/O Devices

The ARMv6-M architecture relies on Memory-Mapped I/O (MMIO) for the processor to interface with peripheral devices. MMIO associates the registers and memory of I/O devices with addresses in memory. Normally, the addresses mapped to I/O devices are documented in the processor's reference manual and are hard-coded in software. Accessing an I/O device involves casting a hard-coded integer address into a pointer. Then the regular memory access instructions can be

used to interface with the peripheral. Clearly, this is problematic for the IHGC whose strict type system does not allow casting values from integer to pointer, so we cannot rely on the traditional MMIO approach.

We provide access to I/O devices using *Object-Mapped I/O* (OMIO). The hardware maps each device to an object handle at reset. The handles are then used to construct pointers that are made available to the processor before the system boots. For example, a system with three devices would allocate that same number of object handles at reset. The handles are then used to construct pointers that are provided to the program via the architectural registers or a preallocated array in memory. The I/O devices can be accessed using these pointers along with the regular memory load and store instructions. As a result, type conversions are no longer required, so OMIO can be used with the IHGC.

An important benefit of OMIO is that the registers and memory for I/O devices do not need to be considered as part of the roots. The objects corresponding to mapped I/O devices will simply be marked and scanned as if they were memory locations. Thus, the IHGC discovers objects referenced from I/O registers, like a pointer to an array stored in a DMA register, without programmer intervention. OMIO also facilitates controlling access to I/O devices because processes cannot access the devices's registers and memory if they do not have the relevant pointers. Normally, access to the pointers is controlled by an Operating System (OS) which shares the pointers with individual processes depending on the configured permissions.

In our experience, OMIO can be easily integrated into BeyondRISC and the ARMv6-M architecture. The only drawback is that some existing system software will no longer operate correctly because they use MMIO. Nevertheless, modern embedded software normally relies on a Hardware Abstraction Layer (HAL) library that provides a high-level interface over the I/O device drivers. The HAL and the drivers themselves are generally developed and maintained by the hardware vendor. This facilitates software development because most programmers do not need to be aware of the platform's configuration details and can instead focus on writing their application. This model is also beneficial from our perspective because migrating the software from MMIO to OMIO only requires self-contained changes to the HAL.

8.2.6 Linking Programs Statically

In IHGC systems, the program's code is contained in objects that receive the same collection treatment during marking as any other object. That is, the objects are marked and later scanned in search for pointers if their deep flag is set as described in Chapter 5. The IHGC does not scan code objects during marking if they do not contain pointers. So it is ideal to eliminate pointers from code objects because this reduces the amount of collection work performed to complete a collection cycle.

Most embedded software is *statically linked* meaning that all the procedures and variables used by the program can be resolved at compile-time and are included in the executable. In

(a) Global variables grouped into a section.

(b) Global variables organized using an array.

Figure 8.7: Existing compilers group global variables into a contiguous chunk of memory often called a section, such as `data` or `bss`. But in the IHGC, it is ideal to allocate an array of globals. Variables of scalar type, such as integers A and C, are stored directly in the array. Variables of complex types, like strings B and D, are allocated as independent objects referenced from the array of globals.

this case, compilers often include pointers alongside code to improve performance and reduce code size. For example, this technique is widely used by the LLVM backend for the ARMv6-M architecture because the Thumb instructions have very short immediate ranges that make it difficult to efficiently perform address calculations relative to a base pointer, like the *pc*. But the pointers introduced in code objects by static linking strategies unnecessarily increase collection work. Fortunately, avoiding this problem using LLVM is simple as the compiler can be configured to generate code using a *position-independent* strategy for both the BeyondRISC and ARMv6-M architectures. Position-independent compilation does not place pointers within code objects and does not incur performance penalties in BeyondRISC, but it does when emitting Thumb instructions as explained before.

Another consideration when statically linking programs for an IHGC system is the organization of global variables into objects. Traditional linkers group global variables into sections, such as `data` and `bss`, that are loaded into memory before the program starts executing. Each section is a contiguous chunk of memory that the program indexes to access the individual variables as shown in Figure 8.7(a). However, this arrangement does not take advantage of the IHGC's features. The memory access bounds of each object would not be automatically checked by the hardware, so errors are not easily detected. Also, the aggregate size of all global objects in large programs is likely to exceed the addressable memory by a pointer offset. To solve these issues, we instead organize global objects using an array as shown in Figure 8.7(b). Each global variable of scalar type, like an integer, is stored directly into the array and can be accessed by simple indexing. Global variables of more complex data types, like strings, are allocated as separate objects that are referenced from the array. Thus, accessing these objects requires following an indirection through the array of globals.

Our proposed solution for organizing the globals can be efficiently implemented using both BeyondRISC and ARMv6-M architectures. In BeyondRISC, the array of globals is referenced by a special-purpose *ep* register. The global variables are accessed efficiently by indexing the array of

```
#define free(x)
#define calloc(n, e)    malloc((n) * (e))

__attribute__((naked)) void *malloc(size_t n) {
    __asm volatile (
        "newm       r0, r0    \n"
        "ret                  \n"
    );
}
```

Listing 8.4: `malloc`, `calloc` and `free` implementation in an architecture with the IHGC.

globals using instructions that have the *ep* as an implicit operand and an explicit offset operand. A similar strategy can also be used in the ARMv6-M architecture although this might require a few new instructions to reduce overheads.

8.3 Case Studies

We evaluated the impact of our proposed architectural changes on existing programs. We compiled the benchmarks from Chapter 7 using our modified LLVM compiler and ran them on a functional simulator for BeyondRISC. Also, we ported popular open-source software libraries to our platform. Through this experience, we outline the steps required to port existing programs to an architecture with the IHGC and assess the difficulties encountered. We are particularly interested in identifying concrete code fragments that conflict with the IHGC and how these compatibility problems can be addressed.

8.3.1 BEEBS and TACLe Benchmark Suites

As described in Section 7.2, the BEEBS and TACLe benchmark suites are largely composed of small, self-contained programs running classic algorithms like quicksort. These benchmarks are entirely written in C and occasionally rely on calls to the standard library explicit memory management functions `malloc`, `calloc`, and `free`. So our porting work was limited to replacing the implementation of these functions at either the linking or preprocessing stages. As shown in Listing 8.4, the BeyondRISC `malloc` consists of a `newm` and procedure return instructions. `calloc` is replaced by a call to `malloc` using the C preprocessor while calls to `free` are simply eliminated. `calloc` no longer requires zeroing the allocated memory because this is automatically performed in the background by the IHGC. No other changes to the existing benchmark source code were required to compile and run the programs successfully.

Replacing the `malloc`, `calloc` and `free` functions with an IHGC implementation has three main benefits. First, the C standard library memory management interface is no longer explicit. Calls to `free` are not necessary because the IHGC automatically reclaims the memory. Second, the run-time of memory management operations decreases significantly when compared to

```
1  void *mempcpy(void *dst0, const void *src0, size_t len0) {
2      char *dst = (char *)dst0;
3      char *src = (char *)src0;
4
5      while (len0--) {
6          *dst++ = *src++;
7      }
8
9      return dst;
10 }
```

Listing 8.5: Implementation of the C standard library function `memcpy` from Newlib [138].

implementations like dlmalloc and nano-malloc. In fact, our BeyondRISC implementation could be optimized further by enabling `malloc` to be inlined. The third benefit is that the program's code size and space for statically allocated data decrease substantially. For example, dlmalloc requires about 2 KB of code and over 1 KB for static data when compiled with a minimal set of features. In contrast, an implementation of `malloc`, `calloc` and `free` alongside the IHGC only requires a handful of instructions and no statically allocated data.

8.3.2 FreeRTOS and Mbed TLS

We ported the open-source software libraries FreeRTOS and Mbed TLS to our simulated BeyondRISC platform. FreeRTOS is a Real-Time Operating System (RTOS) kernel that is widely used in embedded devices [11]. The software is mostly written in C with some architecture-specific assembly functions for core features like scheduling and timers. Mbed TLS is an implementation of the TLS protocol and cryptographic algorithms to establish secure communications over a network [22]. We developed a program that uses both FreeRTOS and Mbed TLS to establish a secure channel between the BeyondRISC system and a host computer over a simulated serial device. Our program exercises and demonstrates the use of our proposals to handle interrupts, exceptions and I/O devices alongside the IHGC.

FreeRTOS and Mbed TLS both rely on explicit memory management based on the traditional `malloc`, `calloc` and `free` interface. But they do not make any assumptions about the underlying memory management algorithm. In fact, FreeRTOS supplies five different memory management algorithms that can be easily configured out-of-the-box [12]. We simply configured our program to use the IHGC-based `malloc`, `calloc` and `free` implementation from Listing 8.4. FreeRTOS and Mbed TLS also have dependencies on the C standard library functions `memcpy` and `memmove` to copy blocks of memory. In their simplest form, these functions are implemented to copy memory one byte at a time as shown in Listing 8.5 from the Newlib libc [138]. However, such implementations are unsuitable for a system with the IHGC because the tag information, which is maintained in memory at word granularity, is not preserved when pointers are copied one byte at a time. To solve this problem, we replaced the default `memcpy` and `memmove` implementation

```
1 pxTopOfStack = &(pxNewTCB->pxStack[ulStackDepth - (uint32_t)1]);
2 pxTopOfStack = (StackType_t *)(pxTopOfStack & ~portBYTE_ALIGNMENT_MASK);
```

Listing 8.6: FreeRTOS calculating the address of the last word in a contiguous stack [11].

with a version that copies memory one word at a time when the locations are aligned to a word boundary.

We developed assembly code that implements architecture-specific components of FreeRTOS. These include: the startup file, drivers for our OMIO timer and serial devices, and stack handling at thread creation and context switch. The changes are all self-contained and none of these required modifying the core FreeRTOS source code. There are two important implications for our program. First, interrupt handling routines are permitted to dynamically allocate memory. Traditional explicit memory managers cannot be used in interrupt handlers because they can potentially run for a long time. Also, these memory managers are often unreliable and can fail due to fragmentation. In contrast, allocations with the IHGC are fast and do not suffer from fragmentation. So we implemented the driver for our serial device such that it dynamically allocates memory buffers on-demand to hold the incoming network data.

The second implication for our program is that the code for handling the function call stack is simplified. FreeRTOS assumes that the stack is contiguous, so the stack size needs to be fixed at thread creation. To mitigate overflows, FreeRTOS can be configured to check simple overflow conditions when a context switch occurs. However, the checks are unreliable and incur performance overheads. Also, stack overflows are likely to be caught after they occur, so the program could have already corrupted the system's memory. These problems are completely eliminated when FreeRTOS runs on BeyondRISC because the stack is linked and the IHGC performs the overflow checks automatically for every memory access instruction. The stack size no longer needs to be fixed and the overheads of stack overflow checking in software are redundant. In fact, we completely disabled the checks because the code is incompatible with the IHGC's approach to memory management.

Only two lines of FreeRTOS's core source code needed changing for the program to execute correctly on BeyondRISC. The problematic lines, shown in Listing 8.6, attempt to calculate the address of the last word in a contiguous stack during thread creation. Line 1 obtains a pointer to the last byte of the allocated stack space. A bitwise-and is then performed in line 2 with the pointer and a bit pattern as operands. The result of the calculation is stored on a data structure that represents the newly created thread. FreeRTOS uses this information to perform stack overflow checks during a context switch. However, BeyondRISC's bitwise-and instruction gives rise to an exception when any of the operands is a pointer. As a result, we deleted these two lines of code as the stack overflow checks are automatically performed by the IHGC and to avoid failures at run-time.

In summary, porting FreeRTOS and Mbed TLS to an architecture with the IHGC requires

```
1 if ((txt[index - 1] == '\n' || txt[index - 1] == '\r') && txt[index] == '\0') {
2     ...
3 }
```

Listing 8.7: Memory access bug in LittlevGL's source code [95].

minimal changes to the software's core source code. Only one change was needed to ensure that bitwise operations are not used on pointers. The IHGC provides benefits, such as allocations in interrupt handlers and the elimination of stack overflows, that are difficult to match in other systems.

8.3.3 LittlevGL

LittlevGL is a library used to implement embedded graphical user interfaces, such as those found in printers and home appliances [95]. We developed a C program based on LittlevGL's benchmarking and demonstration code. This software also relies on the explicit `malloc`, `calloc` and `free` memory management interface. But LittlevGL wraps every call to these functions with its own memory management code. Before calling `malloc`, the library increases the requested allocation size with space to store a header alongside the program's data. The header contains the object size and a *used* flag indicating whether the object has been freed.

LittlevGL's wrapper code around the underlying memory management algorithm performs arithmetic and bitwise operations on pointers. This gives rise to exceptions in an IHGC system, so the wrapper was replaced with the IHGC implementation in Listing 8.4. But this change uncovered memory access bugs in LittlevGL's source code that were hidden by the original memory management wrapper. For example, Listing 8.7 shows a fragment of code where the IHGC raises an out-of-bounds memory access exception. The variable `txt` is a pointer to a dynamically allocated buffer containing a string. So a problem occurs when the `txt` buffer is of zero length; the variable `index` is 0 and the array element accessed is `txt[-1]` which is clearly an error although LittlevGL's developers already published a patch that resolves the problem [94]. The IHGC raises an exception for such memory accesses, but the operation mistakenly results in a load of the object's last header byte when using LittlevGL's wrapper.

A related problem is that the original memory management wrapper code pads the requested allocation such that the new object's size is aligned to a word boundary. For instance, allocating five bytes actually results in a 12 byte object in a 32-bit system due to the padding and header. However, this practice hides out-of-bounds memory access bugs by small amounts e.g. 1, 2 or 3 bytes if the word size is 4 bytes. For example, accessing the byte at index 7 in the 5 byte object would not cause as an out-of-bounds error, although it technically is a failure, because of the padding. We eliminated this risk because our program relies on the IHGC, which automatically checks bounds at byte granularity, instead of LittlevGL's memory management wrapper.

The graphics library uses data types that are represented as 8-bit or 16-bit elements. For

```
1 // Words with small integers have their least significant bit set to 1.
2 // These integers can be represented in 31-bits so they are stored as
3 // individual words
4 static inline bool MP_OBJ_IS_SMALL_INT(mp_const_obj_t o) {
5     return ((((mp_int_t)(o)) & 1) != 0);
6 }
7 #define MP_OBJ_SMALL_INT_VALUE(o) (((mp_int_t)(o)) >> 1)
8 #define MP_OBJ_NEW_SMALL_INT(o)   ((mp_obj_t)((((mp_uint_t)(o)) << 1) | 1))
9
10 // Pointers are stored word-aligned, so their least significant two bits are
11 // always 0
12 static inline bool MP_OBJ_IS_OBJ(mp_const_obj_t o) {
13     return ((((mp_int_t)(o)) & 3) == 0);
14 }
```

Listing 8.8: Code fragment from MicroPython's interpreter performing bitwise operations on arbitrary words to decode type information [115].

example, a color is represented as a 16-bit integer. Therefore, our program occasionally performs *misaligned* memory accesses when loading or storing into data structures. That is, the address of a memory access is not aligned to the machine's natural word boundary e.g. 4 bytes in a 32-bit processor. This is problematic for the IHGC because misaligned word accesses can result in the program partially loading or storing a pointer. Unfortunately, this may be used either by mistake or maliciously to modify a pointer handle, thereby compromising the system's stability. To prevent these problems the resulting type from a misaligned load is always set to data instead of pointer. When performing a misaligned store, we always set the type tag of the affected word in memory to data.

In summary, porting LittlevGL required significant changes to replace the library's memory management wrapper code. However, the changes were self-contained and using the IHGC instead helped us uncover bugs in other parts of the software. Additionally, small changes to our architecture were required to ensure that misaligned memory accesses succeed alongside the IHGC without compromising correctness.

8.3.4 MicroPython

MicroPython is an open-source implementation of the Python programming language for embedded devices [115]. The software consists of two main components: compiler and interpreter. The compiler transforms Python scripts into a bytecode representation. The interpreter is a stack-based virtual machine that executes bytecode programs. For simplicity, we only ported the interpreter to our simulated BeyondRISC platform. Python scripts are precompiled on a Linux computer using MicroPython's compiler and loaded onto the simulator alongside the executable for the interpreter. For these experiments, we set up the interpreter to run the same scripts used in Chapter 7 from the Python Benchmark Suite [66].

The interpreter is mostly written in C, so we successfully compiled it using LLVM. Two main adaptations were required. First, we added BeyondRISC assembly to implement MicroPython's architecture-specific functions, such as exception handling. Second, the interpreter relies on a garbage collector to automatically manage the memory. By default, MicroPython is configured to use a software garbage collector implementing a basic stop-the-world mark-sweep algorithm. We reconfigured MicroPython's build system to replace the default collector with the IHGC using the `malloc`, `calloc` and `free` implementation from Listing 8.4. Neither of these two changes required modifying MicroPython's core source code beyond what is typically necessary to port the software to a new platform.

MicroPython uses the two least significant bits in a word to encode type information. Specifically, 31-bit integers have their least significant bit set to 1. Pointers to objects are always stored in memory word-aligned, so the least significant two bits are 0 in a 32-bit system. Encoding types in this fashion requires casting words between pointer and integer types as implemented by the MicroPython code shown in Listing 8.8. Unfortunately, this conflicts with the IHGC's type system as discussed in Section 8.2.2, so it was necessary to modify the interpreter's source code to eliminate the execution of bitwise and arithmetic operations with operands of illegal type. Specifically, we replaced the C preprocessor macros and functions shown in Listing 8.8 with assembly code that checks the IHGC tag bit to determine whether the word is a value with primitive type, like a small integer, or a pointer to an object.

During experimentation, we discovered bugs in MicroPython thanks to the IHGC's run-time checks. The bugs were caused by MicroPython's use of operations with undefined behavior according to the C standard. These operations gave rise to out-of-bounds memory access exceptions in the IHGC system, but appeared to execute successfully in a conventional processor. We developed fixes for these issues and reported the failures to the MicroPython developers who have acknowledged the problem [9].

The changes to port MicroPython to BeyondRISC are self-contained. Also, the IHGC improved the software's reliability by uncovering bugs in the source code. Compared to our MicroPython port from Chapter 7, the BeyondRISC port has lower memory requirements because the IHGC is an exact collector and the stack is linked instead of contiguous. In addition, the toolchain does not store pointers in code objects, so the amount of collection work is reduced substantially.

8.3.4.1 Modern Languages and the IHGC

By porting MicroPython to BeyondRISC, we demonstrated how Python, a modern programming language, benefits from running on a system with the IHGC. First, the interpreter is simplified because the garbage collector no longer needs to be implemented in software. Second, the performance increases because the collector runs in the background concurrently with the application, so the processor can be fully utilized to execute the user's program as discussed in Chapter 7. And third, the system is safer and more reliable as the IHGC automatically checks

common causes of memory access errors.

The IHGC's benefits can be extended to other modern programming languages that rely on garbage collection. There is a growing ecosystem of modern languages such as JavaScript, PHP, Perl and Julia. These are dynamically typed and interpreted like Python, so they can be implemented in a fashion similar to MicroPython. There are many other garbage collected modern languages, like Java, C# and Go, that are strictly typed and would also benefit from the IHGC. Once again, these can be interpreted as MicroPython, but they can also be compiled directly to machine code. For example, Ed Nutting, from the Trustworthy Systems Laboratory, developed a tool to compile C# code directly into BeyondRISC machine instructions although this work is not published. The resulting programs had a minimal runtime environment as the collector was no longer in software. In conclusion, there is evidence showing that the IHGC can be used alongside modern programming languages.

8.3.4.2 Concurrency, Continuations and the IHGC

Concurrency features have started to appear in modern programming languages. For example, Go provides explicit support for managing the so called *goroutines*, or threads, along with message passing via channels. These features rely on the garbage collector to relieve the programmer from tedious tasks like reclaiming memory or closing unused channels as goroutines terminate. The IHGC can clearly facilitate the efficient implementation of both features. Other programming languages have adopted continuations. For instance, Python supports generators while C# introduced asynchronous programming support with async and await. Continuations force the runtime to keep track of the memory for multiple execution contexts which can be resumed at any time. These can be difficult to manage as programs grow in complexity. Software garbage collectors are known to mitigate these implementation problems of continuations at the expense of high performance overheads [53]. Once again, the IHGC can help develop efficient implementations of continuations because the memory management burden is no longer placed on the processor.

8.4 Summary

In this chapter we discussed the practical issues when integrating the IHGC into new and existing ISAs. Specifically, we explained how instruction semantics and I/O device interfaces can be adapted to work alongside the IHGC's type system. We investigated how linked stacks, stacklets and banked registers are used to implement function call stacks and interrupt handling without obstructing the garbage collector. And we demonstrated how executables can be linked to reduce garbage collection work and take advantage of the IHGC's error-checking capabilities. We explored these architectural features, through compiler design, to guarantee the system's correctness and reliability while broadly maintaining compatibility with existing software.

We ported open-source software to our proposed BeyondRISC ISA to evaluate the architectural changes discussed. We used a functional simulator and the LLVM compiler (with our BeyondRISC backend) for these experiments. In general, we found that the required porting changes are limited to replacing existing memory managers with an implementation using the IHGC. We were seldom required to modify the program's core source code except to fix software bugs or eliminate arbitrary bitwise and arithmetic operations on pointers; nevertheless, the changes were self-contained in both cases. We also discussed other features of modern programming languages, like concurrency and continuations, that the IHGC helps to implement efficiently.

CHAPTER 9

MICROARCHITECTURE OF THE IHGC

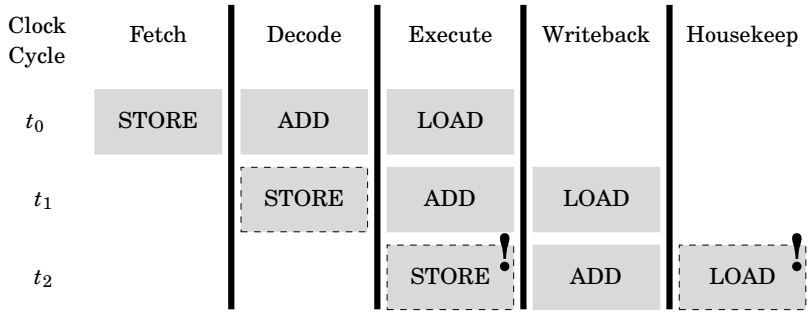
This chapter presents the microarchitecture of the IHGC alongside a pipelined processor. The design takes into consideration the capabilities of the current fabrication technologies used in embedded systems. We discuss the challenges of realizing the IHGC in hardware and estimate the implementation costs and efficiency of our system in comparison to equivalent embedded processors.

The proposed microarchitecture is based on approximations of the capabilities of VLSI technology. This data, obtained from the literature and through experimentation, is used to make informed design decisions and critically evaluate the hardware costs. The design is also used as the basis of a simulation model to evaluate the system's performance.

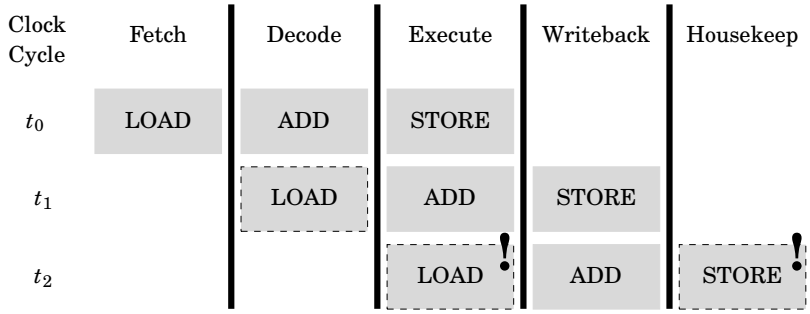
9.1 Overview

Several hardware components, including the main memory and register file, must be carefully considered when realizing the IHGC in hardware. But the directory warrants special attention as it accounts for most of the IHGC's memory overheads. The directory is also in the critical path of execution of every memory access instruction. Therefore, we need to ensure that the directory implementation carefully balances performance requirements and hardware costs.

The most challenging aspect of the IHGC's microarchitecture is minimizing contention within the processor's pipeline to access the directory. Modern embedded processors often have long pipelines to enhance performance. In this case, more than one memory access instruction can be in-flight at different stages of execution. But with the IHGC, all memory instructions require accessing the directory at least once and often more times to, for example, mark pointers loaded from memory. This causes pipeline stalls, and therefore delays, if the microarchitecture is not designed correctly. For example, the 5-stage pipeline shown in Figure 9.1 relies on a directory



(a) Directory contention due to `load` instructions.



(b) Directory contention due to `store` instructions.

Figure 9.1: Multiple pipeline stages in the processor simultaneously require using the directory when executing memory access instructions. The pipeline must occasionally stall if only one directory record can be accessed per memory cycle. The illustrations show two cases where such stalls occur. The dashed boxes are instructions that require a directory access at that pipeline stage. The boxes with the '!' represent pipeline stages that potentially stall during a clock cycle due to directory contention.

that supports accessing one record per clock cycle. The pipeline has a *housekeep* stage that marks pointers loaded from memory and sets an object's deep flag when storing pointers as required by the IHGC. Delays can occur in two cases at clock cycle t_2:

1. A `load` instruction is at the *housekeep* stage while another memory access instruction is at the *execute* stage as illustrated in Figure 9.1(a). In this implementation, the *housekeep* stage requires accessing the directory as it marks loaded pointers from memory. However, the *execute* stage also uses the directory to load the address of the object accessed.

2. A `store` instruction is at the *housekeep* stage while another memory access instruction is at the *execute* stage as shown in Figure 9.1(b). The *housekeep* stage sets the deep flag of the object accessed when the word written is a pointer, so the directory is required. But the *execute* stage also needs the directory to load the address of the object being accessed.

In both cases, one of the pipeline stages must stall due to contention in the directory. The processor model used for the performance evaluation in Chapter 7 did not suffer from these prob-

Processor	16 nm	28 nm	40 nm	90 nm	180 nm
M0			✓	✓	✓
M0+			✓	✓	✓
M3			✓	✓	✓
M4			✓	✓	✓
M23		✓	✓		
M33	✓	✓	✓		
M35P	✓	✓	✓		
M7	✓	✓	✓		

Table 9.1: Process nodes used to manufacture ARM Cortex-M processors [15].

lems because it implements a short 3-stage pipeline where only one memory access was in-flight. We eliminate such simplifying assumptions in this chapter by exploring the microarchitecture of IHGC systems with longer processor pipelines.

We aim to propose a realistic microarchitecture. Therefore, an SRAM compiler is used to guide the design of the directory based on the capabilities of the fabrication technology. Information from the literature is also used to estimate the hardware costs and operating clock frequency of our proposal and to compare them with existing embedded processors.

9.2 Background

In this section, we provide background information about two aspects that significantly influence the IHGC microarchitecture: the *process technology* and memory.

9.2.1 Process Technology

Process technology refers to a specific semiconductor process and its associated design rules used for the fabrication of integrated circuits. Each process technology, also called a *process node*, is named according to its minimum feature size (e.g. 28 nm, 40 nm, 65 nm, etc). For example, the International Technology Roadmap for Semiconductors (ITRS) traditionally defines the process node as the minimum half-pitch of contacted lines in the lowermost metal layer (Metal 1) of the interconnect [154]. However, alternative metrics are often used by integrated circuit manufacturers and foundries.

Process nodes of smaller geometries are regularly introduced to reduce area and power consumption or achieve faster processing speeds. However, embedded systems are rarely manufactured using the most advanced process node available; older process nodes of larger geometries are generally used instead. For example, the Cortex-M series of embedded processors licensed by ARM are normally manufactured using 16 nm to 180 nm process nodes as shown in Table 9.1, although the 10 nm process technology is currently available [176].

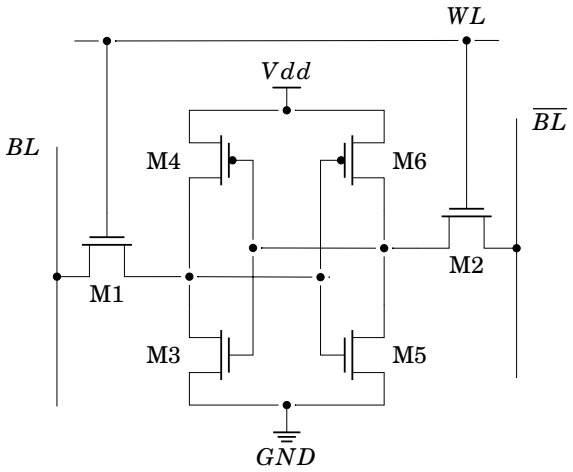


Figure 9.2: Structure of a 6 transistor (6T) SRAM bit-cell.

9.2.2 Memory

Modern embedded processors often include fast on-chip memory. The amount of memory varies depending on the intended application, but typically it ranges from a few KBs to 1 MB. The choice of technology and the amount of memory must be considered carefully as embedded memories often occupy over 50% of the total die area [83]. Ideally, the memory has low access latency, high density and low power consumption, but these goals are generally conflicting.

Static RAM (SRAM) is the dominant on-chip memory technology used in embedded systems. It is a fast memory that can be directly integrated with the CMOS logic, but it has relatively low density compared to *embedded dynamic* RAM (eDRAM). An SRAM consists of a *bit-cell* array along with peripheral circuits and control logic. A bit-cell stores one bit of information and is usually implemented using 6 transistors (6T). However, alternative bit-cell designs with 4, 7, 8 and 10 transistors exist that have different properties with regards to density, timing, power, reliability, etc [6, 63, 125].

The structure of the 6T bit-cell is shown in Figure 9.2. There are two access transistors (M1 and M2) and four transistors implementing two cross-coupled inverters that store the bit-cell's data. A word line (WL) and two bit lines (BL and $\overline{BL}$) are used to read and write the bit-cell. The WL is low when the bit-cell is in standby; the access transistors are disabled and the cell retains the value written last. To read the cell, WL is driven high to enable the access transistors, then the information is sensed at the bit lines by a sense amplifier. To write the cell, the bit value to write and its complement are applied to BL and $\overline{BL}$ respectively, then WL is driven high. This causes the value in the bit lines to override the previous state of the cross-coupled inverters.

The architecture of an SRAM is shown in Figure 9.3. The bit-cells are grouped into an array where each word line (WL) corresponds to a different row. A decoder takes an address as an input and drives high the word line containing the word of data to be read or written. Each pair of bit

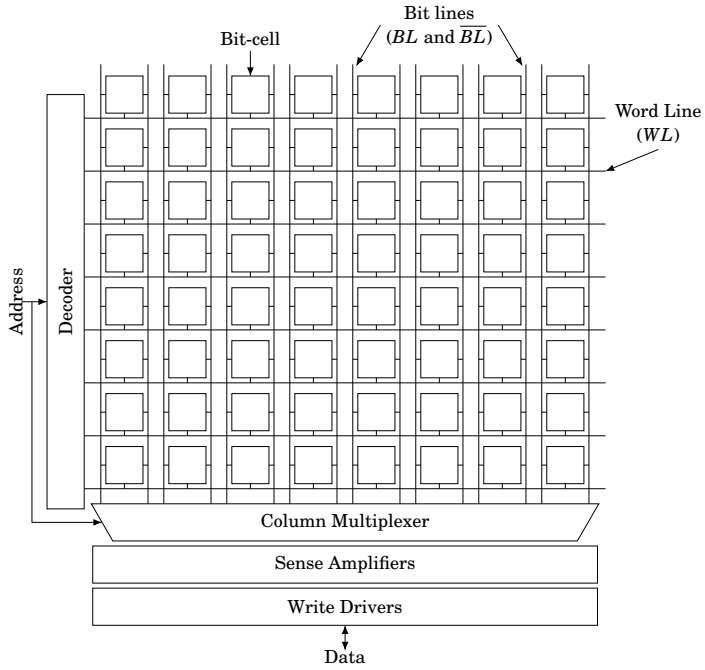


Figure 9.3: Architecture of an SRAM.

lines (BL and $\overline{BL}$) forms a column that is connected to the write drivers and sense amplifiers. Rows often contain multiple data words to reduce the complexity of the address decoder and the length of the bit lines [106]. Therefore, a column multiplexer is needed to connect the write drivers and sense amplifiers to the correct word when reading and writing.

Multi-ported SRAMs can be constructed at the expense of additional hardware. These SRAMs have more than one data and address port, so they support performing multiple read or write operations to different addresses in the same memory cycle. SRAMs with two ports, also called *dual-ported*, operated as normal memory devices can often be found in modern embedded systems while SRAMs with more than two ports are used for specialized hardware like register files. Multi-ported SRAMs require larger bit-cells due to the need for additional access transistors. For example, the 8 transistor cell (8T) is used instead of the 6T cell for dual-port SRAMs. Also, each additional port requires duplicating most of the peripheral circuitry and control logic, such as the address decoder and column multiplexer. This increases hardware costs and access latency, but the ability to complete multiple memory operations simultaneously can result in better performance.

Nowadays, memory compilers are used to automatically generate SRAMs. Memory compilers take advantage of the regular structure of SRAMs to produce designs for various configurations and process nodes quickly. The compiler takes configuration files and a *Process Design Kit* (PDK) as an input. Common configuration parameters include the word size, number of words and number of ports. The PDK is a set of files that describe a process node and its associated design

rules; it is usually provided by the foundry. On success, the memory compiler outputs files describing the SRAM's design along with its area, timing and power features. This information is used to implement integrated circuits, but it is also extremely valuable for architectural and microarchitectural design exploration.

The IHGC's directory is a key component implemented with SRAM. Therefore, we use the open-source memory compiler OpenRAM to automatically generate multiple SRAM configurations [76]. The compiler's output is used to guide the design of our proposed microarchitecture and estimate its area. We also use FreePDK45 for all our experiments with OpenRAM [170]. FreePDK45 is an open-source PDK for a 45 nm process node. Its design rules are constructed based on information collected from the ITRS and conference publications. Therefore, the generated SRAMs cannot be used for fabrication, but they are a reasonable approximation suitable for VLSI research and microarchitectural design exploration.

9.3 Microarchitecture of an IHGC System

This section describes the microarchitecture of an IHGC system. Our aim is to design an embedded processor implementing the BeyondRISC instruction set with the architectural features discussed in Chapter 8.

9.3.1 Main Memory

The main memory stores the header word and contents of every allocated object. The IHGC uses tagged memory to exactly distinguish pointers from data. So every word in memory contains 32 bits of data and a tag bit that indicates the type. For simplicity, we assume that all main memory is on-chip; a common arrangement in modern embedded systems such as Nordic Semiconductor's nRF52832 [126], STMicroelectronic's STM32F7 [172] and NXP's i.MX RT1020 [128].

We implement the main memory using multiple single-ported SRAMs following common industry practice. This is because a large SRAM with 33 bits per word generally has longer access latency and higher power consumption compared to multiple smaller memories with equivalent storage capacity. For example, we could use two SRAMs each with the same number of words. The first SRAM has 16 bits per word and only stores data. The second has 17 bits per word to accommodate 16 data bits and the tag. It is feasible to generate such SRAMs with a word size that is not a power of two, e.g. 17 bits, using memory compilers, like OpenRAM, because unusual word sizes are often required to implement memories with, for example, error correcting bits. Many other arrangements for the main memory are also possible provided that they yield an acceptable tradeoff between power consumption, area and timing delays.

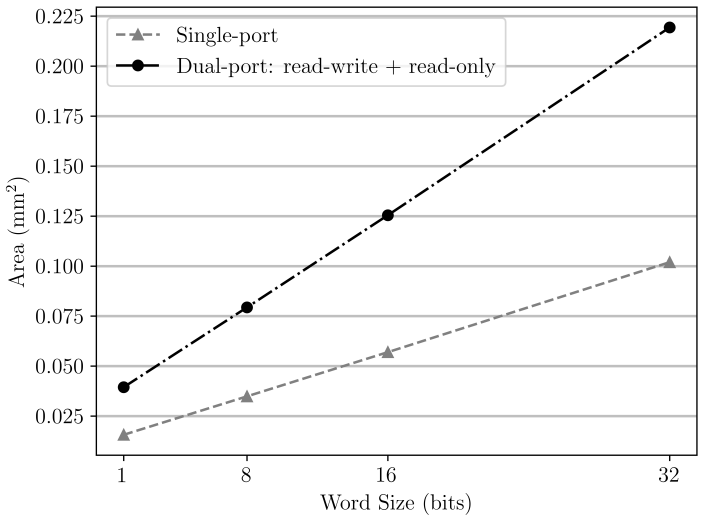


Figure 9.4: Area of single- and dual-ported SRAMs as the word size increases. The memories are 2048 words deep generated using a 45 nm process node in all instances.

9.3.2 Directory

The directory stores metadata for allocated objects and maintenance information for the garbage collector. It is in the critical path of execution of every memory access instruction, so the directory must be fast. Additionally, memory access instructions occasionally require reading or writing the directory more than once. Therefore, we must ensure that concurrent accesses to the directory do not cause excessive pipeline stalls.

Our microarchitecture considers the directory as a self-contained component independent from the processor's main memory. This reduces contention as accesses to the directory and the memory rely on physically separate hardware components that can be accessed in parallel. The directory is implemented using two single-ported SRAM arrays clocked at the same speed as the processor. The first SRAM stores the address and size of every object, while the second contains the mark and deep flags along with the directory's list component and another copy of every object's size. Thus, both memories have the same number of words, i.e. one per directory entry, but their word sizes differ.

Implementing a large memory using two smaller SRAMs is a common technique to decrease memory access delays at the expense of modest hardware costs. This also enables us to perform two directory accesses, potentially using different handles, as each SRAM has its own read-write port. Thus we can eliminate most stalls when pipelining the execution of memory access instructions. For example, the address and size for a load instruction can be read from the first SRAM while the deep flag of an object is set in the second SRAM as part of an earlier store. Similarly, the pipeline can efficiently mark pointers loaded from main memory because the object's size is maintained in both SRAMs. The object size is required to check out-of-bounds

errors while the main memory address to access is resolved for a load or store instruction. The object size is also used to process a loaded pointer for marking because the size must be added to the IHGC's *livesize*. Both operations can be performed in parallel by maintaining two copies of the size in separate SRAMs.

Our directory's split design eliminates the most common pipeline stalls. However, contention still persists in some infrequent situations as explained in Section 9.3.4.3. These issues can be mitigated by implementing the directory using dual-ported SRAMs. But compared to single-ported memories, dual-ported memories have much higher silicon area requirements. According to our experiments with OpenRAM and FreePDK45, dual-ported SRAMs with one read-write port and one read-only port use approximately twice the area of a single-ported SRAM as shown in Figure 9.4. Another disadvantage of dual-ported memories is availability. The requirements of key hardware components, such as memories and register files, vary little across many different processors, so a market for pre-designed IP modules has developed. Engineers purchase these IP models and use them in their designs instead of developing every single component from scratch, thus saving time and reducing cost. However, the high overheads of dual-ported memories have forced engineers to avoid their use in embedded systems, so the available supply of these memories as IP modules is very limited. As a result, we avoided dual-ported memories in the microarchitecture of our system.

9.3.3 IHGC State Machine

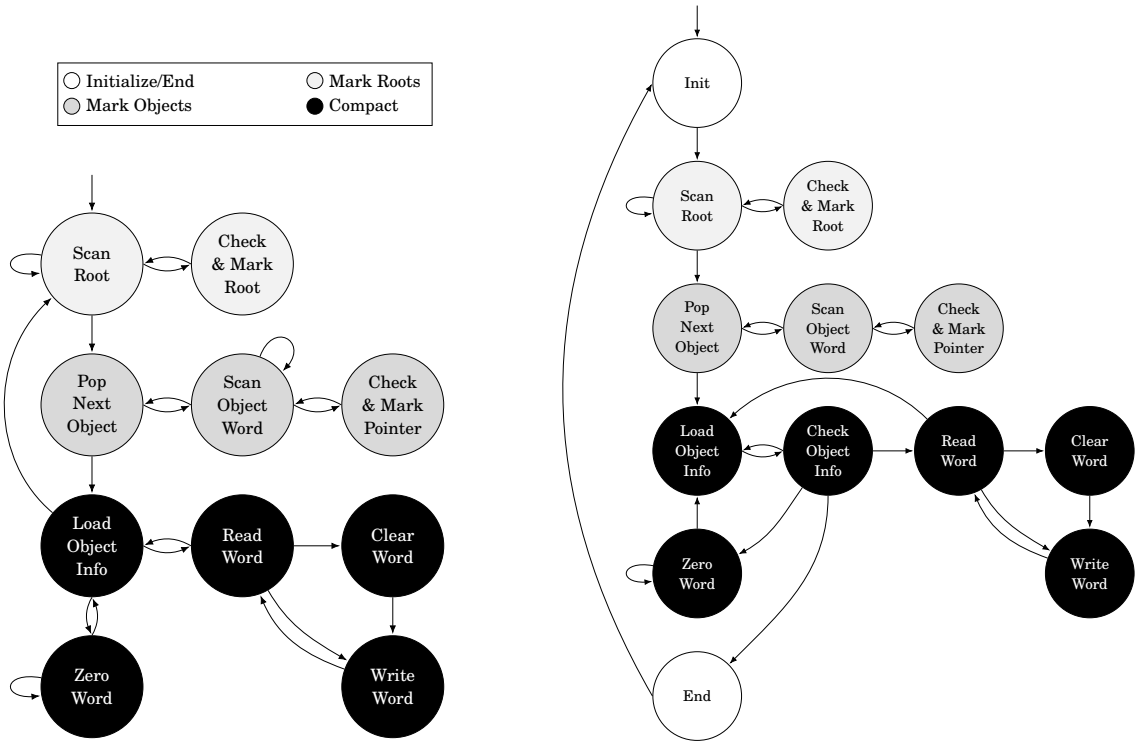
The microarchitecture of the IHGC's state machine is discussed in this section. We consider the main aspects that must be taken into account when realizing the state machine in the hardware.

9.3.3.1 States in the State Machine

The microarchitecture of the IHGC's state machine is shown in Figure 9.5(b). Its structure closely resembles the state machine in Figure 9.5(a) presented in the system-level description from Chapter 5. There are a few minor changes to ensure that every state transition can be completed in one memory cycle without pausing the processor.

- An `Init` state is added to set up the IHGC's internal registers for a new collection cycle. The collector visits this state once during each collection cycle to copy the contents of the register file to the shadow registers before marking the roots.

- An `End` state is added to check for failure conditions when a collection cycle terminates. During this state, the IHGC also notifies the processor, via an interrupt, that a collection cycle was completed. This facilitates error detection and containment, for example when the system has run out of memory, without increasing the complexity of the compact states.



(a) System-level state machine.

(b) Microarchitectural state machine.

Figure 9.5: Comparing the system-level description of the IHGC state machine with its microarchitectural implementation.

- During the compact stage, the IHGC uses the `Load Object Info` state in Figure 9.5(a) to inspect every object from the lowest memory address up to the *heappoint*. For each object, the collector loads the header word from memory to obtain the handle which is then used to load the object's size along with its mark flag from the directory. If the object is marked, the IHGC decides whether copying is needed. Otherwise, the unmarked object is reclaimed by adding its handle to the *free* list and zeroing if necessary. These operations require two memory cycles to complete because the directory load cannot be started before the load of the header word from memory is completed. We eliminate this problem by simplifying the former `Load Object Info` and adding a new state `Check Object Info` as shown Figure 9.5(b). The simpler `Load Object Info` is only used to load the directory metadata of the object being compacted. The new `Check Object Info` state uses the previously loaded directory information to decide whether the object being processed is live, take appropriate action and load the header word of the following object that the IHGC needs to process. This eliminates the dependency between directory and memory accesses and avoids pauses as the new state transitions can be performed in a single memory cycle.

Compared to the system from Chapter 7, the collection cycle of the modified IHGC state

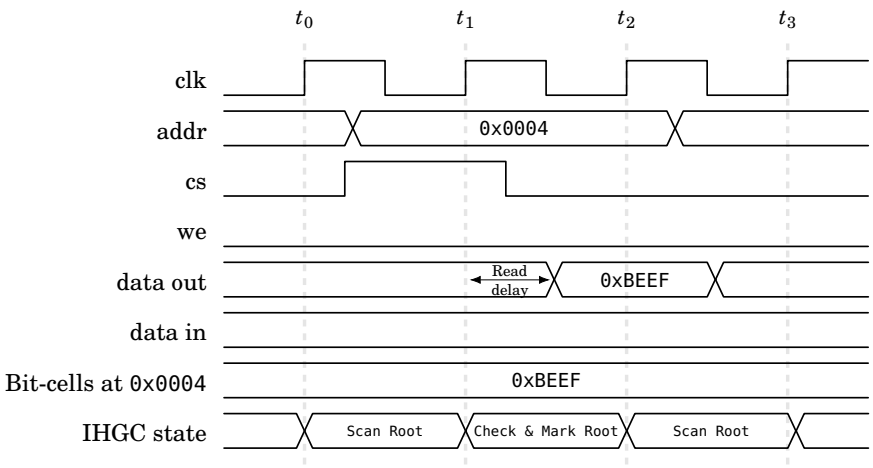


Figure 9.6: Timing of IHGC state transitions using the directory's synchronous memory interface. The data transfer is coordinated by the clock (`clk`) signal. The memory also has address (`addr`), chip select (`cs`), write enable (`we`), `data out` and `data in` signals.

machine has a slightly different run-time due to the addition of three states. But the general structure of the state machine remained unchanged, so both versions of the IHGC are suitable for hard real-time systems using the analysis technique described in Chapter 6. There are only two minor differences when analyzing real-time programs when using the modified collector. First, the constants used to construct the collector's timing model must match the new state machine implementation as described in Section 7.5.1. And second, the modified state machine operates alongside a BeyondRISC processor instead of an ARM Cortex-M0, so the processor's timing model is different.

In summary, our proposed microarchitecture for the IHGC adds three states to the state machine from Chapter 5. The modified IHGC state machine is also suitable for hard real-time systems. The purpose of the new design is to facilitate the state machine's implementation in hardware and ensure that state transitions can be completed in a single memory cycle without pausing the user's program.

9.3.3.2 Timing of State Transitions

Modern microprocessors normally use synchronous memory interfaces where the data transfer is coordinated by the clock. Figure 9.6 shows the operation of the synchronous interface implemented by the SRAMs generated using OpenRAM [76]. To load a word from memory, the address must be driven into the `addr` bus and the `cs` line set high. The SRAM samples the input signals at the next rising edge of the clock and the loaded data is observed in `data out` after a *read delay*. For example, the address `0x0004` in Figure 9.6 is sampled at time t_1 and the loaded data `0xBEEF` is placed in `data out` before t_2. Stores operate in a similar fashion as illustrated between t_2 and t_4 in Figure 9.7. The address and the data to store are driven into `addr` and `data in` respectively.

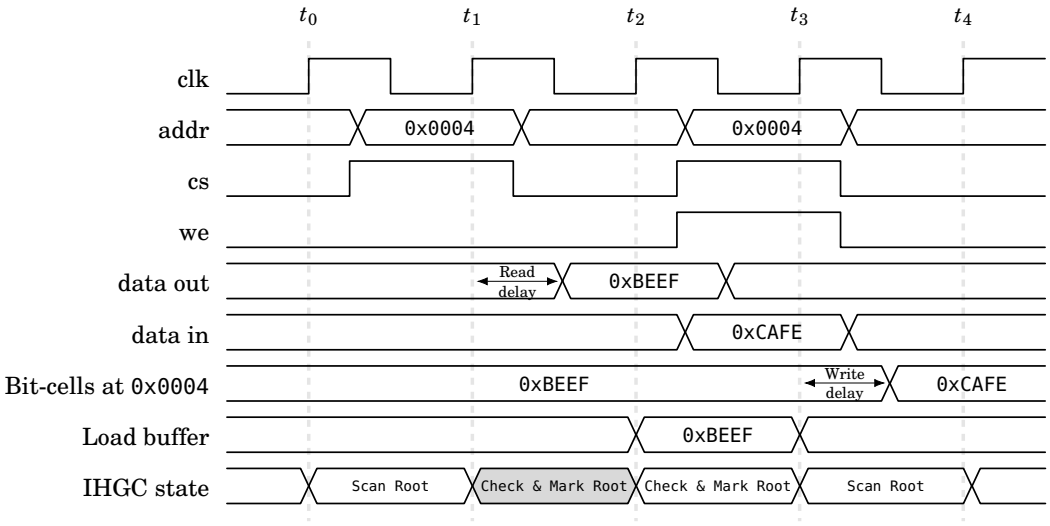


Figure 9.7: Timing of IHGC state transitions using the directory's synchronous memory interface. The shaded IHGC states represents a memory cycle allocated to the processor i.e. the collector is paused. During this time, the data that the collector loaded from memory is buffered until the next state transition occurs.

Also, `cs` and `we` lines are both set high. The memory samples these inputs at the next rising edge of the clock, i.e. at t_3, and the bit-cells corresponding to the word written change value after a *write delay*.

Synchronous memory interfaces must be used carefully to avoid introducing program pauses when implementing the IHGC's state machine. This is because multiple clock cycles are required to load data from memory. For example, in Figure 9.6 the load of `0xBEEF` required two clock cycles to complete: the inputs to the memory are driven before t_1, but the result is only observed in `data out` at t_2. Unfortunately, we cannot allow state transitions that rely on memory to last over one clock cycle as the collector's state machine would eventually pause the processor and consequently the user's program.

Our proposed microarchitecture implements a simple look-ahead strategy to eliminate the problem described above. During a state transition, it is always possible to decide whether the next transition will require accessing the directory or main memory. For example, the transition from `Scan Root` to `Check & Mark Root` at t_1 in Figure 9.6 occurs when a root contains a pointer with handle `0x0004` that needs to be processed for marking. At this time, it is already clear that the collector needs to access the directory because the following transition, from `Check & Mark Root` back to `Scan Root` (see Figure 9.5(b)), uses the mark flag to check whether handle `0x0004` is already marked. Therefore, we can implement the IHGC's state machine to drive the inputs to the directory one state transition *ahead* of when the data is actually needed. In our example, the directory inputs to load the metadata for handle `0x0004` are applied to `addr` and `cs` by t_1 as shown in Figure 9.6. This ensures that the IHGC performs the following state transition from

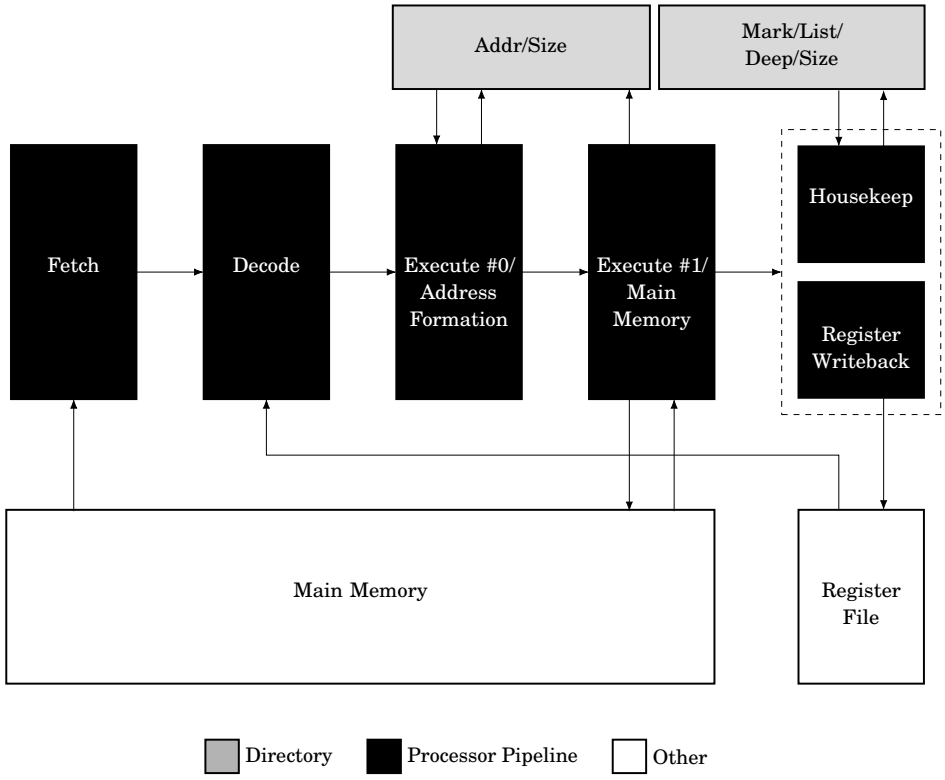


Figure 9.8: Structure of the 5-stage pipeline for an in-order, single issue embedded processor. The register file has four read and two write ports. The directory and main memory are implemented using a collection of single-ported SRAMs. The dashed box indicates that *register writeback* and *housekeep* are two components of a single pipeline stage.

`Check & Mark Root` to `Scan Root` in one clock cycle, i.e. by t_2, as the load would have already completed.

A related issue when accessing the memory is *buffering*. The operation of synchronous SRAMs may cause data loaded from main memory or the directory to appear in `data out` when the IHGC is paused due to interleaving. For example, in Figure 9.7 a load from the directory is initiated at t_1 due to the state transition from `Scan Root` to `Check & Mark Root`. But the processor is using the memory during the next clock cycle, so the IHGC is paused and cannot immediately use the loaded directory data in `data out`. To prevent losing the loaded data, we buffer it in a register while the IHGC is paused as shown in Figure 9.7 between t_2 and t_3. The data in the buffer is later used to perform the following state transition, by t_3, when a memory cycle becomes available.

9.3.4 Processor Pipeline

The proposed microarchitecture is for an in-order, single issue embedded processor. The structure of its 5-stage pipeline is shown in Figure 9.8. The pipeline is connected to a register file. There are

also connections via a bus to SRAMs implementing the directory and the main memory. Access to the memory bus is shared with the IHGC's state machine to allow the seamless interleaving of collection operations and instruction execution.

A *fetch* stage loads instructions from memory and puts them into an instruction buffer. The *decode* stage decodes the instructions in the buffer and drives the control signals and instruction operands for the following pipeline stage. Up to four operands can be read from the register file in one clock cycle. Address formation for memory access instructions occurs in *execute #0*: the object's base address is read from the directory and added to the pointer offset. Branches along with arithmetic and logic operations are also performed during *execute #0*. Main memory is accessed in the *execute #1* stage.

The last stage of the pipeline has two components which operate in parallel as there are no data dependencies between them. *Register writeback* writes the instruction's results (if any) into the register file. A maximum of two values can be simultaneously written into the register file. The *housekeep* component is used to implement collection operations associated with memory access instructions such as marking during loads. The operations at the *housekeep* stage are outside the critical path of instruction execution to avoid stalling the pipeline. However, only one instruction can be executing at the last pipeline stage at any time. It is not possible that, in the same clock cycle, *housekeep* executes a memory access instruction while *register writeback* commits the result of another instruction, like an `add`. Thus, *housekeep* and *register writeback* are two components that comprise a single pipeline stage.

The remainder of this section explains in detail the most important aspects of the pipeline's microarchitecture.

9.3.4.1 Instruction Fetch

The pipeline's *fetch* stage is responsible for loading instructions from main memory. The fetched data is stored in an instruction buffer for later decoding. We assume that the main memory supports 64-bit accesses, i.e. the `data in` and `data out` lines in Figure 9.6 are 64 bits wide. Therefore, about four instructions are loaded per fetch as BeyondRISC instructions are mostly encoded in 16 bits. This ensures that only a small portion of memory bandwidth is used for instruction fetches and more memory cycles can be allocated to the IHGC.

Normally, a directory access is required when loading or storing into an object. This is because the processor uses the object's address in the directory to calculate the location to access in main memory. But using the directory when fetching instructions has serious drawbacks. First, it increases contention on the directory and potentially causes stalls as there would be yet another component in the pipeline that competes for directory cycles. And second, instruction fetches consume more memory cycles that would otherwise be used by the collector. To avoid these problems, the fetch stage relies on cached directory metadata for the code object referenced by the *pc*. As a result, the directory is not accessed when fetching instructions.

A register in the pipeline caches two metadata items of the code object that are used in the fetch stage. The object's address is needed to calculate the address in main memory of the instructions to fetch. Also, the size is required to check that data outside the code object's bounds is not accessed while fetching. Out-of-bounds errors are signaled to the decode stage and an exception is raised when appropriate. The cached address and size metadata is overwritten by logic in the *execute #0* stage when a branch instruction is executed as explained in Section 9.3.4.2. In addition, the IHGC state machine updates the cached address when the collector relocates the object currently referenced by the *pc*.

9.3.4.2 Branches

Branch instructions write a new value into the *pc* so that execution continues at another address. Branches force a *flush* to eliminate partially completed instructions in the pipeline that are not in the new execution path, thus some work is lost. Modern designs mitigate this problem by resolving branches as early as possible in the pipeline. In our microarchitecture, branches are fetched and decoded as any other instruction. The branch destination address is fully resolved at the *execute #0* stage. As a result, only the work performed at the *fetch* and *decode* stages after the branch was fetched is lost when the pipeline is flushed.

The BeyondRISC architecture supports two types of branch instructions. *Relative branches* add or subtract an integer offset from the *pc*. In this case, it is guaranteed that the processor continues to execute instructions from the same code object where the branch originated. So the pipeline needs to be flushed, but the code object metadata cached for the fetch stage does not need to be updated. In contrast, *absolute branch* instructions overwrite the *pc* with a pointer. Therefore, the pipeline is flushed and the cached metadata for the fetch stage is updated with the address and size of the code object at the branch destination. As stated before, both relative and absolute branches are resolved at the *execute #0* stage.

The BeyondRISC processor is aware of every allocated code object, its size and location in memory. So it is possible to implement safety checks, and report errors via exceptions, while executing branches at the *execute #0* stage. For example, it is possible to detect when a program is attempting to branch to an invalid destination, such as a `NULL` pointer, or outside the bounds of a code object. These checks can be completed fully in parallel with program execution and do not require additional instructions. Compared to other systems, the checks help detect and contain errors as soon as they occur without incurring performance or code size overheads.

9.3.4.3 Memory Access Instructions

Memory access instructions are challenging to implement in a pipelined processor alongside the IHGC. Operations, like loads and stores, require accessing the directory at least once and sometimes more times. This causes contention and regular stalls if the pipeline is not structured

well. The design of our microarchitecture is motivated by the need to prevent these problems so that memory access instructions are executed efficiently.

After fetching and decoding, the execution of memory access instructions is split into three parts. First, the address to access in main memory is resolved during *execute #0*. For this operation, the handle is extracted from the instruction's pointer operand and used to read the object's address and size from the directory. The address and the pointer offset are added to calculate the main memory address to access. Simultaneously, the object's size is used to evaluate whether the memory access is out-of-bounds; any errors detected give rise to an exception.

The second part in the execution of memory access instructions is performed in *execute #1*. The address calculated during *execute #0* is used to load or store data to main memory. Typically, the main memory is a collection of large SRAMs, so the read and write delays are longer compared to accessing the directory. Finally, the third part commits the memory access and performs maintenance operations in the last pipeline stage. When storing, *writeback* is usually idle as the registers do not have to be written. The *housekeep* stage sets the directory's deep flag for the accessed object if the word stored in *execute #1* contains a pointer. In the case of loads, *writeback* writes the data read from memory into the register file. If the loaded word is a pointer, then it is processed for marking by *housekeep*. For both loads and stores, the operations performed at *writeback* and *housekeep* do not have data dependencies and can be performed in parallel.

The pipelined implementation of loads and stores requires the directory to be split into two SRAMs, as discussed in Section 9.3.2, to prevent stalls. In addition, both SRAMs must contain a copy of the size component. This arrangement enables up to three memory access instructions to be in-flight at different stages of execution in the pipeline. For example, a pointer loaded from main memory can be processed for marking at *housekeep* while the address for a store is being calculated in *execute #0*. Both operations require using the size component of the directory from potentially different handles simultaneously. This is supported by our microarchitecture without stalling the pipeline.

Stalls can still occur when executing a sequence of memory access instructions that load pointers from memory. This is because processing each pointer for marking takes at most two directory accesses at the *housekeep* stage. First, the pointer handle is used to load the mark and deep flags along with the object's size. Then the mark flag is set if the object is unmarked. The directory read is performed in one cycle during *housekeep*, but the store must occur in the following memory cycle. This means that any subsequent load instructions reading a pointer from main memory will be stalled at *execute #1* during that memory cycle. Fortunately, this situation rarely occurs and the performance impact is negligible as discussed later in Chapter 10.4.4.

9.3.4.4 Memory Allocation

Allocating memory with the IHGC is simple since the free space is always clustered at one end of memory. The following operations are performed in the processor's pipeline after an allocation

instruction (`newm`) is fetched and decoded. During the *execute #0* stage, the IHGC's *free* register is inspected to ensure a handle is available. In parallel, the processor compares the requested amount of storage space with the *heappoint* and the memory size to check whether there is enough free space in main memory. If both checks succeed, the handle at the head of the *free* list is popped. Otherwise, the pipeline is stalled until the IHGC reclaims enough memory and handles to fulfil the allocation.

The *execute #1* stage updates the new object's address and size components in the directory after a handle is successfully allocated. The address is set to the first memory location that was previously free, i.e the *heappoint*, and the size is updated with the requested amount of storage space in bytes. Simultaneously, the object's header is written to main memory and the old value of the *heappoint* is incremented to reference one byte after the end of the newly allocated object. There are no data dependencies between the operations performed at *execute #1*, so the hardware can easily complete them in parallel in a single pipeline stage. The execution of an allocation instruction concludes by writing a pointer to the newly allocated object into the register file during the *writeback* stage. Also, the *housekeep* stage processes that pointer for marking, as described in Section 9.3.4.3, and unconditionally writes the new object's size into the SRAM containing the directory's mark, list, deep and size components.

A pipeline hazard occurs when an allocation instruction is immediately followed by a memory access instruction. The newly allocated object's address and size are written to the directory during *execute #1*. At this time, the address for the subsequent memory access instruction is computed in *execute #0*. This requires reading an object's address from the directory. But the SRAM containing the directory's address and size components has a single read-write port that must be shared between the *execute #0* and *execute #1* stages. Therefore, the memory access must stall at *execute #0* until the allocation instruction reaches the *writeback* stage.

9.3.5 Interleaving

The IHGC operates in the background independently from the processor's pipeline. Both share access to the main memory and directory to perform their work. Collection operations are interleaved with memory accesses from the processor such that the IHGC never pauses the user's program. We can implement this technique using either of two approaches.

1. Access to main memory and the directory is exclusive to one driver. The IHGC cannot use the directory during the same memory cycle that the pipeline is accessing main memory. Similarly, the IHGC cannot use the main memory during the same memory cycle that the pipeline is accessing the directory. We implemented this approach in our evaluation model from Chapter 7. Enforcing exclusive access to the memory facilitates resolving coordination issues between the processor and collector. For example, accesses to objects being compacted are easily dealt with because the collector is paused while memory instructions are executed. Therefore, the interaction between the pipeline and the collector is greatly simplified at

the expense of wasting memory cycles when the pipeline only requires access to one of the memories.

2. The IHGC can access the main memory or the directory as long as the processor's pipeline does not require using that resource. For instance, the IHGC can perform a state transition that only requires accessing the directory while the pipeline is using the main memory. In this case, the IHGC operates in parallel with the processor as collection work progresses simultaneously with the execution of memory access instructions in the pipeline. This maximizes the use of the spare memory bandwidth and increases collection throughput, thus pause times are reduced. However, the design becomes more complex as it is difficult to coordinate the IHGC's operation with the processor.

We use option 1 from the list above in our proposed microarchitecture to simplify our design at the expense of reduced collection throughput. In this approach, access to both the directory and main memory is granted exclusively to either the processor or the IHGC. The rules for allocating a memory cycle are as follows.

- *Housekeep* has the highest priority to access the directory's mark, list, size and deep components.
- *Execute #1* has the highest priority to access the main memory and the directory's address and size components.
- *Execute #0* accesses the directory when *Execute #1* is not using it.
- An instruction fetch is performed when there is space in the instruction buffer and the main memory is not used by *execute #1*.
- The IHGC state machine performs a transition when the directory and the main memory are not in use by the processor pipeline.

9.4 Hardware Costs

In this section, we provide a *rough* estimate of the hardware cost to implement the IHGC state machine, the main memory and the directory. We put our estimates into perspective by comparing them with the hardware requirements of other components.

9.4.1 IHGC State Machine

We developed a *Register-Transfer Level* (RTL) implementation of the IHGC state machine in Verilog.[1] The design was then synthesized using the Yosys Open Synthesis Suite [190] and a

[1]The implementation was developed in cooperation with Ed Nutting from the University of Bristol's Trustworthy Systems Laboratory.

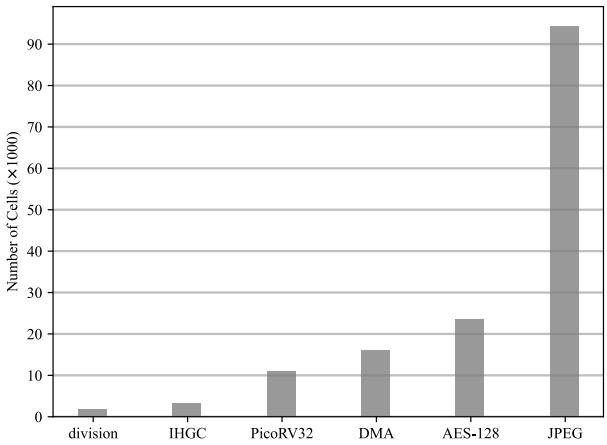


Figure 9.9: Synthesis hardware costs of the IHGC state machine and other open-source designs.

45 nm standard cell library. Synthesis transforms high-level RTL into an implementation that consists of basic hardware cells, like logic gates and multiplexers. We use the cell count as an estimate of the IHGC state machine's hardware cost. However, this is only an approximation as synthesis is the first step to tape-out a hardware design. Further processes, such as placement and routing, must also be performed to provide more accurate hardware cost estimates.

The synthesized IHGC state machine consists of 3284 cells out of which about 15% are D-Type flip-flops. Figure 9.9 compares the IHGC state machine's hardware cost with the synthesis results of the open-source designs listed in Table 9.2. The IHGC state machine is about three times larger than the *division* module. However, the logic required to implement our design is only a small fraction of the hardware used by the other modules. *PicoRV32*, a CPU core optimized for size, is about 3 times larger than the IHGC state machine while a more complex design, like *JPEG*, is over 20 times larger. Surprisingly, *DMA* is approximately 5 times larger than the IHGC even though both modules can be considered as memory controllers. This is because *DMA* implements a complex memory controller with up to 31 channels and is able to operate on complex data structures like linked lists and circular buffers. *DMA* also includes a scheduler to time-slice access to the memory interface across all channels according to configurable priorities.

We estimate that the IHGC state machine would increase the logic requirements of a small processor, like *PicoRV32*, by a factor of 1.3. But logic is only about half (or less) of the total chip area as modern processors have generous amounts of on-chip memory. Therefore, the overall hardware cost of the IHGC is almost negligible for a modern chip.

9.4.2 Main Memory

We quantify the hardware costs of adding a 1-bit type tag to every word in an SRAM. Unfortunately, OpenRAM currently does not support generating memories larger than 2048 words deep,

Design	Description	Reference
division	Integer division module extracted from the PicoRV32 project. Its inputs are 32-bit integers, but the design processes one bit per clock cycle only.	[189]
PicoRV32	Small processor core that implements the RV32IMC instruction set. The design is not pipelined and only includes a basic interrupt and exception handling mechanism.	[189]
DMA	Direct Memory Access (DMA) IP core that performs transfers between two WISHBONE interfaces.	[182]
AES-128	Implementation of the AES-128 cipher.	[181]
JPEG	JPEG encoder from the Video Compression Systems Project.	[183]

Table 9.2: Synthesized open-source designs compared in Figure 9.9.

Capacity (KB)	**Area** (mm^2)
64	0.212
128	0.425
256	0.849
512	1.699
1024	3.397

Table 9.3: SRAM area for selected capacities. The figures are calculated using the cell *area factor* and *array efficiency* from the ITRS 2009 report [152]. The area of the 6T SRAM bit-cell is given by the area factor multiplied by the square of the minimum feature size (F^2); it is $140 \times (45\,\text{nm})^2$ for the 45 nm process node. The array efficiency is the portion of the memory block that is occupied by the SRAM bit-cells as opposed to the port logic or other circuitry. The array efficiency is 70% for the 45 nm process node.

which is too small for modern embedded systems, so we estimated the overheads of type tags using figures from the ITRS 2009 reports instead.

The ITRS provides an *area factor* to measure the area of a 6T SRAM bit-cell as a function of the square of a process node's minimum feature size (F^2). The area factor for the 45 nm process is 140, so the area of an SRAM bit-cell is about $140 \times (45\,\text{nm})^2$. In addition, the ITRS reports the *array efficiency*, that is, the portion of the SRAM that is occupied by bit-cells as opposed to the port logic and other circuitry. The array efficiency in the 45 nm process node is 70%. Using the area factor and array efficiency, we estimate that the memory overheads due to the IHGC's 1-bit type tags is approximately 3.1%.

9.4.3 Directory

We estimate the memory overhead of the directory using the main memory SRAM (without the tag bits) as a baseline. The comparison is fully based on figures from the ITRS 2009 reports for

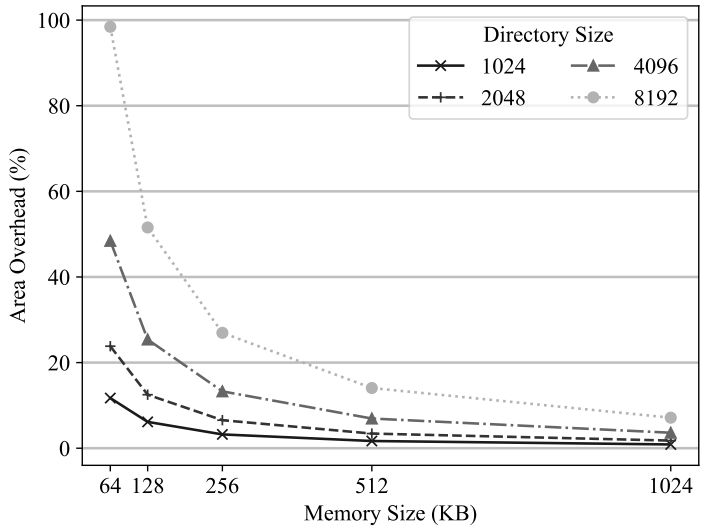


Figure 9.10: Area overhead of the directory when using the main memory as a baseline. The figures are calculated using data from the ITRS 2009 report for the 45 nm process node [152].

the 45 nm process node. The estimated areas for selected sizes of the baseline memory are shown in Table 9.3.

The area of the IHGC's directory depends on the following parameters.

Main Memory Size: The directory component storing the object's base address requires more bits as the depth of the main memory increases.

Maximum Object Size: The directory component storing the object's size requires more bits when the maximum amount of memory that it is possible to allocate for a single object increases. The maximum object size cannot be larger that the bytes addressable using a pointer offset.

Number of Directory Records: Each allocated object has a unique handle and an associated record in the directory. Therefore, the maximum number of live objects must match the number of records, i.e. the depth, in the directory. Also, the directory's list component must have sufficient bits to represent every handle. The maximum number of live handles is configured by the hardware designer and is limited by the number of bits in a pointer handle.

The information in the list above and Table 9.3 was used to calculate the directory's area overhead shown in Figure 9.10 for multiple memory configurations. The directory overhead increases dramatically as the number of records grows in comparison to the main memory size. However, several of these configurations are impractical. For example, a system with only 64 KB of main memory would not be paired with an 8192-record directory. If the entire memory were

Wire Type	**Wire Pitch** (nm)	RC **Delay** (ps/mm)
Metal 1	90	2,100
Intermediate	90	1,892
Global	135	542

Table 9.4: Interconnect features for the 45 nm process nodes. The information is extracted from the ITRS 2007 report [153].

allocated, each object would have an 8 byte average size, which does not happen in practice, so the directory is disproportionately large. Similarly, a 1024-record directory is too small for a 1024 KB memory. Researchers estimated that the average object size in a modern programming languages is 22-36 bytes [30, 184], so in general, we expect the directory area overhead to be 10-20% of the main memory to provide a reasonable number of records although this is highly dependent on system requirements.

9.5 Clock Speed

A design's clock speed is limited by the time it takes for a signal to propagate through the longest path between two sequential components. We found this *critical path* in our design using a Verilog RTL model of the IHGC alongside a BeyondRISC processor. Our experiments indicate that the critical path occurs when calculating an address to access memory. This operation involves using the pointer handle as an index into the directory to load the object's base address in main memory, after which the base address is added to the pointer offset. Therefore, the delay through the critical path is given by the directory's read delay, the propagation delay through the interconnect and the duration of an add. We estimate each of these individually.

Directory: The read delay of an SRAM depends on the memory architecture, its capacity and the process node. The ITRS reports state that the delay of a read access is approximately 0.2-1.2 ns for the 45 nm process node [155]. We assume a pessimistic 1.2 ns read and write delay.

Interconnect: The interconnect refers to the wires carrying the signals between the various hardware components on a chip. Modern microprocessors use several metal layers with wires of different pitches to mitigate interconnection delays. Metal 1, the lowest metal layer, contains the thinnest wires with the least pitch and the largest interconnect delays. These wires are normally used for short, local connections such as between gates. *Intermediate* and *Global* metal layers are higher in the stack and are normally used to connect larger components across greater distances. The interconnect delays for the 45 nm process are shown in Table 9.4. We assume that the components involved in generating an address, our design's critical path, are connected using wires in the intermediate metal layers. We

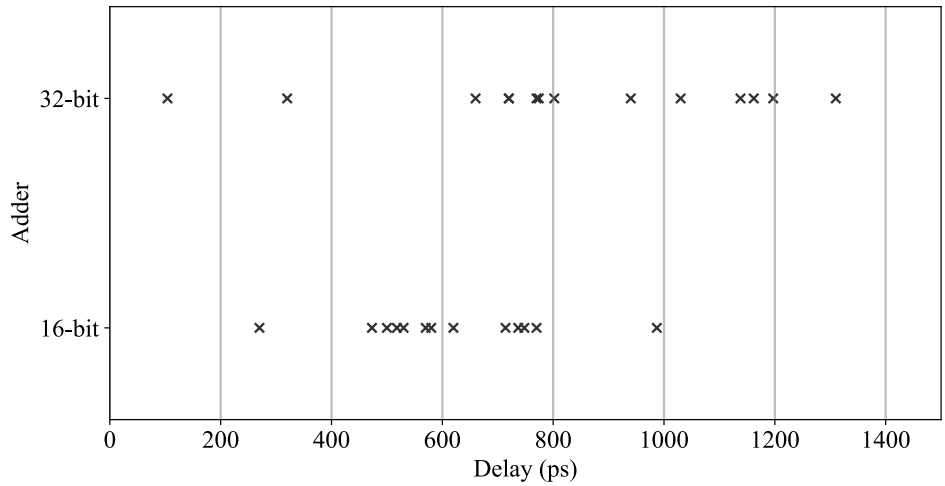


Figure 9.11: Propagation delays for various 16-bit and 32-bit adder designs in the 45 nm process node. The figures are obtained from the literature [90, 119, 120, 173].

also assume that the interconnect length is 0.5 mm, so its delay is about 946 ps. This is a conservative assumption as the total area of most embedded processors is often smaller than 0.5 mm^2.

Add: The delay of an adder is proportional to the number of bits in the result. For example, a 16-bit adder is expected to have a shorter delay than a 32-bit adder. Also, the algorithm that the adder implements greatly influences the adder's features. High-performance adders are optimized to reduce delays, but they have high power and area requirements. In contrast, adders optimized for low-power operation are implemented using much less hardware although they are slower. The delays of various 16-bit and 32-bit adder designs using the 45 nm process node are shown in Figure 9.11. A memory access with the IHGC typically involves performing an addition that does not exceed 20 bits, so we assume a conservative 700 ps delay.

We estimate the critical path in our design to be 2.84 ns. So the clock speed of our embedded processor with the IHGC is approximately 350 MHz when using the 45 nm process. As will be discussed in Section 9.6, this clock speed is within the range of frequencies commonly targeted for embedded processors. Further increasing the clock speed is possible, but this requires introducing significant changes to our proposed microarchitecture.

9.6 Discussion

In this section, we discuss the clock speed and hardware requirements of our proposed microarchitecture. Our aim is to understand how the results fit within the context of embedded systems

and garbage collection.

9.6.1 Clock Speed

Most embedded systems run at clock speeds between a few KHz and 600 MHz as their designs attempt to strike an acceptable tradeoff between run-time performance and power consumption. For example, STMicroelectronics, one of the leading electronics and semiconductor manufacturers, produces a wide range of 32-bit embedded devices based on ARM Cortex-M processors with clock speeds of up to 550 MHz [171]. So we conclude that our IHGC design alongside a BeyondRISC processor operating at up to 350 MHz is acceptably placed towards the higher-end of the range of clock speeds for embedded systems.

9.6.2 Memory Overheads

The IHGC's area overhead due to the directory and tag bits is reasonably small considering that on-chip memory usually occupies about half the chip's area. The remaining area is used for the processor's logic. So compared to a processor without the IHGC, our system only contributes to a total 6-12% area increase due to the directory and tag bits. We also expect this overhead to decrease if the chip contains caches, like the ARM Cortex-M7. Compared to software garbage collectors, we consider that this is an acceptable cost in exchange for the benefits of hardware garbage collection as discussed in Section 7.4.4.

9.6.3 Scaling Up

The proposed IHGC microarchitecture currently targets embedded processors with relatively short pipelines and flat, on-chip memory hierarchies. This simplified our design while allowing us to deliver the benefits of garbage collection with good performance. But how can we integrate our hardware collector into larger computer systems? This requires cross-cutting changes to our microarchitecture and the collector. We briefly discuss two considerations to scale up the IHGC.

Clock Speed: The clock speed in our microarchitecture is limited by the critical path to generate addresses while executing memory access instructions. An alternative to increase the clock speed is to add pipeline stages to break up the critical path into shorter, simpler operations. For example, the address calculation at the *execute #0* stage could be split into two parts: the first loads the object's base address from the directory while the second stage adds it to the pointer offset. However, increasing the pipeline length makes the design more complex.

Real-time Requirements: The IHGC's specification is designed to guarantee that the timing properties of the system can be statically analyzed. However, larger systems are typically soft real-time, so there is scope to modify parts of the collection algorithm to increase performance or decrease hardware costs at the expense of the IHGC's hard real-time

properties. For example, the IHGC's mark on load can be changed to a mark on store (conceptually similar to Brooks' write insertion barrier). This would simplify the pipeline's interaction with the directory and eliminate the need for the *housekeep* stage because we no longer need to wait for pointers to be loaded from memory before they are processed for marking. But Brooks' write barrier also requires a complex termination condition that degrades the collector's timing properties as discussed in Section 3.2.2.

9.7 Summary

We described the microarchitecture of the IHGC alongside a pipelined processor. The most important design decisions were highlighted including the pipeline structure and memory organization. Based on this information, we estimated the IHGC's tag bits to increase the main memory area by about 3% in the 45 nm process node. The directory incurs an extra 10-20% area overhead when using the main memory as a baseline although this depends on the system's configuration. Also, the IHGC's state machine incurs almost negligible logic area overheads for a modern chip. Finally, we estimate the processor to be clocked up to 350 MHz.

CHAPTER 10

EVALUATION OF THE IHGC MICROARCHITECTURE

In this chapter, we evaluate the performance of the hypothetical system proposed in Chapter 9. A timing-accurate simulator of the IHGC alongside the pipelined BeyondRISC processor is used for the experiments presented. We evaluate our design from three angles: memory requirements, pipeline stalls and garbage collection pauses. We also consider simple hardware optimizations to increase the IHGC's performance.

10.1 Evaluation Platform

We implemented a timing-accurate model in SystemC of the processor microarchitecture described in Chapter 9. SystemC is a C++ class library applied in industry to architectural exploration and performance modeling as in this thesis [24]. The library provides an event-driven simulation environment that deliberately resembles hardware description languages, like Verilog and VHDL, while allowing access to most facilities in the C++ programming language. Therefore, SystemC is an ideal technology to implement a realistic simulation model of the IHGC alongside a pipelined BeyondRISC processor.

SystemC enables several coding styles and abstractions to develop performance models. For example, the Transaction-Level Modeling (TLM) interfaces are an abstraction to separate the implementation details of modules from how they communicate. In TLM, the focus is on the details of the transaction rather than the modules communicating. For the IHGC's performance model, we are interested in the implementation details of our microarchitecture, so we adopt a coding style closer to RTL. However, we simplify our simulator in two ways. First, we model the interconnections between large modules, such as the IHGC and the pipeline, using hierarchical channels. Second, our model does not describe the implementation details of components unre-

lated to the IHGC, such as the Arithmetic and Logic Unit (ALU). These simplifications mitigate the model's complexity and reduce development time.

Our SystemC performance model implements the IHGC and processor under the following assumptions.

- The main memory has one read-write port and a main memory access can be completed in one clock cycle.
- A directory access and an add can be completed in one clock cycle. This ensures that address generation can be performed in a single pipeline stage when executing memory access instructions.
- The directory is implemented using two physically separate memories each with a read-write port. The directory memories can be operated independently.
- The timing of arithmetic and other non-memory access instructions remains as stated in the ARM Cortex-M0 Technical Reference Manual [19]

10.2 Benchmarks

We use a subset of the benchmarks from Chapter 7 to evaluate our system. The programs were ported to the BeyondRISC architecture as described in Section 8.3. They are split into four groups depending on their origin.

1. A collection of small C programs from the BEEBS benchmark suite [130].
2. Two C programs, *anagram* and *audiobeam*, from the TACLe benchmark suite [62].
3. A C program, *egui*, that benchmarks the construction of embedded graphical user interfaces, such as those found in printers and home appliances. We developed this software using the open-source library LittlevGL [95].
4. Ten scripts from the Python Benchmark Suite [66] executing on the MicroPython interpreter [115]. The version of the interpreter used for the experiments discussed in this chapter is the same as the program ported in Section 8.3.4. In comparison to the MicroPython interpreter used in Chapter 7, this version is simpler as it does not compile Python scripts at run-time; the scripts are pre-compiled to bytecode and loaded at boot-time.

10.3 Compiler and Toolchain

All benchmarks were compiled with a modified version of LLVM 9 that includes our custom backend for the BeyondRISC architecture [102]. This compiler incorporates the techniques

discussed in Chapter 8 to eliminate the drawbacks of the off-the-shelf LLVM that we used for our experiments in Chapter 7. Namely, our LLVM backend correctly handles function call stacks, emits instructions that do not violate type constraints and does not insert pointers into code objects. We also apply the generic code optimizations implemented by LLVM. Specifically, we set the -O2 flag, disable stack protection checks and emit position-independent code only. As a result, our system runs realistic code emitted by a production-quality compiler while benefiting from the increased security and performance provided by the IHGC.

Each program is compiled from C to assembly language using LLVM. For simplicity, we did not implement the assembler and linker in LLVM. Instead, we developed a custom Python tool that compiles to object code and links the assembly files generated with LLVM. Our Python tool uses the linking strategy outlined in Section 8.2.6 to ensure that the IHGC can check the bounds of global objects and variables at run-time. However, the Python tool does not implement link-time optimizations that are common in production-quality linkers. For example, the GNU Compiler Collection's (GCC) linker performs dead code elimination optimizations to reduce the size of the program binary. These issues are not limitations of the IHGC and can be easily addressed by enhancing our Python linker.

10.4 Results

In this section, we present and discuss our experimental results. Our aim is to quantify the run-time performance and memory usage of the IHGC and compare it with the earlier results from Chapter 7 where appropriate. We also measure empirically any pauses introduced by the collector and the performance implications of various optimization techniques that can be easily integrated with the IHGC microarchitecture that our SystemC model implements.

10.4.1 Memory Requirements

Our BeyondRISC system uses an exact collector and an unconventional linking strategy for embedded devices. Therefore, we expect the memory requirements of this system to change in comparison to the results for the ARM-based processor described in Chapter 7. In this section, we measure the differences between these two systems with regards to memory requirements for statically allocated global variables and the heap.

10.4.1.1 Global Variables

Statically allocated global variables are those whose memory is reserved at compile-time. A toolchain, like GCC, for a conventional architecture, like ARMv6-M, typically allocates space for these variables in the `bss`, `data` or `rodata` sections of the program's binary. As explained in Section 8.2.6, each section is allocated as a contiguous chunk of memory that can be arbitrarily indexed by the program. Our linker for BeyondRISC does not have the concept of program

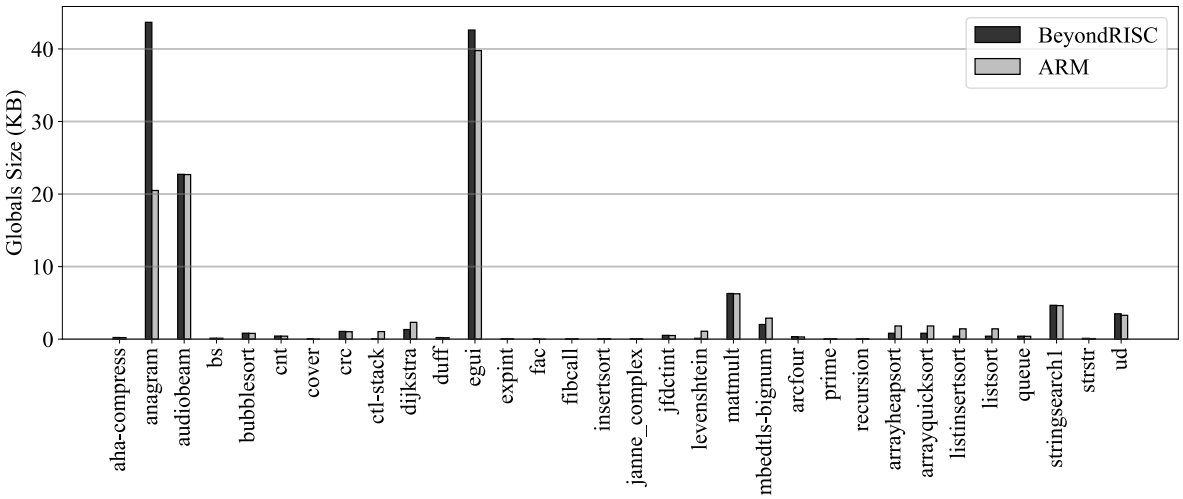


Figure 10.1: Memory size of global variables in our benchmarks. This corresponds to the aggregated size of the `bss`, `data` and `rodata` sections of a program compiled for a conventional architecture such as the ARMv6-M.

sections. Instead, global variables are allocated as independent objects accessible through an array of pointers. This arrangement leverages the benefits of the IHGC in that memory accesses to global variables are checked.

The memory requirements for global variables change considerably depending on the linking strategy. Figure 10.1 shows the space required for statically allocated globals with BeyondRISC and ARMv6-M. We can split the results into three groups. The first group consists of the benchmarks that have few global variables, like *bubblesort* or *fac*, so their memory requirements are very similar. The majority of these benchmarks are small programs where the linking strategy has very little impact on the final binary. The second group consists of benchmarks, such as *arrayheapsort* and *dijkstra*, in which ARM has slightly higher memory requirements compared to BeyondRISC. The difference, usually between 1-2 KB, occurs because these programs rely on `malloc` to dynamically allocate memory at run-time. The measurements shown in Figure 10.1 are taken from programs compiled for the ARMv6-M architecture without the IHGC. Therefore, the linked binary for ARM includes the code from *dlmalloc*, an explicit memory manager popular in embedded systems, that needs 1-2 KB of space for global variables. The implementation of `malloc` in BeyondRISC does not use any global variables, so the memory requirements are lower.

The last group of benchmarks includes *anagram* and *egui* where the memory requirements in BeyondRISC are significantly larger than in ARM. This discrepancy occurs due to memory overheads for headers and references to global objects incurred in the BeyondRISC linking strategy. For example, *anagram* relies on a dictionary of approximately 2600 strings that are statically allocated at run-time. The strings are usually very small, between 1-6 bytes, but each is allocated in a separate global object that is referenced from an array representing the dictionary.

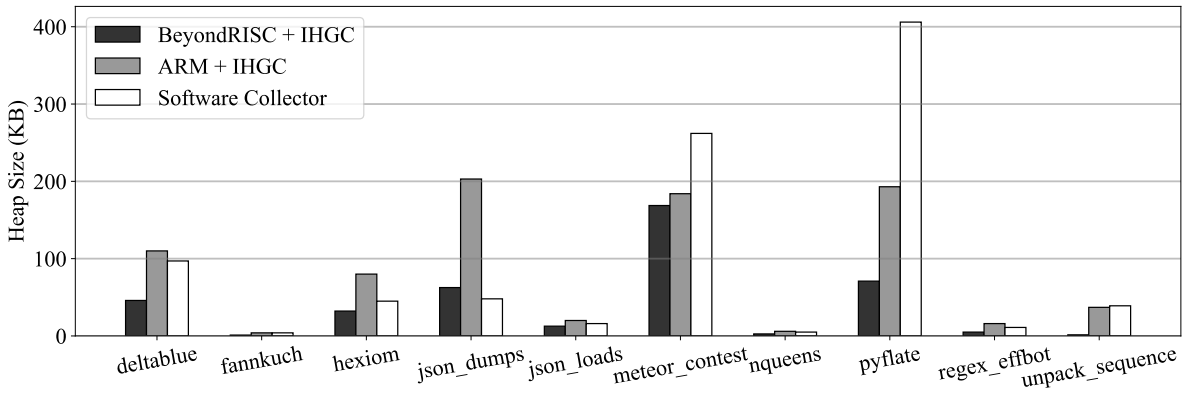


Figure 10.2: Minimum heap size requirements of the MicroPython benchmarks. The second and third bars correspond to the results shown in Figure 7.3 and discussed in Section 7.4.2.2. The 'Software Collector' bar refers to the mark-sweep garbage collector packaged with the off-the-shelf MicroPython interpreter running on the simulated ARM system described in Chapter 7.

So compared to ARM, the BeyondRISC program requires over 20 KB more of space for globals as each string incurs two words of overhead: one for the header and another for the array reference. A similar problem occurs in *egui* although this program allocates fewer global objects of larger size, so the overheads are limited to about 4 KB only.

In summary, the linking strategy that we implemented for BeyondRISC takes advantage of the security benefits of the IHGC. But it incurs memory overheads for statically allocated variables at compile-time. These overheads are generally modest, but they increase significantly when the program uses many small global objects. If required, the overheads can be mitigated by using a traditional linking strategy, similar to that implemented for an ARMv6-M system, at the expense of less effective run-time checks when accessing global variables.

10.4.1.2 Heap

Accurately measuring the heap memory requirements was a recurrent problem in Chapter 7. This is because the version of the compiler used for those experiments did not distinguish between value and pointer types. As a workaround, we relaxed the constraints on pointer operations, but in this case the IHGC was no longer exact. Also, the function call stack was allocated as a contiguous chunk of memory. These design choices occasionally caused the IHGC to retain dead objects for longer than expected, so the memory requirements of our benchmarks were often disproportionate. For example, the *hexion* and *json_dumps* benchmarks have substantially higher heap memory requirements with the ARM processor alongside the IHGC compared to the same processor running the MicroPython software collector as shown in Figure 10.2.

The BeyondRISC system used for the experiments in this chapter fully addresses the problems outlined above. Features in the instruction set, such as using linked instead of contiguous stacks,

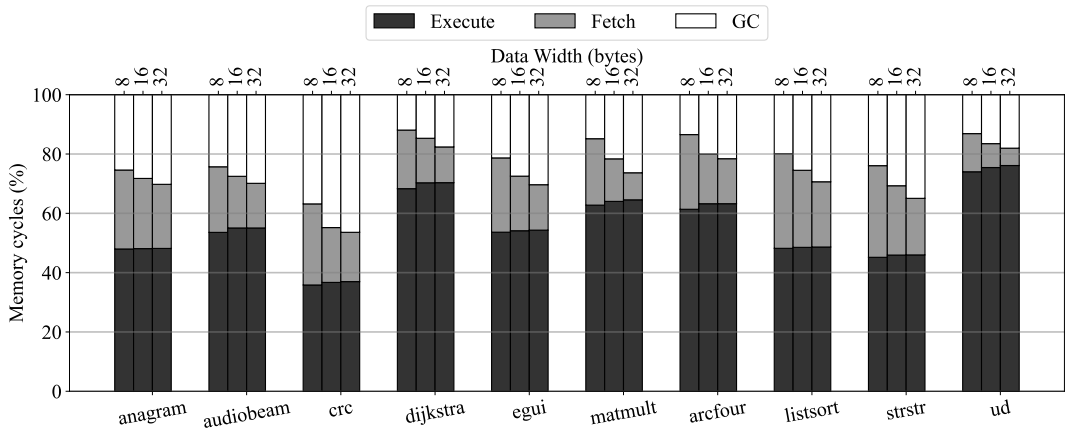
alongside our modified LLVM compiler ensure that the IHGC is once again exact and promptly reclaims garbage objects. As a result, Figure 10.2 shows a significant decrease in heap memory requirements for most of our MicroPython benchmarks in BeyondRISC compared to an ARM processor with either the IHGC or the software collector. *json_dumps* is the only program that shows a higher heap memory requirement in BeyondRISC than in MicroPython's software collector. This occurs because over 55% of the objects allocated by *json_dumps* are very small (often 4-20 bytes in size). Each object requires a header word, so the IHGC incurs higher memory overheads for this program in comparison to the software collector. However, the allocation patterns of *json_dumps* are unusual as the average object size in a modern programming language is 22-36 bytes [30, 184], so this overhead is not generally a concern in practice.

10.4.2 Characterizing Memory Cycles

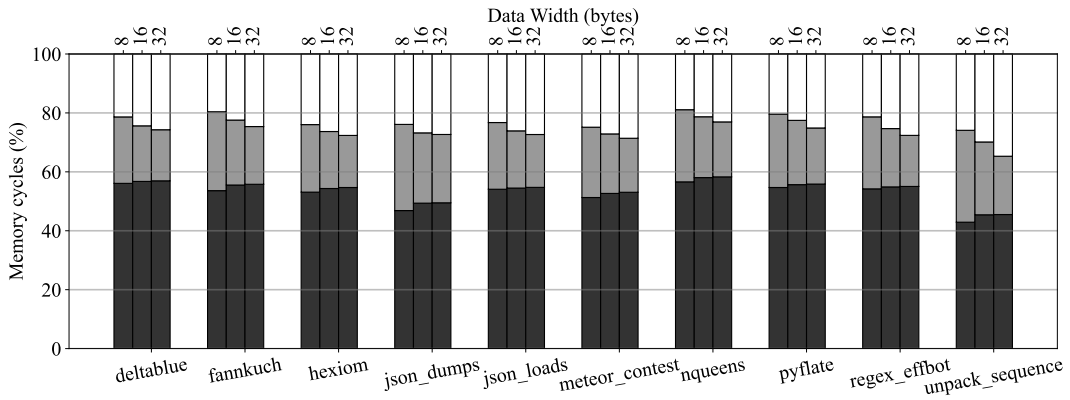
As discussed in Section 7.4.1, the proportion of memory cycles available to the IHGC is important to ensure optimal performance. In our system, there are two main operations that take memory cycles away from the IHGC: memory access instructions and instruction fetches. Figure 10.3 shows that up to 70% of memory cycles are used to execute memory access instructions, such as loads and stores. However, the proportion of memory cycles for instruction execution varies substantially depending on the benchmark. For example, *crc* uses less than 40% of memory cycles for instruction execution while about 70% are used in *ud* for the same purpose. More importantly, the larger programs, like *anagram*, *egui* and the MicroPython scripts, consume about 45-60% of memory cycles for instruction execution.

Instruction fetching consumes a considerable portion of the available memory cycles although this depends on the hardware and architecture. For example, we expect that the processor utilizes approximately 25% of the memory cycles for fetching if the data bus to the memory is 8 bytes wide, i.e. 64 bits, and each instruction is encoded in 16 bits as in BeyondRISC. In general, our experiments indicate that only 10-20% of memory cycles are used exclusively to fetch instructions as shown in Figure 10.3. This value is less than the expected 25% because the processor is pipelined, so fetches are often performed while other memory access instructions are executing as explained in Section 7.4.1. The IHGC could not have used the memory cycle regardless of these fetches, so they do not decrease the number of memory cycles available to the collector.

Naturally, increasing the amount of data loaded per fetch decreases the memory cycles dedicated to instruction fetching. For example, loading 16 bytes instead of 8 decreases the memory cycles for fetching by 10% for *crc*. However, further increasing the amount of loaded data does not have a substantial impact in the distribution of memory cycles. This is because branch instructions are likely executed before the data from the previous fetch is exhausted. Unfortunately, branches force the processor to fetch again, so memory cycles are taken away from the collector. Another consequence of increasing the amount of data loaded per fetch is that the IHGC completes mark-compact cycles faster. For instance, *deltablue* performs 15% more



(a) TACLe, BEEBS and *egui*.



(b) MicroPython.

Figure 10.3: Distribution of memory cycles in the BeyondRISC system as the amount of data loaded per instruction fetch increases. The data width is the size of the hardware memory bus and in these experiments corresponds to the number of bytes loaded per fetch operation. The program binary is exactly the same for all experiments as the software does not take advantage of the larger data widths to optimize performance; memory accesses for instruction execution are limited to 4 bytes.

collection cycles in approximately the same interval of time when the processor fetches 16 instead of 8 bytes. But performing collection cycles faster also increases the likelihood that the processor's pipeline needs to mark pointers loaded from memory. Therefore, the proportion of memory cycles used for instruction execution increases slightly when more data is loaded per fetch.

Comparing the results from Figure 7.1 and Figure 10.3, we observe that the IHGC has fewer memory cycles to operate when used alongside the BeyondRISC processor instead of the ARM Cortex-M0. This occurs because the BeyondRISC processor has a pipeline that is two stages longer than that of the ARM Cortex-M0. Multiple instructions can be in-flight in the longer pipeline at various stages of execution, so it is not guaranteed that the IHGC performs a state

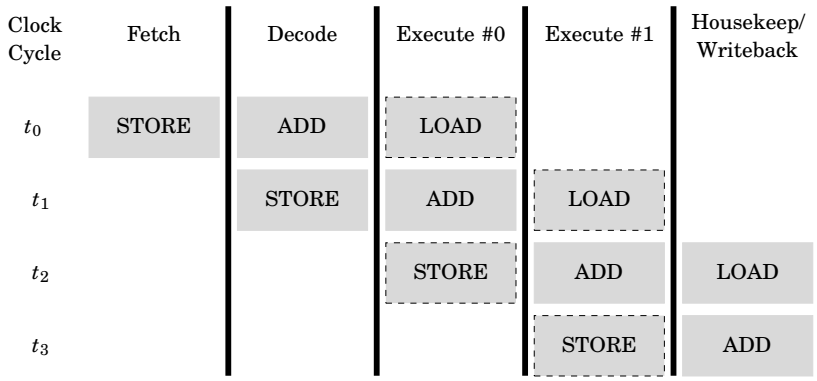


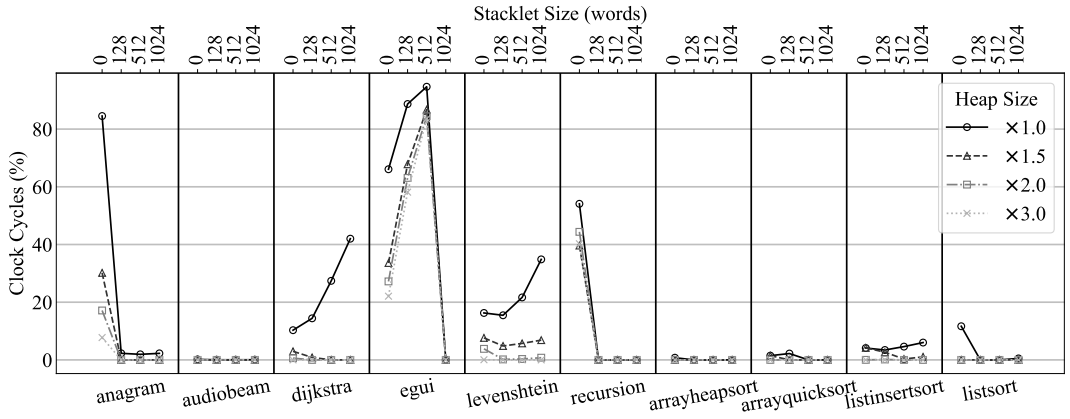
Figure 10.4: The IHGC is not guaranteed to operate when the BeyondRISC pipeline executes non-memory access instructions, like an `add`. The dashed boxes represent instructions requiring access to the directory or main memory when executing at that pipeline stage.

transition when a non-memory access instruction is executed. For example, Figure 10.4 shows that the IHGC cannot operate at cycle t_1 even though an `add` instruction is executing in the BeyondRISC pipeline because there is a `load` in the *execute #0* stage. Similarly, a `store` starts executing in the following cycle t_2 when the `add` is at *execute #1*, so the IHGC cannot operate at this time either. This situation does not occur in the ARM-Cortex-M0 as only one instruction can be at the execution stage during each memory cycle, thus it is guaranteed that the IHGC performs a state transition when a non-memory access instruction, like an `add`, is executing as long as an instruction fetch is not required. As a result, more memory cycles are used for instruction execution, and consequently less cycles are available to the IHGC, in the BeyondRISC processor compared to the ARM Cortex-M0.

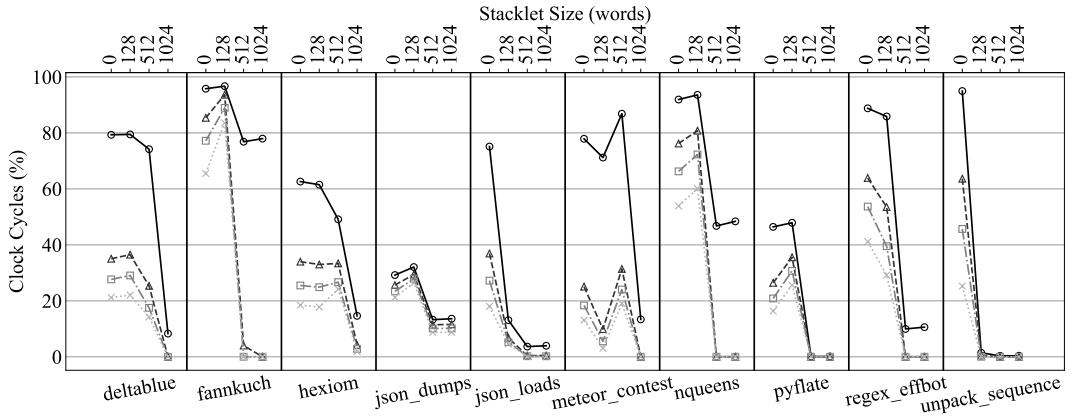
In summary, the IHGC has about 20-25% of memory cycles to operate when 8 bytes are loaded per fetch, but this can change significantly depending on the workload. Additionally, the proportion of memory cycles available to the collector increases when more data is loaded per fetch. However, fetching more than 16 bytes simultaneously does not provide sufficient benefits to justify the added hardware expense of a wider memory bus. Finally, the proportion of memory cycles available to the IHGC decreases as the length of the pipeline increases.

10.4.3 Pauses

The IHGC pauses the user's program when memory is allocated faster than it can be reclaimed. Eventually, the system will reach an out-of-memory condition because there is simply not enough free space or handles to satisfy an allocation. So the program is paused until the IHGC reclaims enough memory to meet the immediate allocation demands. In this section, we investigate how our design decisions affect pauses for benchmarks run on the BeyondRISC processor.



(a) TACLe, BEEBS and *egui*.



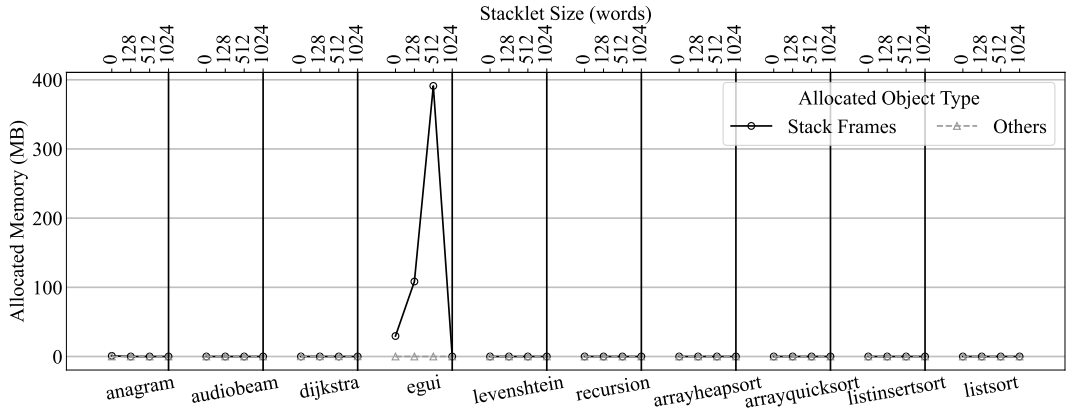
(b) MicroPython.

Figure 10.5: Proportion of clock cycles that the garbage collector pauses the user's program. For each benchmark, samples were taken for selected heap and stacklet sizes. A 0 words stacklet size corresponds to using linked stacks.

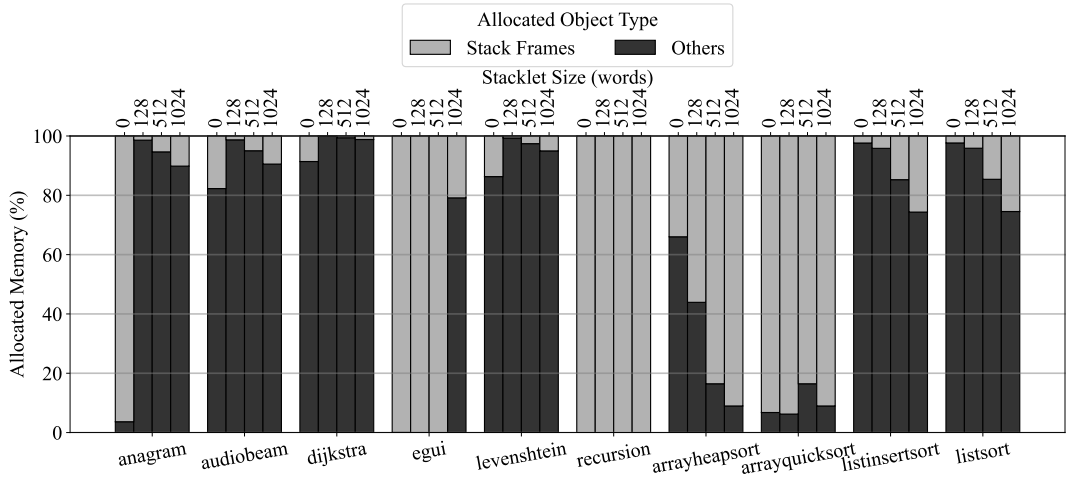
10.4.3.1 Linked Stacks

We first consider a BeyondRISC system using linked stacks as described in Section 8.2.3.2. Figure 10.5 shows the proportion of time that programs are paused due to collection operations as the heap size increases. Clearly, the system incurs high overheads when the heap size is set to the minimum amount of memory required for the program to operate correctly, i.e ×1.0 in the plot. This is because there is not sufficient excess memory to serve incoming allocation requests while the IHGC reclaims garbage objects. Increasing the heap size dramatically decreases pause times as we also discussed in Section 7.4.3. For example, the pauses for *deltablue* decrease from 80% to 35% when the heap size is increased by a factor of 1.5, although subsequent increases in memory size reduce pauses by a further 15% only.

Comparing the results in Figure 10.5 with our earlier data in Section 7.4.3, it is apparent



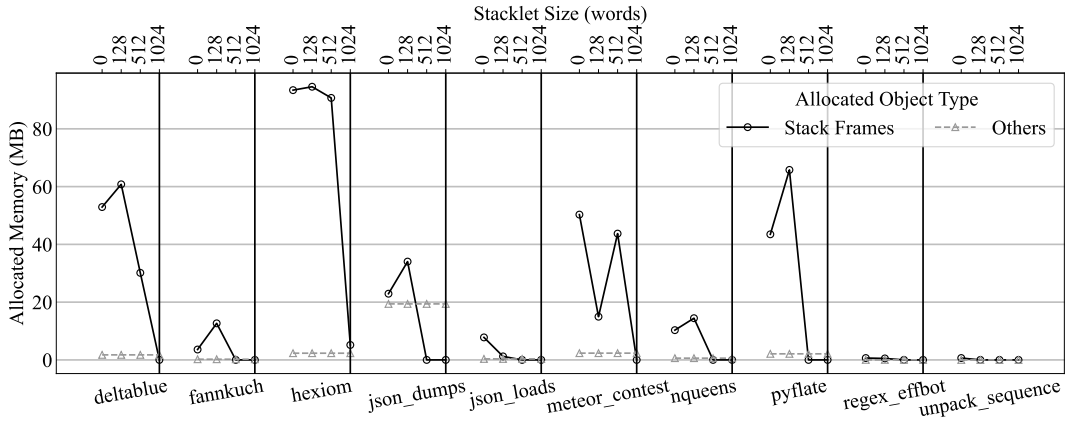
(a) Amount of memory allocated.



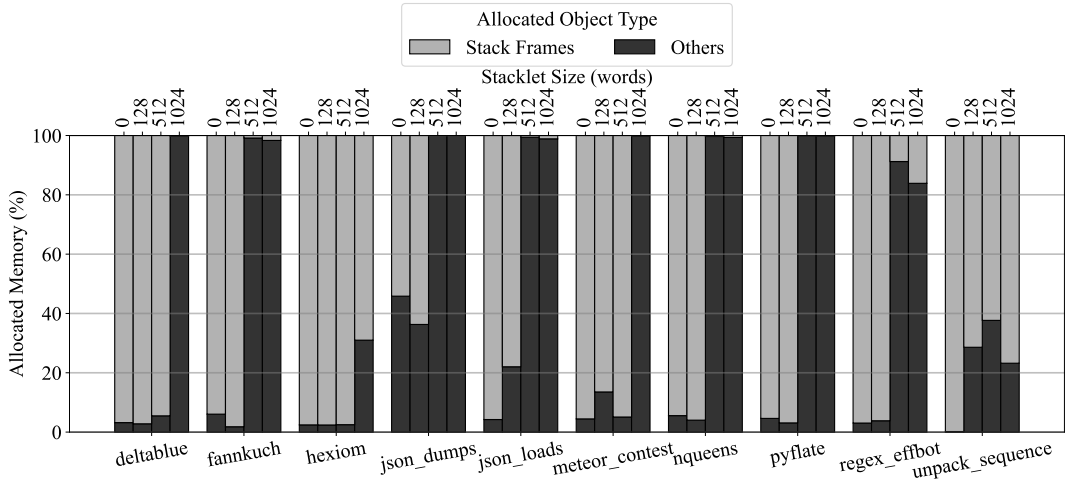
(b) Proportion of allocated memory for stack frames.

Figure 10.6: Memory allocations by object type in the TACLe, BEEBS and *egui* benchmarks as the stacklet size changes. A 0 words stacklet size corresponds to using linked stacks. The processor's data width is 8 bytes in all cases.

that the proportion of pauses in the BeyondRISC system is significantly higher than in the ARM system with the IHGC. Also, the BeyondRISC system requires a greater amount of heap memory to decrease pause times. For instance, increasing the heap by a factor of 4.0 in the BeyondRISC system running MicroPython reduces pauses to about 20% in the best case while a heap size increase by a factor of 1.5 only mostly eliminates pauses in the ARM system. The cause of this discrepancy becomes clear when we consider our benchmark's allocation patterns. Figure 10.6 and Figure 10.7 show that the vast majority of objects allocated by the programs suffering pauses correspond to stack frames. In fact, stack frames account for over 90% of all allocations in 12 of our benchmarks and the aggregated size of stack frame objects is 2-400 times larger than the heap space allocated for any other purpose.



(a) Amount of memory allocated.



(b) Proportion of allocated memory for stack frames.

Figure 10.7: Memory allocations by object type in the MicroPython benchmarks as the stacklet size changes. A 0 words stacklet size corresponds to using linked stacks. The processor's data width is 8 bytes in all cases

We can conclude that the excessive amount of memory allocated for stack frame objects when using linked stacks is causing the pauses as the IHGC is not sufficiently fast to reclaim the garbage, even when the heap size is increased fourfold.

10.4.3.2 Stacklets

The clear drawback of using a linked stack is that some programming patterns trigger high volumes of allocations that cause collection pauses. Stacklets are an alternative stack organization that addresses this problem. Each stacklet is a contiguous block of memory that can accommodate several stack frames as explained in Section 8.2.3.3. Upon entering a function, the callee places its stack frame on the current top stacklet if there is sufficient space; otherwise, the stack frame

is placed in a new, dynamically allocated stacklet.

Introducing stacklet support in BeyondRISC is relatively simple thanks to the IHGC. We add a `newstk` instruction that replaces `newm` when the program requests a new stack frame. `newstk` takes two input operands: a pointer to the current top stacklet, i.e. the *sp* register, and an integer indicating the required size (in words) of the new stack frame. Its output is a pointer to the beginning of the new stack frame. `newstk` uses the *sp* handle to load the size of the top stacklet from the directory. This is used, along with the *sp* offset and the requested stack frame's size, to determine whether there is sufficient space in the current top stacklet to accommodate the new stack frame. If so, a new pointer is constructed by decrementing the *sp* offset, otherwise a new stacklet is dynamically allocated. The size of stacklets is chosen by a configurable parameter that we set at boot-time, but larger stacklets are allocated when the program requests a stack frame larger than the configured size. Therefore, configuring the stacklet size to 0 words is equivalent to using a linked stack.

Stacklets have the potential to dramatically reduce collection pauses as shown in Figure 10.5. For example, pauses are reduced by 30% for *json_loads* when the heap size is 1.5 larger than the minimum required to run the program and the stacklet size is 128 words. This is because stack frame allocations were reduced by about 20% as shown in Figure 10.7. However, choosing the appropriate stacklet size for a program is difficult. An incorrect parameter often results in worse pause times as occurred in *egui* when the stacklets are 128 and 512 words in size. Two situations can cause this undesirable effect. First, larger stacklets are detrimental when the original cause of the pauses was object allocations unrelated to stack frames. For instance, most heap allocations in *levenshtein* and *dijkstra* do not correspond to stack frame objects, so increasing the stacklet size also increases pauses because the IHGC will take longer to mark and compact the stack. In this case, the proportion of memory allocated for stack frames changes little when the stacklet size increases, so larger stacklets are unnecessary and in this case counterproductive.

The second situation where larger stacklets introduce longer pauses occurs in *egui* and *fannkuch* as shown in Figure 10.5. The problem here is that collection work increases substantially when the stack size does not align well to a multiple of the stacklet size. For example, consider a program that calls a function A which uses a one word stack frame. Then A calls another function B that requires a 128 word stack frame and recursively calls A. If the stacklet size is 128 words, then each call to A will consume a full stacklet even though only one word is actually in use. In addition to the considerable space wasted, the IHGC has to mark and compact almost twice the number of words for those stacklets compared to using linked stacks. Therefore, collection cycles take longer and pauses increase accordingly.

Clearly, the best results in terms of pauses are observed in Figure 10.5 when the stacklet size is set to 1024 words. This is because in almost all benchmarks a single stacklet has sufficient space to accommodate the program's maximum stack size. In other words, the stacklet size is sufficiently large that the system effectively ended up using a contiguous stack. As explained

before, the benefit is that allocations in connection to stack frames are minimized, but the IHGC is no longer able to properly check access bounds when accessing memory in the stack. Also, the contents of stale stack frames below the *sp* are not zeroed, so there could be a delay in reclaiming garbage objects that increases the system's heap memory requirements.

In summary, an alternative stack arrangement like stacklets can be used to significantly reduce collection pauses at the expense of reduced bounds checking capabilities and the risk of higher memory requirements. Stacklets are straight-forward to implement alongside the IHGC by leveraging the object size information from the directory. However, choosing the appropriate stacklet size for a program is challenging; the wrong parameter can have detrimental effects on pause times.

10.4.3.3 Data Width

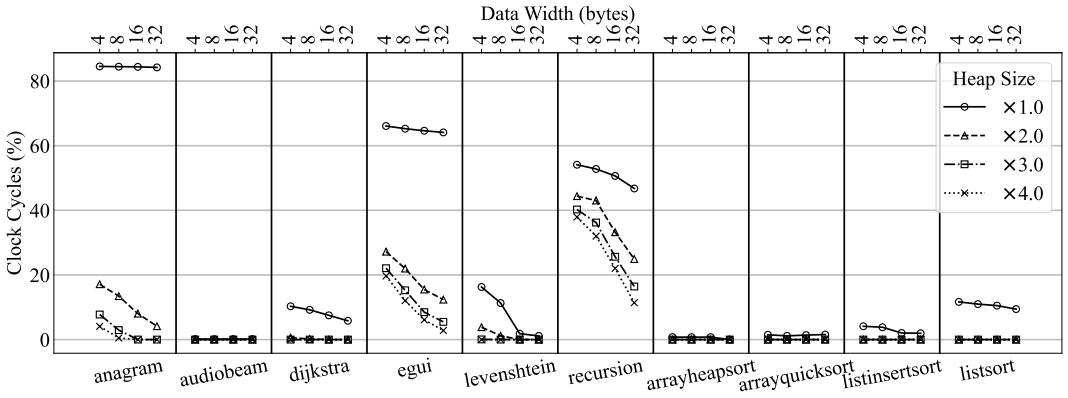
In this section, we discuss how the width of the main memory's data bus impacts pauses. The data width indicates the number of contiguous bytes that can be loaded or stored in a single memory cycle. We changed this parameter in Section 10.4.2 to increase the amount of data loaded per fetch in the hope of increasing the proportion of memory cycles available to the collector. We extended this idea to the IHGC by enabling it to compact and zero more than one word from the same object during a single state transition.

In general, increasing the data bus width decreases collection pauses as shown in Figure 10.8. However, the significance of the effect depends on the program. Programs, like *recursion* and *json_dumps*, where the compact stage of the collection cycle is relatively long, gain substantial benefits as the data width increases. For example, *json_dumps* allocates many small objects that the IHGC has to either copy or zero. So doubling the data width from 8 to 16 bytes reduces pauses by 10% although a further doubling does not have nearly as much impact because words from multiple objects cannot be copied or zeroed in a single memory cycle.

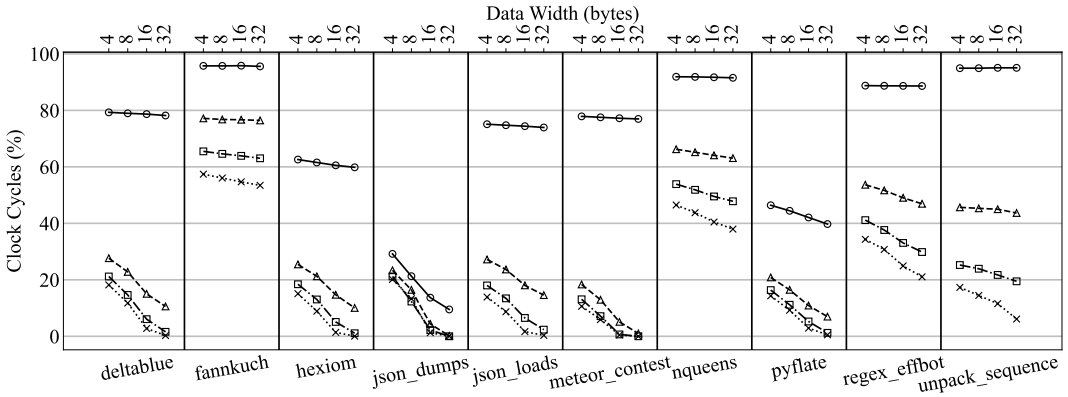
Programs where the collector spends most of its time marking do not benefit much from increasing the data width. For example, *fannkuch* and *nqueens* only experience about 3% reduction in pauses with each doubling of the data width. The reason is that we did not optimize the IHGC's marking stage to take advantage of the full data width. So objects are still scanned one word at a time during the marking stage, but this can be easily rectified by modifying the state machine to, for instance, skip over multiple words that do not contain pointers when scanning deep objects.

10.4.3.4 Caching Stack Frame Metadata

Computer architectures have designated registers to reference code (*pc*) and the stack (*sp*). Additionally, there are often architectural registers for referencing objects like global variables or constant pools. In BeyondRISC, these registers are *special-purpose* meaning that they can only be read or written by a small subset of instructions; special-purpose registers cannot be used in arbitrary computations. Figure 10.9 shows that instructions using a special-purpose



(a) TACLe, BEEBS and *egui*.

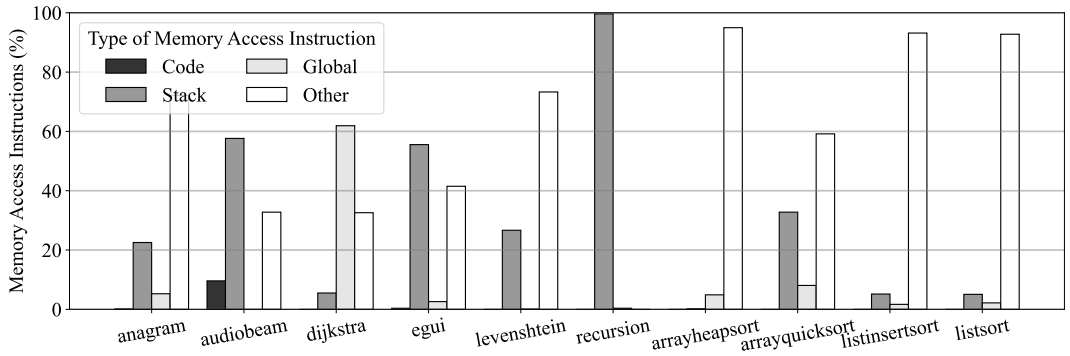


(b) MicroPython.

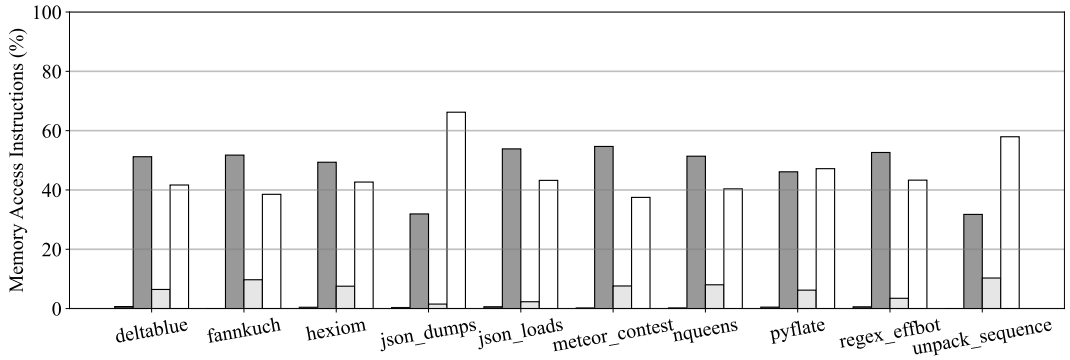
Figure 10.8: Proportion of clock cycles that the garbage collector pauses the user's program as the data width and heap size increase. A 4 byte data width indicates that the IHGC either loads or stores 4 bytes per state transition, but the processor loads 8 bytes per instruction fetch. For every other data width in the figure, i.e. 8, 16 and 32 bytes, the amount of data moved by the IHGC equals the number of bytes loaded per instruction fetch. Linked stacks are used in all cases.

register operand as a base pointer to access memory are executed very frequently. In particular, stack accesses using the *sp* account for 20-50% of the executed memory instructions in most of our benchmarks. This observation motivated a microarchitectural optimization that attempts to reduce the proportion of memory cycles used for instruction execution.

Memory instructions in an IHGC system are executed in two steps: the directory is accessed first, then the main memory. But the directory does not need to be read if the processor has a copy of the base address and size metadata for the object being accessed. In this case, the execution of the memory instruction takes one memory cycle less while the overall latency remains unchanged, so more memory cycles are potentially available for the IHGC. We can implement this optimization by caching object metadata in the processor's pipeline when it is first accessed; subsequent memory accesses to the same object simply use the information in the



(a) TACLe, BEEBS, *egui*.

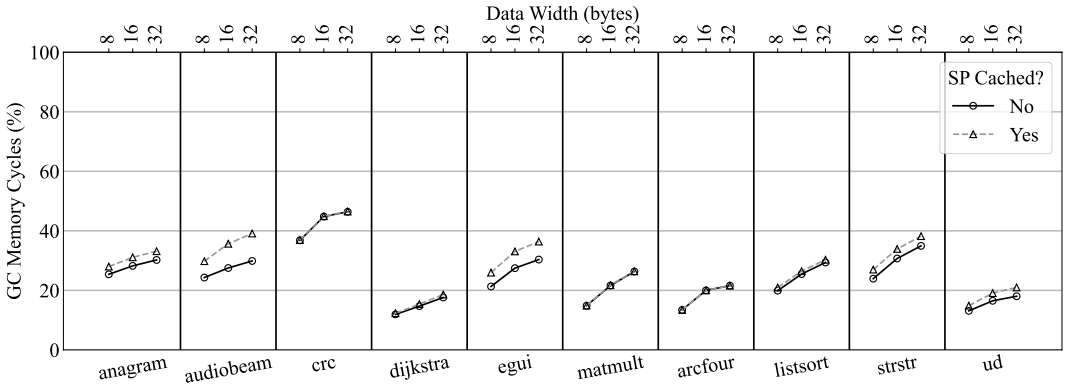


(b) MicroPython.

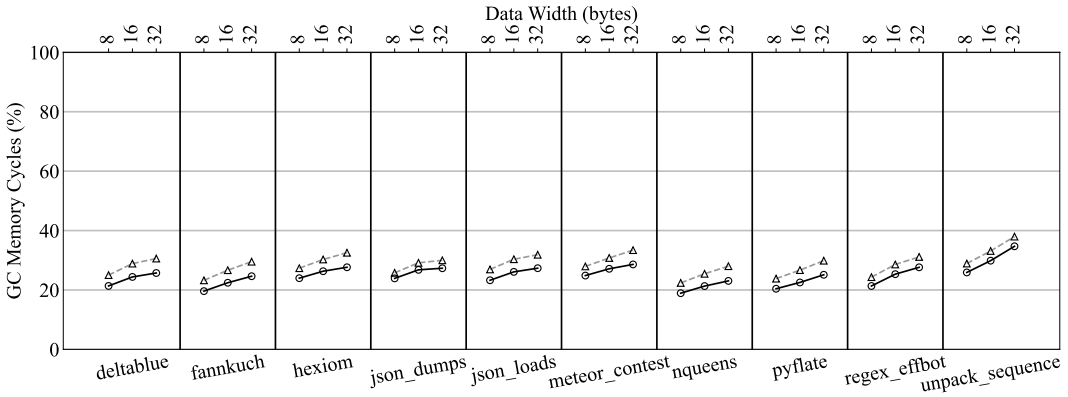
Figure 10.9: Distribution of memory access instruction segregated by type. 'Code' instructions implicitly use the *pc* register as the base pointer for the memory access. 'Stack' and 'Global' are analogous to 'Code', but for memory instructions accessing the top stack frame and global variables respectively. 'Other' refers to generic memory access instructions that use a general-purpose register as a base pointer.

cache instead of reading the directory. However, it is difficult to realize this optimization for a large number of objects because the cache contents need to be synchronized with the directory as the IHGC compacts. Therefore, we modified our SystemC model to only cache the metadata for the top stack frame, i.e. the object referenced by the *sp*.

The proportion of memory cycles available to the IHGC is shown in Figure 10.10 for BeyondRISC systems with and without caching the stack frame metadata. According to the plot, the IHGC has 0-10% more memory cycles in the system with the cache in comparison to the system without it. This marginal increase is expected because, as a rule of thumb, 30% of instructions correspond to loads and stores in a program. So only 15% of all instructions have the potential to free up memory cycles for the IHGC if we assume that 50% of memory access instructions use the *sp* as a base pointer because $30 \div 2 = 15\%$. Ignoring memory accesses at the *housekeep* pipeline stage, each instruction accessing the stack would free up one memory cycle out of the two it normally uses, so we can expect only $15\% \div 2 = 7\%$ more memory cycles for the IHGC in



(a) TACLe, BEEBS, *egui*.



(b) MicroPython.

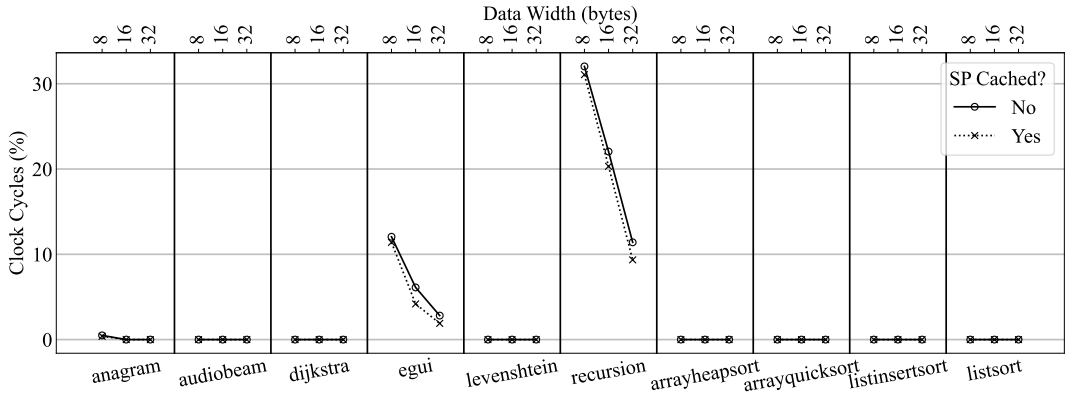
Figure 10.10: Proportion of memory cycles available for the IHGC when the stack object's base address and size, i.e. the object referenced by the *sp* register, is and is not cached.

this case.

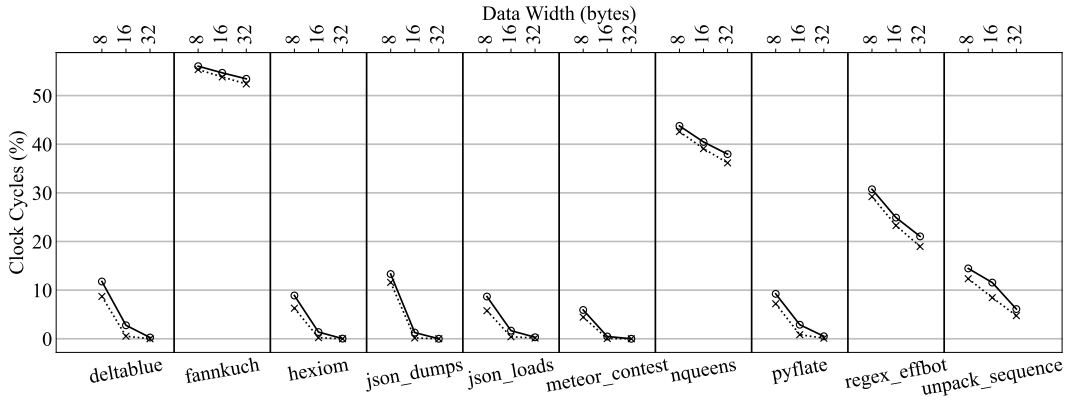
The extra memory cycles available to the IHGC in the system caching the *sp* metadata reduce pauses by up to 5% as shown in Figure 10.11. This gain is modest, but the caching optimization is very simple to implement. Also, it requires little extra hardware: a small register for the cache and a few gates for the control logic. It would be feasible to extend this scheme to cache metadata for references from a few other registers to achieve further reductions in pauses. As a result, the caching optimization is suitable for embedded systems despite the modest gain.

10.4.4 Pipeline Stalls

The microarchitecture of the BeyondRISC processor aims to minimize pipeline stalls. But sequences of memory access instructions can give rise to stalls due to contention when accessing the directory as explained in Section 9.3.4. Specifically, a stall occurs when a memory access instruction that accesses the directory during the *housekeep* pipeline stage is immediately fol-



(a) TACLe, BEEBS, *egui*.

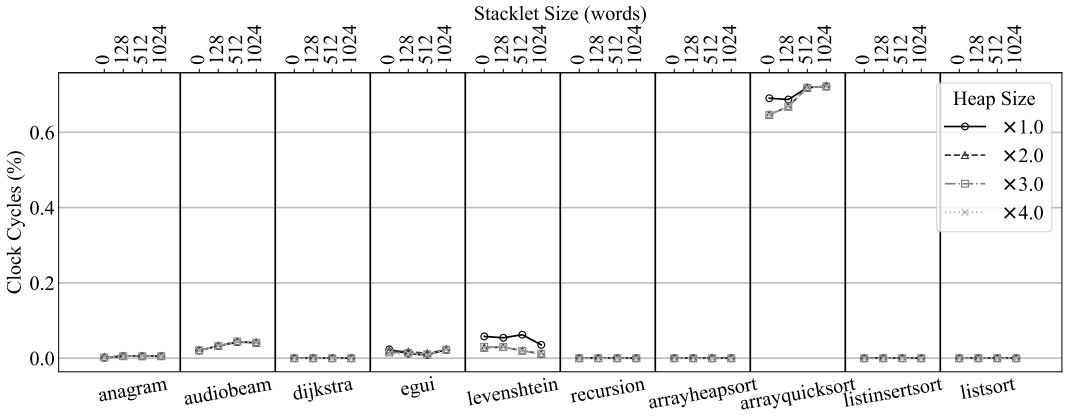


(b) MicroPython.

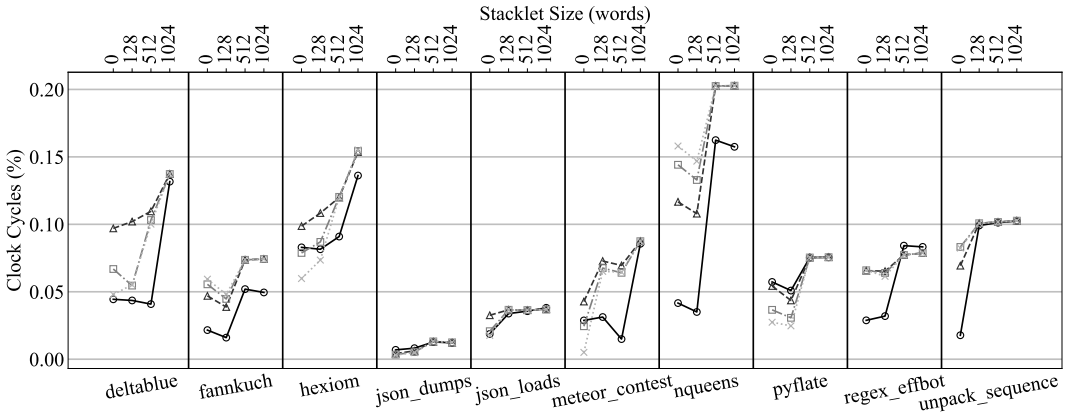
Figure 10.11: Proportion of clock cycles that the garbage collector pauses the user's program when the stack object's base address and size, i.e. the object referenced by the *sp* register, is and is not cached. Linked stacks are used in all cases.

lowed by a `load` or `newm` instruction. For example, the *housekeep* stage must set the *deep* flag for an object when a `store` writes a pointer into that object. The *housekeep* stage must also mark a pointer read from main memory while executing a `load` instruction. Both operations require accessing the directory. Therefore, the pipeline is stalled for one memory cycle when the `store` is immediately followed by the `load` because the directory is single-ported and can only serve one request per memory cycle.

The portion of clock cycles that the pipeline is stalled due to directory contention at *housekeep* is shown in Figure 10.12. Clearly, the delays are almost negligible as in all cases the stalls account for less than 1% of a program's run-time. Also, the proportion of stalled clock cycles increases slightly when collection pauses during allocations decrease. For example, Figure 10.5 shows that collection pauses are minimal when the stacklet size is 1024 words, and consequently, the proportion of stalled clock cycles increases in that case. This is because the total run-time of a



(a) TACLe, BEEBS and *egui*.



(b) MicroPython.

Figure 10.12: Proportion of clock cycles that the pipeline is stalled due to contention when accessing the directory. A stacklet size of 0 words corresponds to linked stack frames. The processor's data width is 8 bytes in all cases.

program decreases when collection pauses are reduced, so pipeline stalls related to directory contention are proportionally higher.

The performance impact of pipeline stalls related to directory contention is negligible because the instruction sequences likely to cause these stalls are rare. We identified three such sequences:

1. A `newm` followed by a `load` instruction that reads a pointer from main memory. This causes a stall as the *housekeep* stage must unconditionally read and write the directory record of the newly allocated handle as described in Section 9.3.4.4. However, this sequence of instructions rarely happens in our programs because `newm` is only executed in two cases: to allocate a stack frame and inside our `malloc` implementation. In the former, the `newm` is usually followed by stores as the program saves the registers to the stack. Inside `malloc`, the `newm` is immediately followed by a branch to return to the caller function.

2. A memory access instruction followed by `newm` can give rise to directory contention. The stall only occurs if the memory access instruction stores a pointer or loads an unmarked pointer while the IHGC is marking. Once again, this situation rarely occurs in practice because `newm` is generally the first instruction executed when entering a procedure, so it is preceded by a branch instead of a memory access instruction.

3. A memory access instruction followed by a `load` can also give rise to directory contention while the IHGC is marking. The stall only occurs if the first instruction loads an unmarked pointer or stores a pointer and the subsequent `load` reads a pointer from memory. But few coding patterns execute instructions that meet these conditions as programs manipulate data more often than pointers.

In summary, the run-time overheads of pipeline stalls due to directory contention are negligible because the conditions causing the stalls are rarely met. Therefore, optimizing these stalls yields few performance improvements at the expense of a potentially large hardware investment. For example, implementing the directory using dual-ported memories, as discussed in Section 9.3.2, eliminates the stalls and improves performance by less than 1%, but this change almost doubles the hardware cost of the directory.

10.5 Summary

We evaluated the IHGC through simulation using a SystemC model of the hypothetical microarchitecture for a BeyondRISC processor described in Chapter 9. For these experiments, we also used a production-quality compiler with a custom backend that takes into account the features provided by the IHGC. In comparison to the experimental setup from Chapter 7, the system used here implements an exact collector, so heap memory requirements are generally lower although programs that allocate large amounts of small objects incur memory overheads due to the object header.

The results indicate that a BeyondRISC system implementing linked stacks introduces substantial program pauses. This occurs because stack frames are allocated faster than they can be reclaimed even when the system is supplied with four times the minimum amount of memory required to run the program. These pauses can be completely eliminated using stacklets at the expense of reduced memory access bounds checking capabilities and potentially higher memory requirements. Also, the stacklet size must be carefully chosen to avoid introducing pauses. Finally, we observed that two hardware techniques, namely increasing the data width of the memory bus and caching object metadata, can be implemented to reduce pauses by up to 10%.

Part IV

Conclusions

CHAPTER 11

CONCLUSIONS AND FUTURE WORK

Modern programming languages are ubiquitous in web services and mobile phone apps. These languages are successful because they provide high-level data representation and control structures that improve programmer productivity and software reliability. This, coupled with the ever-increasing capabilities of large computers, like servers and mobile phones, has accelerated software development despite the significant performance drawbacks of modern languages when compared to older technologies like C and C++.

Modern programming languages underpin the rapid growth of the web and mobile software development industries. But other industries, like the Internet of Things (IoT), have seen limited adoption of modern languages despite their productivity and reliability benefits. This is because the vast majority of IoT devices are embedded systems with small processors and little memory that often have real-time requirements, so they cannot tolerate the overheads and run-time unpredictability of modern languages.

The overheads and real-time deficiencies mentioned above are not inherent to modern languages. These problems stem from the reliance on software garbage collectors to implement modern languages on embedded systems; the collectors are simply unsuitable for efficient implementation on conventional embedded computer architectures. So in this thesis, we investigated a shift in architecture towards hardware garbage collection. We proposed an Integrated Hardware Garbage Collector (IHGC) that demonstrates how a collector can be fully implemented in hardware alongside an embedded processor to minimize the run-time and memory overheads inherent to software collectors. We also proposed a real-time analysis technique that shows how the timing properties of a hardware garbage collector can be analyzed to guarantee that hard real-time requirements are met. Thus, this thesis positions hardware garbage collection as a viable alternative to support modern programming languages in embedded systems.

In the remainder of this chapter, we state our main contributions, propose future research

directions for hardware garbage collection and conclude the thesis.

11.1 Contributions

An important contribution of this thesis is that it ties together research from hardware design, garbage collection and real-time analysis to develop, implement and evaluate a hardware garbage collector with real-time capabilities. To achieve this, we proposed the IHGC and explained its main design choices in connection to the literature. We also presented a timing model of the IHGC along with a hard real-time analysis technique to guarantee that the collector never pauses the user's program. Finally, we described the design of a practical embedded system with the IHGC and evaluated its costs and performance.

11.1.1 Design of the IHGC

We proposed the IHGC, a hardware garbage collector that operates in the background concurrently with the processor. The design uses three techniques to ensure that the collector does not normally pause the user's program. First, the IHGC relies on tags to accurately distinguish words with pointers from those with regular values in memory and the registers, so the collector is exact and its timing behavior can be statically analyzed. Second, the IHGC implements a mark-compact algorithm fully in hardware as a small state machine having the property that each state transition can be completed in a single memory access cycle. This enables collection operations to be seamlessly interleaved with memory accesses from the processor without pausing the program. And third, the IHGC implements an indirection through handles that facilitates efficient coordination between processor and collector to maintain program correctness since both operate concurrently.

The IHGC relies on a mechanism, conceptually similar to Baker's read barrier [32], although fully implemented in the hardware instead of software, to guarantee that all reachable objects are marked and retained during a collection cycle. Also, the IHGC is tightly coupled with the processor, in a similar fashion to Meyer's hardware collector [113], to ensure that coordination operations between collector and processor are performed efficiently in hardware and seamlessly from the programmer's point of view. These design choices eliminate program pauses and make the collector's timing behavior predictable, so the IHGC can be used for real-time applications.

Finally, a major contribution of our work is an investigation on how alternative choices in the design of a hardware garbage collector impact its timing properties. For example, we compared the benefits and drawbacks of the four basic garbage collection algorithms, i.e. mark-sweep, mark-compact, copying and reference counting. We also discussed how using Brooks' or Yuasa's write barriers [42, 195], as opposed to Baker's read barrier, complicate the processor's microarchitecture and hard real-time analysis with the IHGC.

11.1.2 Real-time Analysis with the IHGC

We developed an analysis technique that estimates the amount of memory required by an embedded system with a hardware garbage collector to run a given hard real-time program without pausing. Our formulation is loosely based on Robertz and Henriksson's proposal although adapted to the context of a hardware garbage collector [142], i.e. the IHGC, that operates whenever the memory is not used for instruction execution. Our contributions also include a timing model of the IHGC along with an automated software tool to statically estimate a program's allocation rate and memory bandwidth usage, which are inputs required for our analysis technique. Therefore, this thesis provides the necessary elements for programmers to use hardware garbage collection in real-time applications.

11.1.3 The IHGC in a Practical Embedded System

We considered the design of a practical embedded system with the IHGC from three perspectives. First, we investigated the impact of a hardware garbage collector on RISC Instruction Set Architectures (ISA) with an emphasis on maintaining compatibility with existing programs. To achieve this, we proposed semantic changes to instructions, like add, subtract and branches, to ensure that existing compilers generate code that does not perform invalid operations on pointers. Additionally, we proposed mechanisms to efficiently support other architectural features needed in an embedded system with the IHGC such as the function call stack, I/O handling, exceptions and interrupts. Support for these features in the context of a hardware garbage collector has rarely been considered in the literature.

It is unlikely that a system like the IHGC would be widely adopted if we could not demonstrate broad compatibility with existing embedded programs and modern programming languages. Therefore, our second contribution is an assessment of the amount of effort required to port existing open-source projects, like FreeRTOS and MicroPython, to run on an embedded system with the IHGC. Through this work, we showed that the required porting effort is minimal for the majority of programs and running existing software on the IHGC system often uncovers bugs.

Finally, an important contribution of our work is the microarchitecture of the IHGC alongside a pipelined embedded processor. We discussed the main challenges in the design while taking into account the features of a modern fabrication technology suitable for embedded systems.

11.1.4 Evaluation of the IHGC

We evaluated the IHGC's hardware costs using a hypothetical model of the collector alongside a pipelined processor. The results indicate that an embedded processor with the IHGC fabricated at a 45 nm process node has an estimated clock speed of up to 350 MHz which is comparable with that of state-of-the-art embedded processors. Also, we estimated the IHGC's memory overheads to be 10-20% due to the directory and tag bits.

Another contribution of this thesis is an evaluation of the IHGC in terms of performance and pauses. To achieve this, we simulated an ARM Cortex-M0 processor alongside the IHGC. Our experiments indicate that, compared to a processor without the IHGC, our system has comparable or better performance when running C programs and is 1.5-7 times faster when running scripts written in Python, a modern programming language. These performance improvements are due to the IHGC's ability to run memory management operations in the background, so the processor can be fully utilized to run the user's program. In contrast, programs are eventually paused when running on a system without the IHGC because the processor must be used to perform memory management operations. Additionally, we empirically measured the IHGC's pauses on simulated ARM and BeyondRISC systems. We found that the IHGC introduces few pauses when the heap is minimally increased, e.g. by a factor of 1.5 beyond the minimum operational requirement. The experiments also show that features, like the memory's data bus width and the function call stack organization, significantly impact pauses.

We showed our real-time analysis in practice and demonstrated that programs do not pause when running on a simulator of the ARM Cortex-M0 processor. The study showed how parameters, like the clock speed, affect the estimated memory requirements to eliminate collection pauses. We also discussed why existing embedded benchmarks are unsuitable to evaluate novel real-time garbage collectors.

11.2 Future Work

This thesis shows that it is feasible to use hardware garbage collection to efficiently support modern languages in real-time embedded systems. But further work is needed to develop a software ecosystem around the IHGC and scale up the design to work alongside larger computer systems. Therefore, we suggest following up the investigation from this thesis in three main directions.

11.2.1 Software Ecosystem

A large portion of our work was dedicated to exploring the IHGC's impact on software and compilers. We demonstrated that existing software can be easily ported to an embedded system with the IHGC when using the LLVM compiler and a custom backend. But the LLVM backend developed as part of this thesis is only a prototype and needs improvement before it can be used in production settings. Additionally, existing programs that are simply ported to an IHGC system could see significant performance improvements if they are rewritten under the assumption that there is a hardware garbage collector. For example, memory access bounds and other checks that are normally implemented in software could be eliminated since the IHGC performs them in hardware without fetching and executing additional instructions.

11.2.1.1 Compilers

We developed a prototype compiler for BeyondRISC by extending LLVM with a new backend. This compiler is suitable for our embedded system because it emits code that does not violate the IHGC's type constraints. However, LLVM discards type information very early on the compilation process, so our prototype backend does not always emit the most efficient code possible. The linker that we used in the compilation process of our experimental programs is also a prototype and does not implement many optimizations. Thus, a future research direction is to explore how the existing compilation toolchain could be improved to take advantage of the full set of features offered by the IHGC.

11.2.1.2 Modern Language Implementation

Modern programming languages, like Python, C#, Go and JavaScript, are implemented using a mixture of compilers and runtime environments. But these implementations often have software garbage collectors baked in, so many design decisions are motivated to facilitate memory management. Additionally, existing runtimes and compilers implement safety and reliability mechanisms, such as memory access bounds checks, in software as this is not natively supported in commodity hardware. Obviously, these assumptions are incorrect in a system with the IHGC, and in fact, can often incur overheads. Therefore, there is a need to investigate how modern languages can be efficiently implemented in an embedded system with the IHGC.

11.2.1.3 Benchmarks

This thesis showed that there is a need for an ecosystem of benchmarking software for embedded systems. Existing benchmarks are mostly focused on measuring the run-time performance and power consumption of embedded processors with very small kernels and C programs. But these benchmarks ignore that the applications of embedded systems have evolved substantially in the last few decades. Currently, these devices are required to run complex software such as networking stacks and machine learning algorithms. As a result, there is a need to develop new benchmark suites that capture the requirements of practical, modern embedded software and are widely accepted by the industrial and research communities.

11.2.2 Real-Time Analysis

We presented a real-time analysis technique along with a timing model of the IHGC that allows running programs without pauses. Both of these require input values statically extracted from the program, so we developed an automated software tool to help with this task. However, the tool is a prototype: it only automates the estimation of two parameters. Also, the analysis technique introduced in this thesis is for hard real-time systems, but many embedded systems have soft real-time requirements instead.

11.2.2.1 Static Program Analysis

Our static analysis tool currently automates the process of estimating a program's allocation rate and memory bandwidth usage. But the tool relies on programmers to input bounds for loops and memory allocations. Also, the tool does not estimate other parameters required by our hard real-time analysis, like the live size and the number of pointers. Future work should focus on overcoming these limitations, at least partially.

11.2.2.2 Soft Real-Time Garbage Collection

The analysis technique presented in this thesis is suitable for hard real-time systems that are always required to meet timing deadlines. But a large number of embedded systems do not have such strict timing constraints. These soft real-time systems are expected to consistently meet their deadlines, but occasionally missing one is not catastrophic. Thus, it would be ideal to develop alternative soft real-time analysis techniques that enable estimating, for example, the likelihood of the system meeting a deadline. This analysis would help programmers to improve the quality of service provided by an embedded system. In addition, changes to the IHGC's design can be introduced to optimize soft real-time systems. For instance, the IHGC could be extended with a hardware mechanism that allows an operating system to forcibly increase the amount of memory cycles available to the collector at the expense of introducing run-time delays for the user's program. This would enable an embedded system to avoid long pauses by increasing its collection rate when the IHGC is under pressure.

11.2.3 Scaling Up the IHGC

This thesis studied hardware garbage collection in the context of embedded processors that have relatively small, on-chip memories. We consider that a similar idea can be investigated for larger systems that have more complex memory architectures and features like caching.

11.2.3.1 Caching

Modern embedded processors are being used alongside large memories to fulfil the ever-increasing storage requirements of embedded software. The long latencies to access these memories have motivated the use of caching to mitigate performance penalties. For example, the ARM Cortex-M7 processor, released in 2014, supports up to one level of instruction and data caches [15]. However, the IHGC design presented in this thesis does not account for caching.

Future research should investigate how the IHGC can work with caches. We consider that there are three main points that must be considered to achieve this. First, garbage collection should not cause large amounts of data to be evicted from the cache while compacting, so the cache must be indexed using object handles rather than memory addresses as it occurs in traditional memory architectures. Second, cache lines must accommodate an object's contents as well as

its metadata, i.e. the size, deep flag, etc, to reduce the processor's reliance on the directory, for example, when checking memory access bounds. And third, the IHGC must not cause data to be loaded onto the cache during the collection process as this can evict the user program's data thus decreasing performance.

11.2.3.2 Distributed Memory

It is often desirable to construct embedded systems with *distributed memory* architectures on-chip. These systems have multiple processors operating in parallel each with its own private memory. The processors typically communicate over a Network on Chip (NoC). For example, the XMOS xCORE-200 is an embedded system with 16 processors that is intended for IoT applications [193]. The collector's task in such systems is to reclaim dead objects in all private memories, so a solution is to include an instance of the IHGC alongside each processor. However, the collector needs to ensure that objects referenced from outside its own private memory, e.g. another processor's registers, are retained. Thus, a future research direction is to investigate how the multiple IHGC instances in a distributed embedded system can be coordinated to prevent objects from being reclaimed by mistake.

11.2.3.3 Multi-Core Systems

Large multi-core processors are commonly used for applications like web servers and mobile phones. These systems consist of multiple processors that all have direct access to a *shared memory*. Also, there are often deep memory hierarchies with multiple levels of cache and complex coherency mechanisms to enforce a consistent view of the memory. As the number of processors increases, it is important to investigate whether there are sufficient spare memory cycles for collection operations when implementing a hardware garbage collector, like the IHGC, in such systems. Additionally, virtual memory usually features in large multi-core processors as it allows implementing key functions of modern operating systems like process management. However, virtual memory already imposes an indirection similar to the IHGC's on every memory access. Therefore, a future research direction is to investigate how a hardware garbage collector could be implemented in large multi-core systems.

11.3 Conclusions

Researchers have long been preoccupied with developing software garbage collectors that can meet real-time requirements. However, these collectors impose high overheads that are unacceptable for small embedded devices. This thesis is an effort to remedy the problem by exploring a shift in architecture from software to hardware garbage collection with a focus on efficiently supporting modern programming languages in embedded systems. We showed that this approach delivers substantial performance improvements with few hardware overheads when compared to software

collectors. We also demonstrated that our hardware collector has timing properties that make it ideal for use in real-time applications. We conclude that hardware garbage collectors are a viable alternative to designing real-time embedded systems that can efficiently support modern programming languages.

Appendices

APPENDIX A

INTEGRATED HARDWARE GARBAGE COLLECTOR STATE MACHINE

This appendix contains the **speccam** specification of the Integrated Hardware Garbage Collector's (IHGC) state machine. **speccam** is a notation language to facilitate the specification and behavioral descriptions of the hardware at a high level. **speccam** is based on the **occam 2** programming language developed at INMOS Ltd [80].

The contents of this appendix are a detailed specification of the system-level description of the IHGC in Chapter 5. This specification does not describe the microarchitecture of the state machine. However, it can be used as a starting point for a working implementation of the IHGC in either a Hardware Description Language (HDL), such as Verilog, or a simulation environment using any programming language. Only minor details, like error detection and containment, have been omitted or simplified in this specification as their implementation depends on the processor's microarchitecture and the instruction set.

A.1 Notation

speccam programs are built from processes. This specification uses five kinds of processes.

skip: A process that starts, performs no action and terminates.

Assignment: Assigns the result of an expression to a variable and then terminates. The assignment operator is ←.

Parallel: Lists multiple processes that are performed concurrently. All processes of a parallel block start simultaneously, and proceed together. The parallel block terminates when all its processes terminate. Processes in a parallel block are separated by the & operator.

Replicator: Produces a fixed number of similar processes that are performed in parallel. Replicators use the syntax i **is** s **for** n **in** $P(i)$ where i is the index value, s is the first index, n is the number of times that the process is replicated and P is the replicated process.

Conditional: Lists multiple processes each guarded by a boolean expression. The boolean expressions are evaluated in the order they are listed within the conditional. If a boolean expression evaluates to true its associated process is performed, and the conditional terminates [80]. The contents of a conditional process are enclosed by { and } and each case is separated by the | operator. A conditional case consists of a boolean expression followed by the ⇒ operator followed by a process.

Processes can be grouped using procedures and functions. Functions must be terminated by a **return** process that evaluates an expression and returns its result to the caller. Expressions can be abbreviated at the beginning of a function or procedure or at the beginning of the program using the keyword **val**. Similarly, variables can be aliased to a different name using the keyword **alias**. A name may alias one of an arbitrary number of elements in a conditional expression. For example, the declaration

```
alias n is ( a ⇒ b
           | c ⇒ d
           ):
```

aliases the name n to variable b if a evaluates to true, otherwise to variable d if c evaluates to true.

bit is the basic data type in **speccam**; it can be used to declare a scalar variable or multi-dimensional arrays. Arrays are declared by prefixing the type with [followed by the array size followed by]. The array size must be a constant integer. **bool** is the type for boolean values.

A.2 Definitions

The IHGC is tightly coupled with the processor and does not normally pause the user's program. The IHGC and processor both share access to the main memory that we define as an array of msz words each containing bpw bits per word. The IHGC also has read-only access to an array *regs* with rsz elements that represent the machine's register file. A separate *regsbuf* buffer, that the processor cannot modify, is used to make a copy of the register file contents at the start of the collection cycle.

```
[msz][bpw]bit mem:
[rsz][bpw]bit regs:
[rsz][bpw]bit regsbuf:
```

The IHGC uses exact garbage collection as it distinguishes pointers using type information. Every word in memory and the registers has a 1-bit tag that indicates whether it contains a

Identifier	Description
addr	The object's base address in physical memory
size	The object's size in bytes
list	Space to store a handle to chain entries into linked lists
mark	The garbage collection mark flag
deep	Flag indicating whether the object is deep

Table A.1: Metadata items stored in each directory entry.

pointer or a data value. The tag is stored in the most significant bit of a word, i.e. at index $bpw-1$, and can be easily accessed using the syntax $w.ptr$. For example, $mem[0].ptr$ contains the tag of the word at index 0 in memory. The remaining $bpw-1$ bits in a word contain the last value stored at that location.

Words containing pointers have their tag bits set. The remaining bits in the pointer word are conceptually split into handle and offset. These components can be conveniently accessed using the syntax *w.handle* and *w.offset* respectively. The handle uniquely identifies the referenced object and is an index into the directory. We define the directory as an array of dsz entries each containing a configurable amount of bits bpe depending on the system's requirements.

```
[dsz][bpe]bit dir:
```

Each directory entry contains enough space to accommodate the metadata items listed in Table A.1. The individual components of a directory entry can also be easily accessed using the '.' syntax along with the identifiers in the Table A.1. For instance, *dir*[1].*deep* refers to the deep flag of the entry at index 1. A `NULL` pointer is defined for convenience with the tag set and the handle and offset zeroed; therefore, the element at index 0 in the directory is unused.

The collector uses several registers to store its internal state. Some of these are also used by the processor to perform mark on load and redirection operations while executing memory access instructions. The *state* register contains the 4-bit identifier of the current state. Each individual state, its **speccam** identifier and transitions are shown in Figure A.1.

```
[4]bit state:
```

The IHGC compacts live objects by copying them toward the lowest memory address, which implies that the free memory is similarly clustered at the high memory addresses. The *heappoint* is the register that indicates the boundary between the live and free memory clusters; it points to the lowest location in physical memory that is free. A related register, *livesize*, stores the aggregated size of all objects marked during the current collection cycle. Both the *heappoint* and *livesize* registers must have as many bits bpa as the *addr* component in the directory.

```
[bpa]bit heappoint:
[bpa]bit livesize:
```

A *free* variable contains the handle of the directory entry at the head of a linked list of free handles. The entries are chained using the *list* component from the directory. Another list of handles, *next*, is used by the collector during a cycle to maintain a record of the live and deep objects that have been marked but are yet to be scanned. Both the *free* and *next* registers have as many bits bph as the handle component of a pointer.

```
[bph]bit free:
[bph]bit next:
```

The register *regindex* records the index of the register that will be processed in the next state transition when the IHGC is scanning the roots. Clearly, this register must contain enough bits $bpri$ to represent the index of every element in the register file $regs$ without overflowing.

```
[bpri]bit regindex:
```

The IHGC uses six registers to process objects at both the mark and compact stages of the collection cycle. *obj* contains the handle of the object being processed. For example, *obj* is set to the handle of the object being marked during the marking stage. The related registers *src* and wsz contain an object's physical address, i.e. the *addr* component from the directory, and its size respectively. The *index* register tracks the index of the word currently being processed by the collector. The *dest* register records the new memory address of an object being relocated during the compact stage. And the *buffer* register is a temporary buffer that stores the word being copied while compacting.

```
[bph]bit obj:
[bpa]bit src:
[bpa]bit dest:
[bpa]bit wsz:
[bpa]bit index:
[bpw]bit buffer:
```

Finally, the *alloc* and *oomcount* variables are used to detect basic out-of-memory errors, e.g. when the program's live size exceeds the memory size. *alloc* has two components, the first is *alloc.waiting* which signals whether the processor is currently paused waiting for an allocation to complete. The second, *alloc.wsize* is only valid when *alloc.waiting* is set. It indicates the minimum number of free bytes required to fulfil the pending allocation request that caused the collection pause. The IHGC counts the number of collection cycles that have been performed, using the *oomcount* register, after an allocation gave rise to a pause, i.e. when *alloc.waiting* is set. An out-of-memory error is raised when two collection cycles are performed, so *oomcount* is effectvely a 1-bit flag. This mechanism is only a placeholder to detect and report basic out-of-memory conditions in this specification. But it is unsuitable for complex multi-threaded embedded systems as the *alloc* and *oomcount* registers are unable to track pauses caused by more than one process.

```
[bpa + 1]bit alloc:
```

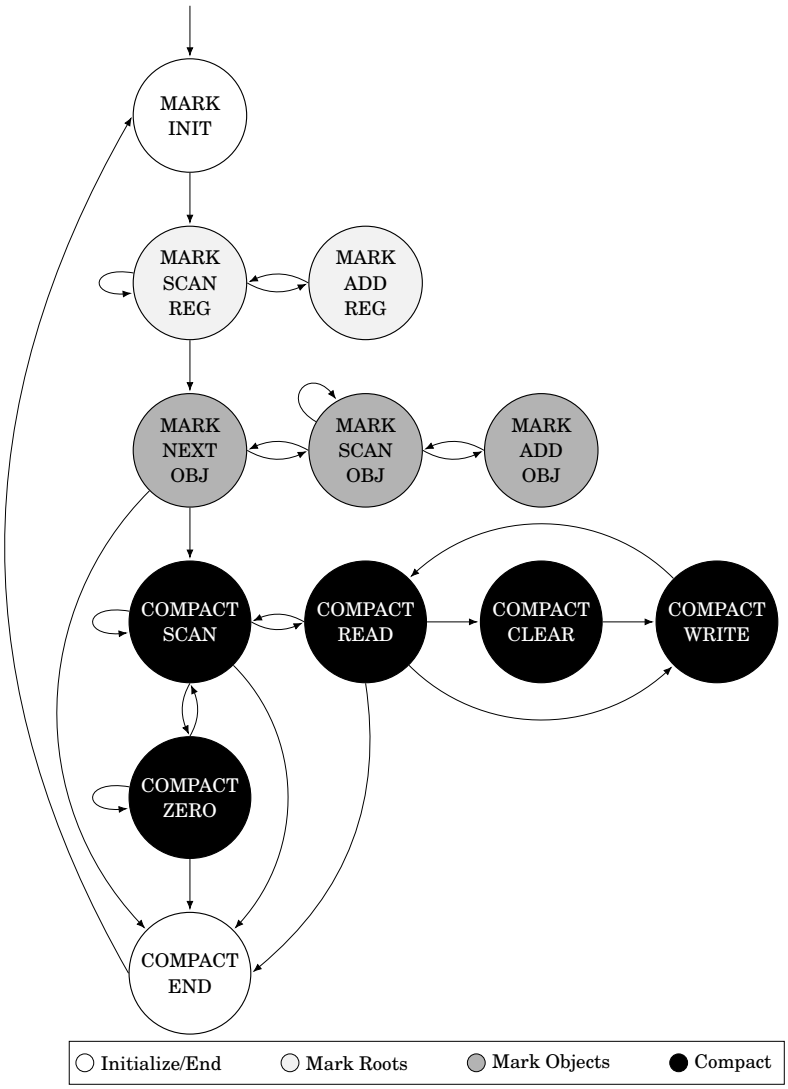


Figure A.1: IHGC state machine corresponding to the **speccam** specification. The state names in the image are the same as the state identifiers used throughout the code with '_' replaced by spaces.

```
bit oomcount:
```

A.3 Initialization

The IHGC is initialized by a simple state transition that copies the register file contents into a buffer of shadow registers. The *regindex* register is also set to 0 and the collector transitions to the mark roots stage.

```
procedure MarkInit():
```

```
{ i is 0 for REGS_COUNT in regsbuf[i] ← regs[i]
& state, regindex ← MARK_SCAN_REG, 0
}
```

A.4 Mark Roots

The IHGC scans the contents of the shadow registers during the mark roots stage. Each pointer found in the shadow registers is processed for marking using the MARK_ADD_REG state. Unmarked objects referenced by pointers in the register file are marked and their size is added to the *livesize* register. The newly marked object's handle is also inserted at the front of the *next* list if the object is deep.

```
procedure MarkScanReg():
    val indexoutbounds is regindex = rsz:
    val indexinbounds  is regindex < rsz:
    val isnotnull      is regsbuf[regindex] ≠ NULL:
    val reghasptr      is regsbuf[regindex].ptr and isnotnull:
{ indexinbounds  ⇒ { reghasptr ⇒ { obj     ← regsbuf[regindex].handle
                                   & state, regindex ← MARK_ADD_REG, regindex + 1
                                   }
                   | not reghasptr ⇒ regindex ← regindex + 1
                   }
| indexoutbounds ⇒ state ← MARK_NEXT_OBJ
}


procedure MarkAddReg():
    val nlivesize is livesize + wordcount(dir[obj].size) + 1:
{ not dir[obj].mark ⇒ { not dir[obj].deep ⇒ { dir[obj].mark, livesize ← true, nlivesize
                                              & state ← MARK_SCAN_REG
                                              }
                      | dir[obj].deep    ⇒ { dir[obj].list, next ← next, obj
                                             & dir[obj].mark, livesize ← true, nlivesize
                                             & state ← MARK_SCAN_REG
                                             }
                      }
| dir[obj].mark     ⇒ state ← MARK_SCAN_REG
}
```

A.5 Mark Objects

The IHGC processes the *next* list in search of pointers to unmarked live objects. Each handle is popped from the head of the *next* list and the contents of its corresponding object are scanned one word at a time. Unmarked objects referenced by pointers in deep objects are marked and

their size is added to the *livesize* register. The newly marked object's handle is also inserted at the front of the *next* list if the object is deep. The mark objects stage terminates when all live and deep objects have been marked and scanned; the *next* list is empty and the IHGC transitions to the compact stage.

```
procedure MarkNextObj():
    val hasnext      is next ≠ NULL.handle:
    val isheapempty  is heappoint = 0:
    val needscompact is (not hasnext) and (not isheapempty):
{ hasnext      ⇒ { src, wsz ← dir[next].addr, wordcount(dir[next].size)
                 & index, next, state ← 1, dir[next].list, MARK_SCAN_OBJ
                 }
| isheapempty  ⇒ state ← COMPACT_END
| needscompact ⇒ dest, src, obj, state ← 0, 0, mem[0].handle, COMPACT_SCAN
}


procedure MarkScanObj():
    val indexinbounds is index ≤ wsz:
    val hasnext       is next ≠ NULL.handle:
    val waddr         is src + index:
{ indexinbounds ⇒ { mem[waddr].ptr     ⇒ { obj ← mem[waddr].handle
                                           & state ← MARK_ADD_OBJ
                                           & index ← index + 1
                                           }
                  | not mem[waddr].ptr ⇒ index ← index + 1
                  }
| not hasnext   ⇒ state ← MARK_NEXT_OBJ
| hasnext       ⇒ { next, src, index ← dir[next].list, dir[next].addr, 1
                  & wsz ← wordcount(dir[next].size)
                  }
}


procedure MarkAddObj():
    val nlivesize is livesize + wordcount(dir[obj].size) + 1:
    val needsmark is (not NULL) and (not dir[obj].mark):
{ needsmark     ⇒ { not dir[obj].deep ⇒ { dir[obj].mark, livesize ← true, nlivesize
                                         & state ← MARK_SCAN_OBJ
                                         }
                  | dir[obj].deep     ⇒ { dir[obj].list, next ← next, obj
                                         & dir[obj].mark, livesize ← true, nlivesize
                                         & state ← MARK_SCAN_OBJ
                                         }
                  }
| not needsmark ⇒ state ← MARK_SCAN_OBJ
}
```

A.6 Compact

During the compact stage, the IHGC processes memory starting from the lowest address up to the *heappoint*. The header word of every object in memory is loaded to determine its handle. The IHGC then uses this information to retrieve the corresponding directory entry. Marked objects are retained and copied toward the bottom of the memory space one word at a time if necessary. Unmarked objects are reclaimed; their handles are added to the *free* list and their memory locations are zeroed or overwritten with live objects.

```
procedure CompactScan():
    val nsrc          is src + wordcount(dir[obj].size) + 1:
    val wcount        is wordcount(dir[obj].size):
    val needscompact  is src ≠ dest:
    val gcended       is nsrc = heappoint:
    val isoverwritten is (src + wcount) < livesize:
    val needspartzero is src < livesize:
    val needsfullzero is not needspartzero:
{ dir[obj].mark     ⇒ { dir[obj].mark, wsz ← false, wcount
                      & { needscompact ⇒ index, state ← 0, COMPACT_READ
                        | gcended      ⇒ livesize, state ← 0, COMPACT_END
                        | not gcended  ⇒ { src, dest ← nsrc, nsrc
                                          & obj ← mem[nsrc].handle
                                          }
                        }
                      }
| not dir[obj].mark ⇒ { dir[obj].list, free, wsz ← free, obj, wcount
                      & { isoverwritten ⇒ src, obj ← nsrc, mem[nsrc].handle
                        | needspartzero ⇒ index, state ← livesize - src, COMPACT_ZERO
                        | needsfullzero ⇒ index, state ← 0, COMPACT_ZERO
                        }
                      }
}


procedure CompactZero():
    val indexinbounds is index ≤ wsz:
    val nsrc          is src + wsz + 1:
    val gcended       is nsrc = heappoint:
{ indexinbounds ⇒ mem[src + index], index ← 0, index + 1
| gcended       ⇒ heappoint, livesize, state ←  dest, 0, COMPACT_END
| not gcended   ⇒ src, obj, state ← nsrc, mem[nsrc].handle, COMPACT_SCAN
}


procedure CompactRead():
    val indexinbounds is index ≤ wsz:
    val nsrc           is src + wsz + 1:
    val ndest          is dest + wsz + 1:
```

```
    val gcended        is nsrc = heappoint:
    val isoverwritten  is (src + index) < livesize:
    val needspartzero  is not isoverwritten:
{ indexinbounds ⇒ { buffer ← mem[src + index]
                  & { isoverwritten ⇒ state ← COMPACT_WRITE
                    | needspartzero ⇒ state ← COMPACT_CLEAR
                    }
                  }
| gcended       ⇒ { dir[obj].addr, dest ← dest, ndest
                  & heappoint, livesize, state ← ndest, 0, COMPACT_END
                  }
| not gcended   ⇒ { dir[obj].addr, dest ← dest, ndest
                  & src, obj, state ← nsrc, mem[nsrc].handle, COMPACT_SCAN
                  }
}

procedure CompactClear():
    mem[src + index], state ← 0, COMPACT_WRITE

procedure CompactWrite():
    mem[dest + index], index, state ← buffer, index + 1, COMPACT_READ
```

A.7 Termination

A collection cycle terminates when the IHGC transitions to the COMPACT_END state. The implementation details of the end state transition are dependent on the processor's microarchitecture and the instruction set. Therefore, the below specification of the end state is simply a placeholder that performs out-of-memory error checks and raises an exception when a failure is detected.

```
procedure CompactEnd():
    val nofreespace  is heappoint + alloc.wsize + 1 > msz:
    val nofreehandle is free = NULL.handle:
    val cannotalloc  is nofreehandle or nofreespace:
    val iscpupaused  is alloc.waiting and cannotalloc:
{ iscpupaused     ⇒ { oomcount = 1 ⇒ exception()
                    | oomcount = 0 ⇒ state, oomcount ← MARK_INIT, 1
                    }
| not iscpupaused ⇒ state, oomcount ← MARK_INIT, 0
}
```

A.8 Memory Access

Load and store procedures are provided for programs to access memory. The implementation of these procedures is tightly integrated with the processor's pipeline to eliminate coordination

delays between processor and IHGC. Both load and store procedures are augmented with functionality to redirect memory accesses to the correct location while the IHGC is compacting. In addition, load operations process the pointer read for marking when the IHGC is in the marking stage. Store operations set an object's deep flag when a pointer is written into that object.

```
procedure LoadWord(val pointer, alias output):
    val rwindex    is bytetoword(pointer.offset) + 1:
    val srcaddr    is dir[pointer.handle].addr:
    val atbuffer   is (pointer.handle = obj) and (rwindex = index):
    val atdest     is (pointer.handle = obj) and (rwindex <  index):
    val atsrc      is (pointer.handle ≠ obj) or (rwindex ≥ index):
    val srcword    is ( atbuffer and isgcwriting() ⇒ buffer
                      | atdest and isgcmoving()    ⇒ mem[dest + rwindex]
                      | atsrc                      ⇒ mem[srcaddr + rwindex]
                      ):
    val ismarked   is srcword.ptr and dir[srcword.handle].mark:
    val isunmarked is not ismarked:
{ output ← srcword
& { isunmarked and isgcmarking()    ⇒ markObject(srcword.handle)
  | ismarked or (not isgcmarking()) ⇒ skip
  }
}


procedure StoreWord(val pointer, val input):
    val rwindex    is bytetoword(pointer.offset) + 1:
    val srcaddr    is dir[pointer.handle].addr:
    val atbuffer   is (pointer.handle = obj) and (rwindex = index):
    val atdest     is (pointer.handle = obj) and (rwindex <  index):
    val atsrc      is (pointer.handle ≠ obj) and (rwindex ≥ index):
    alias destword is ( atbuffer and isgcwriting() ⇒ buffer
                      | atdest and isgcmoving()    ⇒ mem[dest + rwindex]
                      | atsrc                      ⇒ mem[srcaddr + rwindex]
                      ):
{ destword ← input
& { input.ptr     ⇒ dir[pointer.handle].deep ← true
  | not input.ptr ⇒ skip
  }
}
```

A.9 Memory Allocation

A memory allocation procedure implements the newm instruction. Allocations are simple because the free space is clustered at one end of the memory. So the IHGC only needs to pop a handle from the *free* list, update the directory with the object's information, increment the *heappoint*

and store the object's header word in memory. Objects are allocated marked to ensure that they are retained in the current collection cycle.

```
procedure NewM(val allocsz, alias output)
    val nofreespace    is heappoint + wordcount(allocsz) + 1 > msz:
    val nofreehandle   is free = NULL.handle:
    val cannotallocate is nofreespace or nofreehandle:
    val canallocate    is not cannotallocate:
    val nlivesize      is livesize + wordcount(allocsz) + 1:
{ cannotallocate ⇒ alloc.waiting, alloc.wsize ← true, wordcount(allocsz)
| canallocate    ⇒ { dir[free].addr, dir[free].size ← heappoint, allocsz
                   & dir[free].deep ← false
                   & { isgccollecting()     ⇒ { dir[free].mark ← true
                                                & livesize ← nlivesize
                                                }
                     | not isgccollecting() ⇒ dir[free].mark ← false
                     }
                   & heappoint ← heappoint + wordcount(allocsz) + 1
                   & mem[heappoint].handle ← free
                   & mem[heappoint].ptr ← true
                   & output.ptr, output.handle, output.offset ← true, free, 0
                   & free, alloc.waiting ← dir[free].list, false
                   }
}
```

A.10 Helper Functions and Procedures

The following functions and procedures are provided to simplify the specification of the IHGC's state machine.

```
procedure markObject(val objhan):
    val nlivesize is livesize + wordcount(dir[objhan].size) + 1:
{ dir[objhan].deep     ⇒ { dir[objhan].list, next ← next, objhan
                         & dir[objhan].mark, livesize ← true, nlivesize
                         }
| not dir[objhan].deep ⇒ dir[objhan].mark, livesize ← true, nlivesize
}


function [bpa]bit wordcount(val [bpa]bit s)
    val Bpw is bpw ÷ 8:
{ return ( (s % Bpw) = 0 ⇒ s ÷ Bpw
         | true          ⇒ (s + Bpw) ÷ Bpw
         )
}
```

```
function [bpa]bit bytetoword(val [bpa]bit s)
   return s ÷ (bpw ÷ 8)

function bool isgcwriting():
    return (state = COMPACT_CLEAR) or (state = COMPACT_WRITE)

function bool isgcmoving():
    return (state = COMPACT_READ) or (state = COMPACT_CLEAR) or (state = COMPACT_WRITE)

function bool isgcmarking():
    return (state = MARK_SCAN_REG)  or (state = MARK_CHECK_REG) or
           (state = MARK_ADD_REG)   or (state = MARK_SCAN_OBJ)  or
           (state = MARK_CHECK_OBJ) or (state = MARK_ADD_OBJ)   or
           (state = MARK_NEXT_OBJ)

function bool isgccollecting():
    return (state ≠ MARK_INIT) and (state ≠ COMPACT_END)
```

BIBLIOGRAPHY

[1] *SNU real-time benchmarks*.
http://www.cprover.org/goto-cc/examples/snu.html.
[Online; last accessed 22-April-2020].

[2] ABSINT, *aiT*.
https://www.absint.com/ait/index.htm.
[Online; last accessed 29-October-2020].

[3] ABSINT, *StackAnalyzer*.
https://www.absint.com/stackanalyzer/index.htm.
[Online; last accessed 03-October-2019].

[4] AICAS GMBH, *JamaicaVM*.
https://www.aicas.com/cms/en/JamaicaVM.
[Online; last accessed 22-April-2020].

[5] E. ALBERT, S. GENAIM, AND M. GÓMEZ-ZAMALLOA GIL, *Live heap space analysis for languages with garbage collection*, in Proceedings of the 2009 International Symposium on Memory Management, 2009, pp. 129–138.

[6] R. E. ALY AND M. A. BAYOUMI, *Low-power cache design using 7T SRAM cell*, IEEE Transactions on Circuits and Systems II: Express Briefs, 54 (2007), pp. 318–322.

[7] A. AMAYA GARCÍA, *MicroPython porting code and evaluation scripts for the IHGC*.
https://github.com/andresag01/micropython/tree/sure-port-gc/ports/sure_gc, 2018.
[Online; last accessed 4-August-2021].

[8] A. AMAYA GARCÍA, *Thumb timing simulator*.
https://github.com/andresag01/thumb-sim, 2018.
[Online; last accessed 06-April-2020].

[9] A. AMAYA GARCÍA, *Incorrect loop conditional*.
https://github.com/micropython/micropython/issues/6066, 2020.
[Online; last accessed 28-May-2020].

[10] A. AMAYA GARCÍA AND K. GEORGIOU, *Bristol Worst-Case Analysis Tool*. https://github.com/andresag01/bwca, 2019. [Online; last accessed 4-August-2021].

[11] AMAZON WEB SERVICES INC, *FreeRTOS*. https://www.freertos.org, 2020. [Online; last accessed 14-05-2020].

[12] AMAZON WEB SERVICES INC, *Memory management*. https://www.freertos.org/a00111.html, 2020. [Online; last accessed 14-May-2020].

[13] A. W. APPEL, *Simple generational garbage collection and fast allocation*, Software: Practice and Experience, 19 (1989), pp. 171–183.

[14] A. W. APPEL, J. R. ELLIS, AND K. LI, *Real-time concurrent collection on stock multiprocessors*, in Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, 1988, pp. 11–20.

[15] ARM LIMITED, *ARM Cortex-M series processors*. https://developer.arm.com/ip-products/processors/cortex-m. [Online; last accessed 14-July-2020].

[16] ARM LIMITED, *ARM architecture reference manual*, 1996.

[17] ARM LIMITED, *The ARM-THUMB procedure call standard*. http://infocenter.arm.com/help/topic/com.arm.doc.espc0002/ATPCS.pdf, 2000. [Online; last accessed 28-April-2020].

[18] ARM LIMITED, *ARM architecture reference manual*, 2005.

[19] ARM LIMITED, *Cortex-M0 technical reference manual*, 2009.

[20] ARM LIMITED, *ARMv6-M architecture reference manual*, 2017.

[21] ARM LIMITED, *GNU ARM embedded toolchain v7.3.1*. https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm, 2017. [Online; last accessed 06-April-2020].

[22] ARM LIMITED, *Mbed TLS*. https://github.com/ARMmbed/mbedtls, 2019. [Online; last accessed 23-January-2019].

[23] ASPENCORE, *2017 embedded markets study*. https://m.eet.com/media/1246048/2017-embedded-market-study.pdf, 2017. [Online; last accessed 20-February-2020].

[24] I. S. ASSOCIATION ET AL., *IEEE standard for standard systemc language reference manual*, IEEE Computer Society, (2012).

[25] J. AUERBACH, D. F. BACON, P. CHENG, D. GROVE, B. BIRON, C. GRACIE, B. MCCLOSKEY, A. MICIC, AND R. SCIAMPACONE, *Tax-and-spend: Democratic scheduling for real-time garbage collection*, in Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT), 2008, pp. 245—254.

[26] J. AUERBACH, D. F. BACON, P. CHENG, AND R. RABBAH, *Lime: a Java-compatible and synthesizable language for heterogeneous architectures*, in Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2010, pp. 89–108.

[27] D. F. BACON, P. CHENG, AND V. RAJAN, *Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java*, ACM SIGPLAN Notices, 38 (2003), pp. 81–92.

[28] D. F. BACON, P. CHENG, AND V. RAJAN, *A real-time garbage collector with low overhead and consistent utilization*, ACM SIGPLAN Notices, 38 (2003), pp. 285–298.

[29] D. F. BACON, P. CHENG, AND S. SHUKLA, *And then there were none: A stall-free real-time garbage collector for reconfigurable hardware*, in Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2012, pp. 23–34.

[30] D. F. BACON, S. J. FINK, AND D. GROVE, *Space- and time-efficient implementation of the Java object model*, in European Conference on Object-Oriented Programming, Springer, 2002, pp. 111–132.

[31] H. G. BAKER, *The Treadmill: real-time garbage collection without motion sickness*, ACM SIGPLAN Notices, 27 (1992), pp. 66–70.

[32] H. G. BAKER JR, *List processing in real time on a serial computer*, Communications of the ACM, 21 (1978), pp. 280–294.

[33] M. BENGTSSON, *Real-Time Garbage Collection*, PhD thesis, Lund University, 1990.

[34] M. BERKELAAR, K. EIKLAND, AND P. NOTEBAERT, *lp_solve version 5.5 — open source (mixed-integer) linear programming system*. https://sourceforge.net/projects/lpsolve/, 2005.

[Online; last accessed 16-November-2020].

[35] N. BINKERT, B. BECKMANN, G. BLACK, S. K. REINHARDT, A. SAIDI, A. BASU, J. HESTNESS, D. R. HOWER, T. KRISHNA, S. SARDASHTI, ET AL., *The gem5 simulator*, ACM SIGARCH Computer Architecture News, 39 (2011), pp. 1–7.

[36] S. M. BLACKBURN, R. GARNER, C. HOFFMANN, A. M. KHANG, K. S. MCKINLEY, R. BENTZUR, A. DIWAN, D. FEINBERG, D. FRAMPTON, S. Z. GUYER, ET AL., *The DaCapo benchmarks: Java benchmarking development and analysis*, in Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, 2006, pp. 169–190.

[37] S. M. BLACKBURN AND A. L. HOSKING, *Barriers: Friend or foe?*, in Proceedings of the 4th International Symposium on Memory Management, 2004, pp. 143–151.

[38] S. M. BLACKBURN AND K. S. MCKINLEY, *Ulterior reference counting: Fast garbage collection without a long wait*, in Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, 2003, pp. 344–358.

[39] G. E. BLELLOCH AND P. CHENG, *On bounding time and space for multiprocessor garbage collection*, in ACM SIGPLAN Notices, vol. 34, ACM, 1999, pp. 104–117.

[40] G. BOLLELLA AND J. GOSLING, *The real-time specification for Java*, Computer, 33 (2000), pp. 47–54.

[41] R. BOSCH AND M. TRICK, *Integer programming*, in Search Methodologies, Springer, 2005.

[42] R. A. BROOKS, *Trading data space for reduced time and code space in real-time garbage collection on stock hardware*, in Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, ACM, 1984, pp. 256–262.

[43] 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 ACM SIGARCH Computer Architecture News, vol. 40, IEEE Computer Society, 2012, pp. 225–236.

[44] T. E. CARLSON, W. HEIRMAN, AND L. EECKHOUT, *Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation*, in Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–12.

[45] S. CASS, *The top programming languages 2019*. `https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019`, 2019.

[Online; last accessed 10-February-2019].

[46] Y. CHANG, *Garbage Collection for Flexible Hard Real-Time Systems*, PhD thesis, University of York, 2007.

[47] Y. CHANG AND A. WELLINGS, *Hard real-time hybrid garbage collection with low memory requirements*, in Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06), IEEE, 2006, pp. 77–88.

[48] Y. CHANG AND A. WELLINGS, *Garbage collection for flexible hard real-time systems*, IEEE Transactions on Computers, 59 (2010), pp. 1063–1075.

[49] C. J. CHENEY, *A nonrecursive list compacting algorithm*, Communications of the ACM, 13 (1970), pp. 677–678.

[50] P. CHENG AND G. E. BLELLOCH, *A parallel, real-time garbage collector*, ACM SIGPLAN Notices, 36 (2001), pp. 125–136.

[51] S. CHEREM, L. PRINCEHOUSE, AND R. RUGINA, *Practical memory leak detection using guarded value-flow analysis*, in 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 480–491.

[52] C. CLICK, G. TENE, AND M. WOLF, *The Pauseless GC algorithm*, in Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, ACM, 2005, pp. 46–56.

[53] W. D. CLINGER, A. H. HARTHEIMER, AND E. M. OST, *Implementation strategies for first-class continuations*, Higher-Order and Symbolic Computation, 12 (1999), pp. 7–45.

[54] G. E. COLLINS, *A method for overlapping and erasure of lists*, Communications of the ACM, 3 (1960), pp. 655–657.

[55] D. DETLEFS, *A hard look at hard real-time garbage collection*, in Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE, 2004, pp. 23–32.

[56] L. P. DEUTSCH AND D. G. BOBROW, *An efficient, incremental, automatic garbage collector*, Communications of the ACM, 19 (1976), pp. 522–526.

[57] 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*, in Language Hierarchies and Interfaces: International Summer School, 1976, pp. 43–56.

[58] 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. 966–975.

[59] A. DUNKELS AND LWIP DEVELOPERS, *Lightweight IP*. `https://savannah.nongnu.org/projects/lwip/`, 2019. [Online; last accessed 23-January-2019].

[60] EMBEDDED MICROPROCESSOR BENCHMARK CONSORTIUM, *EEMBC benchmarks*. `https://www.eembc.org/products/`, 2020. [Online; last accessed 01-September-2020].

[61] A. ERMEDAHL, C. SANDBERG, J. GUSTAFSSON, S. BYGDE, AND B. LISPER, *Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis*, in 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[62] H. FALK, S. ALTMEYER, P. HELLINCKX, B. LISPER, W. PUFFITSCH, C. ROCHANGE, M. SCHOEBERL, R. B. SØRENSEN, P. WÄGEMANN, AND S. WEGENER, *TACLeBench: A benchmark collection to support worst-case execution time research*, in Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016), M. Schoeberl, ed., vol. 55 of OpenAccess Series in Informatics (OASIcs), Dagstuhl, Germany, 2016, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 2:1–2:10.

[63] M.-L. FAN, Y.-S. WU, V. P.-H. HU, C.-Y. HSIEH, P. SU, AND C.-T. CHUANG, *Comparison of 4T and 6T FinFET SRAM cells for subthreshold operation considering variability — a model-based approach*, IEEE Transactions on Electron Devices, 58 (2011), pp. 609–616.

[64] K. FARVARDIN AND J. REPPY, *From folklore to fact: comparing implementations of stacks and continuations*, in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 75–90.

[65] S. GATZKA, T. GEITHNER, AND C. HOCHBERGER, *The Kertasarie VM*.

[66] A. GAYNOR, A. VASSALOTTI, A. PITROU, A. GUPTA, B. PETERSON, B. IMPOLLONIA, B. CANNON, C. WINTER, D. LAING, D. MALCOLM, D. JEMEROV, F. PAPA, G. BRANDL, J. ABBATIELLO, J. YASSKIN, M. FIJALKOWSKI, R. KLECKER, S. MONTANARO, S. BEHNEL, T. WOUTERS, V. STINNER, AND Z. WARE, *The Python benchmark suite*. `https://github.com/python/performance`, 2018. [Online; last accessed 31-August-2018].

[67] I. GOG, J. GICEVA, M. SCHWARZKOPF, K. VASWANI, D. VYTINIOTIS, G. RAMALINGAM, M. COSTA, D. G. MURRAY, S. HAND, AND M. ISARD, *Broom: Sweeping out garbage*

collection from big data systems, in Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS {XV}), 2015.

[68] S. C. GOLDSTEIN, *The implementation of a threaded abstract machine*, Tech. Rep. UCB/CSD-94-818, EECS Department, University of California, Berkeley, May 1994.

[69] S. C. GOLDSTEIN, *Lazy threads: compiler and runtime structures for fine-grained parallel programming*, PhD thesis, University of California, Berkeley, 1997.

[70] S. C. GOLDSTEIN, K. E. SCHAUSER, AND D. E. CULLER, *Lazy threads: Implementing a fast parallel call*, Journal of Parallel and Distributed Computing, 37 (1996), pp. 5–20.

[71] D. J. GREAVES AND S. SINGH, *Kiwi: Synthesis of FPGA circuits from parallel programs*, in Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2008, pp. 3–12.

[72] N. GRECH, K. GEORGIOU, J. PALLISTER, S. KERRISON, J. MORSE, AND K. EDER, *Static analysis of energy consumption for LLVM IR programs*, in Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, 2015, pp. 12–21.

[73] D. GROSSMAN, G. MORRISETT, T. JIM, M. HICKS, Y. WANG, AND J. CHENEY, *Region-based memory management in Cyclone*, ACM SIGPLAN Notices, 37 (2002), pp. 282–293.

[74] F. GRUIAN AND Z. SALCIC, *Designing a concurrent hardware garbage collector for small embedded systems*, in Proceedings of the Asia-Pacific Conference on Advances in Computer Systems Architecture, Springer, 2005, pp. 281–294.

[75] J. GUSTAFSSON, A. BETTS, A. ERMEDAHL, AND B. LISPER, *The Mälardalen wcet benchmarks: Past, present and future*, in Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[76] M. R. GUTHAUS, J. E. STINE, S. ATAEI, B. CHEN, B. WU, AND M. SARWAR, *OpenRAM: An open-source memory compiler*, in Proceedings of the 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), IEEE, 2016, pp. 1–6.

[77] M. HAMPTON, *Using contaminated garbage collection and reference counting garbage collection to provide automatic storage reclamation for real-time systems*, Master's thesis, Washington University, 2003.

[78] D. L. HEINE AND M. S. LAM, *A practical flow-sensitive and context-sensitive C and C++ memory leak detector*, in ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, 2003, pp. 168–181.

[79] R. HENRIKSSON, *Scheduling garbage collection in embedded systems*, PhD thesis, Lund University, 1998.

[80] INMOS, *occam 2 reference manual*, Prentice hall, 1988.

[81] ISO/IEC JTC 1/SC 22, *ISO/IEC 9899:2011*, Information technology — Programming languages — C, (2011).
[Online; last accessed 14-May-2020].

[82] ISO/IEC JTC 1/SC 22, *ISO/IEC 14882:2017*, Information technology — Programming languages — C++, (2017).
[Online; last accessed 15-Dec-2020].

[83] S. S. IYER AND E. J. NOWAK, *45 nm SOI and beyond — getting to a general purpose technology*, in Proceedings of the 2007 IEEE International SOI Conference, IEEE, 2007, pp. 1–4.

[84] JERRYSCRIPT, *A JavaScript engine for internet of things*.
`https://jerryscript.net/`, 2019.
[Online; last accessed 10-February-2019].

[85] A. JOANNOU, J. WOODRUFF, R. KOVACSICS, S. W. MOORE, A. BRADBURY, H. XIA, R. N. WATSON, D. CHISNALL, M. ROE, B. DAVIS, ET AL., *Efficient tagged memory*, in Proceedings of the 2017 IEEE International Conference on Computer Design (ICCD), IEEE, 2017, pp. 641–648.

[86] J. A. JOAO, O. MUTLU, AND Y. N. PATT, *Flexible reference-counting-based hardware acceleration for garbage collection*, in Proceedings of the 36th Annual International Symposium on Computer Architecture, 2009, pp. 418–428.

[87] M. S. JOHNSTONE AND P. R. WILSON, *Non-compacting memory allocation and real-time garbage collection*, PhD thesis, University of Texas at Austin, 1997.

[88] R. JONES, A. HOSKING, AND E. MOSS, *The garbage collection handbook: the art of automatic memory management*, Chapman and Hall/CRC, 2016.

[89] T. KALIBERA AND R. JONES, *Handles revisited: optimising performance and memory costs in a real-time collector*, ACM SIGPLAN Notices, 46 (2011), pp. 89–98.

[90] R. KATREEPALLI AND T. HANIOTAKIS, *High speed power efficient carry select adder design*, in Proceedings of the 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), IEEE, 2017, pp. 32–37.

[91] T. KIM, N. CHANG, N. KIM, AND H. SHIN, *Scheduling garbage collector for embedded real-time systems*, ACM SIGPLAN Notices, 34 (1999), pp. 55–64.

[92] T. KIM, N. CHANG, AND H. SHIN, *Joint scheduling of garbage collector and hard real-time tasks for embedded applications*, Journal of Systems and Software, 58 (2001), pp. 247–260.

[93] T. KIM AND H. SHIN, *Scheduling-aware real-time garbage collection using dual aperiodic servers*, in Real-Time and Embedded Computing Systems and Applications, Springer, 2004, pp. 1–17.

[94] G. KISS-VAMOSI, *txt position fixes*.
`https://tinyurl.com/y2zq7jyr`, 2019.
[Online; last accessed 8-January-2021].

[95] G. KISS-VÁMOSI, *LittlevGL*.
`https://github.com/littlevgl/lvgl`, 2019.
[Online; last accessed 23-January-2019].

[96] D. A. KRANZ, R. H. HALSTEAD JR, AND E. MOHR, *Mul-T: A high-performance parallel LISP*, in Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, 1989, pp. 81–90.

[97] A. KWON, U. DHAWAN, J. M. SMITH, T. F. KNIGHT JR, AND A. DEHON, *Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security*, in Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, ACM, 2013, pp. 721–732.

[98] D. LEA, *A memory allocator*.
`http://gee.cs.oswego.edu/dl/html/malloc.html`, 2000.
[Online; last accessed 15-February-2019].

[99] D. LEATHERDALE, S. LAVINGTON, AND I. MACCALLUM, *Atlas*, 1963.
[Online; last accessed 28-April-2020].

[100] Y.-T. LI, S. MALIK, AND A. WOLFE, *Efficient microarchitecture modeling and path analysis for real-time software*, in Proceedings of the 16th IEEE International Real-Time Systems Symposium (RTSS'95), IEEE, 1995, pp. 298–307.

[101] H. LIEBERMAN AND C. HEWITT, *A real-time garbage collector based on the lifetimes of objects*, Communications of the ACM, 26 (1983), pp. 419–429.

[102] LLVM FOUNDATION, *The LLVM compiler infrastructure*.
`https://releases.llvm.org/`, 2018.
[Online; last accessed 06-April-2020].

[103] D. LOCKE, J. J. HUNT, R. ALLEN, J. NIELSEN, M. FULTON, G. BOLLELLA, D. HARDIN, M. SCHOEBERL, T. HENTIES, A. WELLINGS, AND S. ANDERSEN, *Java specification request 302: Safety critical Java technology*.
https://jcp.org/en/jsr/detail?id=302.
[Online; last accessed 20-February-2020].

[104] M. MAAS, K. ASANOVIĆ, AND J. KUBIATOWICZ, *A hardware accelerator for tracing garbage collection*, in Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2018, pp. 138–151.

[105] T. MANN, M. DETERS, R. LEGRAND, AND R. K. CYTRON, *Static determination of allocation rates to support real-time garbage collection*, ACM SIGPLAN Notices, 40 (2005), pp. 193–202.

[106] M. M. MANO AND C. R. KIME, *Logic and Computer Design Fundamentals: Pearson New International Edition*, Pearson Education Limited, 2014.

[107] D. MAY, *The XMOS XS1 Architecture*, XMOS, 2009.

[108] D. MAY, *The SURE architecture*.
http://people.cs.bris.ac.uk/~dave/gcoll.pdf, 2016.
[Online; last accessed 01-April-2020].

[109] J. MCCARTHY, *Recursive functions of symbolic expressions and their computation by machine, Part I*, Communications of the ACM, 3 (1960), pp. 184–195.

[110] B. MCCLOSKEY, D. F. BACON, P. CHENG, AND D. GROVE, *Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors*, Report RC24504, IBM, (2008).

[111] M. MEYER, *A novel processor architecture with exact tag-free pointers*, IEEE Micro, 24 (2004), pp. 46–55.

[112] M. MEYER, *An on-chip garbage collection coprocessor for embedded real-time systems*, in Proceedings of the 11th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05), IEEE, 2005, pp. 517–524.

[113] M. MEYER, *A true hardware read barrier*, in Proceedings of the 5th International Symposium on Memory Management, ACM, 2006, pp. 3–16.

[114] MICROPYTHON.ORG, *Memory manager*.
https://github.com/micropython/micropython/wiki/Memory-Manager, 2018.
[Online; last accessed 07-April-2020].

[115] MICROPYTHON.ORG, *The MicroPython project*.
`https://github.com/micropython/micropython`, 2018.
[Online; last accessed 31-August-2018].

[116] MISRA CONSORTIUM ET AL., *MISRA-C: 2004 — guidelines for the use of the C language in critical systems*, Rapp. tech. ISBN 0, 9524156 (2004), p. 3.

[117] D. A. MOON, *Garbage collection in a large LISP system*, in Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, ACM, 1984, pp. 235–246.

[118] D. A. MOON, *Symbolics architecture*, IEEE Computer, 20 (1987), pp. 43–52.

[119] A. K. MUKHOPADHYAY, *Study and performance analysis of a 32 bit arithmetic logic unit (ALU) designed using two different logic styles in deep submicron (DSM) technology*, in Proceedings of the International Conference on VLSI and Signal Processing (ICVSP), 2014, pp. 10–12.

[120] V. NAGANATHAN, *A comparative analysis of parallel prefix adders in 32nm and 45nm static CMOS technology*, PhD thesis, The University of Texas at Austin, 2015.

[121] S. NAGARAKATTE, J. ZHAO, M. M. MARTIN, AND S. ZDANCEWIC, *SoftBound: Highly compatible and complete spatial memory safety for C*, ACM SIGPLAN Notices, 44 (2009), pp. 245–258.

[122] S. NETTLES AND J. O'TOOLE, *Real-time replication garbage collection*, ACM SIGPLAN Notices, 28 (1993), pp. 217–226.

[123] K. D. NILSEN, *Reliable real-time garbage collection of C++*, Computing Systems, 7 (1994), pp. 467–504.

[124] K. D. NILSEN AND W. J. SCHMIDT, *Cost-effective object space management for hardware-assisted real-time garbage collection*, ACM Letters on Programming Languages and Systems (LOPLAS), 1 (1992), pp. 338–354.

[125] H. NOGUCHI, S. OKUMURA, Y. IGUCHI, H. FUJIWARA, Y. MORITA, K. NII, H. KAWAGUCHI, AND M. YOSHIMOTO, *Which is the best dual-port SRAM in 45nm process technology? — 8T, 10T single end, and 10T differential —*, in Proceedings of the 2008 IEEE International Conference on Integrated Circuit Design and Technology, IEEE, 2008, pp. 55–58.

[126] NORDIC SEMICONDUCTOR, *nRF52832: Versatile Bluetooth 5.2 SoC supporting Bluetooth Low Energy, Bluetooth mesh and NFC*.
`https://www.nordicsemi.com/products/nrf52832`.
[Online; last accessed 4-August-2021].

[127] E. NUTTING, *Feasibility of an integrated hardware garbage collector*. https://dbms.services.bris.ac.uk/media/user/327146/Thesis.pdf, 2017. [Online; last accessed 01-April-2020].

[128] NXP SEMICONDUCTORS, *i.MX RT1020 Crossover MCU with Arm Cortex-M7 core*. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt1020-crossover-mcu-with-arm-cortex-m7-core:i.MX-RT1020. [Online; last accessed 4-August-2021].

[129] ORACLE, *Java platform, standard edition HotSpot virtual machine garbage collection tuning guide*. https://docs.oracle.com/javase%2F8%2Fdocs%2Ftechnotes%2Fguides%2Fvm%2Fgctuning%2F%2F/. [Online; last accessed 20-February-2020].

[130] J. PALLISTER, S. HOLLIS, AND J. BENNETT, *BEEBS: Open benchmarks for energy measurements on embedded platforms*, arXiv preprint arXiv:1308.5174, (2013).

[131] P. PERSSON, *Live memory analysis for garbage collection in embedded systems*, ACM SIGPLAN Notices, 34 (1999), pp. 45–54.

[132] F. PIZLO, D. FRAMPTON, E. PETRANK, AND B. STEENSGAARD, *Stopless: A real-time garbage collector for multiprocessors*, in Proceedings of the 6th International Symposium on Memory Management (ISMM), 2007, pp. 159—172.

[133] F. PIZLO, E. PETRANK, AND B. STEENSGAARD, *A study of concurrent real-time garbage collectors*, in Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2008, pp. 33—-44.

[134] F. PIZLO, L. ZIAREK, AND J. VITEK, *Real time Java on resource-constrained platforms with Fiji VM*, in Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, 2009, pp. 110–119.

[135] PTC, *Real-time Java – PTC Perc*. https://www.ptc.com/en/products/developer-tools/perc. [Online; last accessed 22-April-2020].

[136] I. PUAUT, *Real-time performance of dynamic memory allocation algorithms*, in Proceedings of the 14th Euromicro Conference on Real-Time Systems, IEEE, 2002, pp. 41–49.

[137] W. PUFFITSCH, B. HUBER, AND M. SCHOEBERL, *Worst-case analysis of heap allocations*, in Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Springer, 2010, pp. 464–478.

[138] RED HAT, *Newlib*.
`http://www.sourceware.org/newlib/`, 2018.
[Online; last accessed 5-February-2019].

[139] J. REGEHR, N. COOPRIDER, W. ARCHER, AND E. EIDE, *Efficient type and memory safety for tiny embedded systems*, in Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems, ACM, 2006, p. 6.

[140] T. RITZAU, *Hard real-time reference counting without external fragmentation*, in Proceedings of the Java Optimization Strategies for Embedded Systems Workshop at ETAPS 2001, 2001.

[141] T. RITZAU, *Memory efficient hard real-time garbage collection*, PhD thesis, Linköping University, 2003.

[142] S. G. ROBERTZ AND R. HENRIKSSON, *Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems*, in Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, 2003, pp. 93–102.

[143] P. G. SALGADO, *Design of CPython's garbage collector*.
[Online; last accessed 20-February-2020].

[144] D. SANCHEZ AND C. KOZYRAKIS, *ZSim: Fast and accurate microarchitectural simulation of thousand-core systems*, ACM SIGARCH Computer Architecture News, 41 (2013), pp. 475–486.

[145] W. J. SCHMIDT AND K. D. NILSEN, *Performance of a hardware-assisted real-time garbage collector*, ACM SIGOPS Operating Systems Review, 28 (1994), pp. 76–85.

[146] M. SCHOEBERL, *JOP: A Java optimized processor*, in Proceedings of the OTM Confederated International Conferences "On the Move to Meaningful Internet Systems", Springer, 2003, pp. 346–359.

[147] M. SCHOEBERL, *Real-time garbage collection for Java*, in Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06), IEEE, 2006, pp. 9–pp.

[148] M. SCHOEBERL, *Scheduling of hard real-time garbage collection*, Real-Time Systems, 45 (2010), pp. 176–213.

[149] M. SCHOEBERL, T. B. PREUSSER, AND S. UHRIG, *The embedded Java benchmark suite JemBench*, in Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, 2010, pp. 120–127.

[150] M. SCHOEBERL AND W. PUFFITSCH, *Non-blocking object copy for real-time garbage collection*, in Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems, 2008, pp. 77–84.

[151] M. SCHOEBERL AND J. VITEK, *Garbage collection for safety critical Java*, in Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems, 2007, pp. 85–93.

[152] SEMICONDUCTOR INDUSTRY ASSOCIATION AND OTHERS, *The international technology roadmap for semiconductors*, tech. rep., 2005.
System Drivers chapter.

[153] SEMICONDUCTOR INDUSTRY ASSOCIATION AND OTHERS, *The international technology roadmap for semiconductors*, tech. rep., 2007.
Interconnect chapter.

[154] SEMICONDUCTOR INDUSTRY ASSOCIATION AND OTHERS, *The international technology roadmap for semiconductors*, tech. rep., 2013.

[155] SEMICONDUCTOR INDUSTRY ASSOCIATION AND OTHERS, *The international technology roadmap for semiconductors*, tech. rep., 2013.
System Drivers chapter.

[156] R. SHAHRIYAR, S. M. BLACKBURN, AND D. FRAMPTON, *Down for the count? getting reference counting back in the ring*, in Proceedings of the 2012 International Symposium on Memory Management, 2012, pp. 73–84.

[157] R. SHAHRIYAR, S. M. BLACKBURN, AND K. S. MCKINLEY, *Fast conservative garbage collection*, in Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, 2014, pp. 121–139.

[158] R. SHAHRIYAR, S. M. BLACKBURN, X. YANG, AND K. S. MCKINLEY, *Taking off the gloves with reference counting immix*, ACM SIGPLAN Notices, 48 (2013), pp. 93–110.

[159] F. SIEBERT, *Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector*, ACM SIGPLAN Notices, 34 (1998), pp. 130–137.

[160] F. SIEBERT, *Hard real-time garbage collection in the Jamaica virtual machine*, in Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99), IEEE, 1999, pp. 96–102.

[161] F. SIEBERT, *Real-time garbage collection in multi-threaded systems on a single processor*, in Proceedings of the 20th IEEE International Real-Time Systems Symposium (RTSS'99), IEEE, 1999, pp. 277–278.

[162] F. SIEBERT, *Eliminating external fragmentation in a non-moving garbage collector for Java*, in Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, 2000, pp. 9–17.

[163] F. SIEBERT, *Constant-time root scanning for deterministic garbage collection*, in International Conference on Compiler Construction, Springer, 2001, pp. 304–318.

[164] F. SIEBERT, *Hard Real-Time Garbage Collection in Modern Object Oriented Programming Languages*, PhD thesis, University of Karlsruhe, 2002.

[165] W. SRISA-AN, C.-T. LO, AND J.-M. CHANG, *Active memory processor: A hardware garbage collector for real-time Java embedded devices*, IEEE Transactions on Mobile Computing, 2 (2003), pp. 89–101.

[166] STACK OVERFLOW, *Developer survey results 2019*. https://insights.stackoverflow.com/survey/2019, 2019. [Online; last accessed 10-February-2019].

[167] 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, ACM, 2007, pp. 173–182.

[168] STANDARD PERFORMANCE EVALUATION CORPORATION, *SPEC's benchmarks*. https://www.spec.org/benchmarks.html, 2020. [Online; last accessed 06-April-2020].

[169] G. L. STEELE JR, *Multiprocessing compactifying garbage collection*, Communications of the ACM, 18 (1975), pp. 495–508.

[170] J. E. STINE, I. CASTELLANOS, M. WOOD, J. HENSON, F. LOVE, W. R. DAVIS, P. D. FRANZON, M. BUCHER, S. BASAVARAJAIAH, J. OH, ET AL., *FreePDK: An open-source variation-aware design kit*, in Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education (MSE'07), IEEE, 2007, pp. 173–174.

[171] STMICROELECTRONICS, *STM32 32-bit ARM Cortex MCUs*. https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html. [Online; last accessed 13-January-2021].

[172] STMICROELECTRONICS, *STM32F7 series of very high-performance MCUs with Arm Cortex-M7 core*. https://www.st.com/content/st_com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus/stm32f7-series.html.

[Online; last accessed 4-August-2021].

[173] R. SUGANYA AND D. MEGANATHAN, *High performance VLSI adders*, in Proceedings of the 3rd International Conference on Signal Processing, Communication and Networking (ICSCN), IEEE, 2015, pp. 1–7.

[174] M. SULLIVAN AND R. CHILLAREGE, *Software defects and their impact on system availability — a study of field failures in operating systems*, in Proceedings of the 21st International Symposium on Fault-Tolerant Computing, IEEE, 1991, pp. 2–9.

[175] SYNOPSYS, *Coverity static application security testing*. `https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html`, 2021.
[Online; last accessed 6-January-2021].

[176] TAIWAN SEMICONDUCTOR MANUFACTURING COMPANY LIMITED, *10nm technology*. `https://www.tsmc.com/english/dedicatedFoundry/technology/10nm.htm`.
[Online; last accessed 14-July-2020].

[177] G. THOMAS, *A proactive approach to more secure code*. `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/`, 2019.
[Online; last accessed 03-October-2019].

[178] D. UNGAR, *Generation scavenging: A non-disruptive high performance storage reclamation algorithm*, ACM SIGPLAN Notices, 19 (1984), pp. 157–167.

[179] D. UNGAR, R. BLAU, P. FOLEY, D. SAMPLES, AND D. PATTERSON, *Architecture of SOAR: Smalltalk on a RISC*, in ACM SIGARCH Computer Architecture News, vol. 12, ACM, 1984, pp. 188–197.

[180] L. UNNIKRISHNAN, S. D. STOLLER, AND Y. A. LIU, *Optimized live heap bound analysis*, in Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation, Springer, 2003, pp. 70–85.

[181] R. USSELMANN, *AES (Rijndail) IP Core*. `https://opencores.org/projects/aes_core`.
[Online; last accessed 28-July-2020].

[182] R. USSELMANN, *WISHBONE DMA/Bridge IP Core*. `https://opencores.org/projects/wb_dma`.
[Online; last accessed 28-July-2020].

[183] VIDEO COMPRESSION SYSTEMS PROJECT, *Video Compression Systems*. https://opencores.org/projects/video_systems, 2009. [Online; last accessed 28-July-2020].

[184] N. VIJAYKRISHNAN, N. RANGANATHAN, AND R. GADEKARLA, *Object-oriented architectural support for a Java processor*, in Proceedings of the European Conference on Object-Oriented Programming, Springer, 1998, pp. 330–354.

[185] J. WEIZENBAUM, *Recovery of reentrant list structures in SLIP*, Communications of the ACM, 12 (1969), pp. 370–372.

[186] R. WILHELM, J. ENGBLOM, A. ERMEDAHL, N. HOLSTI, S. THESING, D. WHALLEY, G. BERNAT, C. FERDINAND, R. HECKMANN, T. MITRA, ET AL., *The worst-case execution-time problem — overview of methods and survey of tools*, ACM Transactions on Embedded Computing Systems (TECS), 7 (2008), p. 36.

[187] I. WILLIAMS, *The MUSHROOM machine — an architecture for symbolic processing*, in Proceedings of the IEE Colloquium on VLSI and Architectures for Symbolic Processing, IET, 1989, pp. 4–1.

[188] M. WOLCZKO AND I. WILLIAMS, *The influence of the object-oriented language model on a supporting architecture*, in The Interaction of Compilation Technology and Computer Architecture, Springer, 1994, pp. 223–247.

[189] C. WOLF, *PicoRV32 — A Size-Optimized RISC-V CPU*. https://github.com/cliffordwolf/picorv32. [Online; last accessed 28-July-2020].

[190] C. WOLF, J. GLASER, AND J. KEPLER, *Yosys — a free verilog synthesis suite*, in Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip), 2013.

[191] J. WOODRUFF, R. N. WATSON, D. CHISNALL, S. W. MOORE, J. ANDERSON, B. DAVIS, B. LAURIE, P. G. NEUMANN, R. NORTON, AND M. ROE, *The CHERI capability model: Revisiting RISC in an age of risk*, in Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), IEEE, 2014, pp. 457–468.

[192] XILINX, *Virtex-5 family overview*. https://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, 2015. [Online; last accessed 08-August-2021].

[193] XMOS, *Everyday capable xCore-200*. https://www.xmos.ai/xcore-200/. [Online; last accessed 30-December-2020].

[194] X. YANG, S. M. BLACKBURN, D. FRAMPTON, AND A. L. HOSKING, *Barriers reconsidered, friendlier still!*, ACM SIGPLAN Notices, 47 (2012), pp. 37–48.

[195] T. YUASA, *Real-time garbage collection on general-purpose machines*, Journal of Systems and Software, 11 (1990), pp. 181–198.

[196] ZERYNTH, *Zerynth virtual machine*.
`https://www.zerynth.com/zerynth-virtual-machine`, 2019.
[Online; last accessed 10-February-2019].

[197] B. ZORN, *Barrier methods for garbage collection*, Department of Computer Science, University of Colorado Boulder, 1990.

[198] B. ZORN, *The measured cost of conservative garbage collection*, Software: Practice and Experience, 23 (1993), pp. 733–756.