Hard Real-Time Garbage Collection on Chip Multi-Processors

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Wolfgang Puffitsch

Registration Number 0125944

to the Faculty of Informatics
at the Vienna University of Technology

Advisors: Martin Schöberl
and Herbert Grünbacher

The dissertation has been reviewed by:

(Martin Schöberl)

(Anders P. Ravn)

Wien, January 20, 2012

(Wolfgang Puffitsch)

Erklärung zur Verfassung der Arbeit

Wolfgang Puffitsch
Hegergasse 17/2/19
1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. Jänner 2012

(Wolfgang Puffitsch)

Abstract

Automatic memory management, also known as garbage collection, is a suitable means to increase productivity in the development of computer programs. As it helps to avoid common errors in memory management, such as memory leaks or dangling pointers, it also helps to make programs safer. Conventional garbage collection techniques are however not suited for use in hard real-time systems. As a failure in these systems can have catastrophic consequences—such as the loss of human life—it is of utmost importance to meet the timely requirements of the respective system. In the past few years, methods for garbage collection that are suitable for use in hard real-time systems have been developed. However, these techniques are suited only for uniprocessors. On multi-processors, garbage collection does not yet meet the requirements for hard real-time systems.

Garbage collection reclaims memory objects that are provably inaccessible to the application. Objects that are referenced by global or local variables can be accessed at any time. Starting at these objects, the object graph is traversed by the garbage collector to determine which objects are reachable. As there is the possibility that reachable objects are accessed by the application, they must not be reclaimed. Determining the object references in local variables is challenging in hard real-time systems, because the timely requirements usually do not permit stopping the application to do so. The development of appropriate techniques is one of the core areas of this dissertation.

Memory fragmentation cannot be tolerated in hard real-time systems, because otherwise, it would be impossible to determine reasonable bounds on the maximum memory consumption. In order to defragment memory, it is necessary to relocate objects. To achieve this without interrupting the application, a hardware unit has been developed, which enables transparent object relocation on chip multi-processors.

Multi-processors often use memory models that provide only few guarantees on the ordering of memory accesses. The effects of such memory models on garbage collection are investigated in this thesis. Also, an analysis to determine the maximum size of memory allocations of tasks is presented. Such an analysis is necessary to compute whether the allocation requests of an application can be met even in the worst case.

This thesis points out solutions which enable the reconciliation of garbage collection with the requirements of hard real-time systems on chip multi-processors. The theoretic results are supported by measurement results, which indicate that low release jitter can be achieved in multi-processor systems in the presence of garbage collection.

Kurzfassung

Automatische Speicherbereinigung, auch bekannt als Garbage Collection, ist ein geeignetes Mittel um die Produktivität bei der Entwicklung von Computerprogrammen zu erhöhen. Da sie dabei hilft, häufige Fehler im Bereich der Speicherverwaltung wie Speicherlecks und ungültige Referenzen zu vermeiden, hilft sie auch dabei, Programme sicherer zu machen. Herkömmliche Techniken zur automatischen Speicherbereinigung sind jedoch nicht für den Einsatz in harten Echtzeitsystemen geeignet. Da ein Fehlverhalten in solchen Systemen katastrophale Auswirkungen – wie den Verlust von Menschenleben – haben kann, ist die Einhaltung der zeitlichen Anforderungen des betreffenden Systems von höchster Bedeutung. In den letzten Jahren wurden Methoden zur automatischen Speicherbereinigung entwickelt, die auch für den Einsatz in harten Echtzeitsystemen geeignet sind. Diese Methoden sind jedoch nur für den Einsatz in Einzelprozessoren geeignet. Auf Multiprozessoren erfüllt automatische Speicherbereinigung bisher noch nicht die Anforderungen für harte Echtzeitsysteme.

Automatische Speicherbereinigung gibt Speicherobjekte wieder frei, auf die die Anwendung erwiesenermaßen nicht mehr zugreifen kann. Die Applikation kann jederzeit auf Objekte zugreifen auf die globale oder lokale Variablen verweisen. Beginnend bei diesen Objekten wird der Objektgraph von der automatischen Speicherbereinigung durchlaufen, um festzustellen, welche Objekte erreichbar sind. Da die Möglichkeit besteht, dass die Anwendung auf erreichbare Objekte zugreift, dürfen diese nicht freigegeben werden. Die Bestimmung der Referenzen in lokalen Variablen ist dabei in harten Echtzeitsystemen eine Herausforderung, da die zeitlichen Anforderungen es üblicherweise nicht erlauben, die Anwendung dafür anzuhalten. Die Entwicklung entsprechender Techniken ist einer der Schwerpunkte dieser Dissertation.

In harten Echtzeitsystemen kann eine Fragmentierung des Speichers nicht toleriert werden, weil sich sonst der maximale Speicherverbrauch einer Applikation nicht sinnvoll begrenzen lassen würde. Für die Defragmentierung von Speicher ist es notwendig, Objekte im Speicher zu verschieben. Um dies ohne Unterbrechung der Anwendung zu bewerkstelligen, wurde eine Hardware-Einheit entwickelt, die ein transparentes Verschieben von Objekten auf Multiprozessoren ermöglicht.

In Multiprozessoren kommen oftmals Speicher-Modelle zum Einsatz, die nur wenige Garantien bezüglich der Ordnung von Speicherzugriffen bieten. Die Auswirkungen solcher Speichermodelle auf die automatische Speicherbereinigung werden im Rahmen dieser Dissertation untersucht. Ebenso wird eine Analyse vorgestellt, die es erlaubt, die maximale Größe von Speicheranforderungen eines Tasks zu bestimmen. Eine solche Analyse ist notwendig, um zu berechnen ob die Speicheranforderungen der Applikation unter allen Umständen erfüllt werden können.

Im Rahmen dieser Dissertation werden Lösungen aufgezeigt, die es erlauben, automatische Speicherbereinigung mit den Anforderungen für harte Echtzeitsysteme auf Multiprozessoren miteinander zu vereinbaren. Die theoretischen Ergebnisse werden durch Messungen bestätigt, die demonstrieren, dass auch in Multiprozessor-Systemen mit automatischer Speicherbereinigung die Abweichungen der Task-Startzeiten gering gehalten werden können.

Preface

Garbage, sewage they feed on.
— James Joyce, Ulysses.

This thesis subsumes research that has been carried out in the past few years. Chapter 1 provides an introduction to the topics covered in the body of the thesis. Chapters 2, 3, 4, 5, and 6 present results that have been published as peer-reviewed papers at international workshops and conferences. These papers have been included in this thesis with only reformatting and careful copy-editing applied, without changing their contents. Therefore, the respective chapters each contain an abstract, an introduction and a related work section. As there is an inevitable overlap, some parts may appear to be repetitive. On the other hand, each of these chapters is self-contained and may be read without necessarily having to read the preceeding chapters. The results of the thesis are summarized in Chapter 7, which also provides an outlook on future work. Chapters 2, 3, 4, 5, and 6 are based on the following publications:

Chapter 2 Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, JTRES '08, pages 68–76, Santa Clara, California, USA, September 2008. ACM Press.

Chapter 3 Wolfgang Puffitsch. Decoupled root scanning in multi-processor systems. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 91–98, Atlanta, Georgia, USA, October 2008. ACM Press.

Chapter 4 Wolfgang Puffitsch. Data caching, garbage collection, and the Java memory model. In *Proceedings of the 7th international workshop on Java technologies for real-time and embedded systems*, JTRES '09, pages 90–99, Madrid, Spain, September 2009. ACM Press.

Chapter 5 Wolfgang Puffitsch, Benedikt Huber and Martin Schoeberl. Worst-case analysis of heap allocations. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation (ISoLA '10)*, LNCS 6416, pages 464–478, Heraclion, Greece, October 2010. Springer-Verlag.

Chapter 6 Wolfgang Puffitsch. Hard real-time garbage collection for a Java chip multi-processor. In *Proceedings of the 9th international workshop on Java technologies for real-time and embedded systems*, JTRES '11, pages 64-73, York, United Kindom, September 2011. ACM Press.

Acknowledgements

First of all, I am deeply grateful to my family for all their support during my whole lifetime.

I would like to thank my advisor Martin Schöberl for his motivation, his valuable support and the inspiring discussions. Without him, I probably would not have found my way into research and written this thesis.

Thanks to my colleagues and friends Benedikt Huber and Roland Kammerer for interesting discussions and occasionally distracting me from working on my thesis.

I would also like to thank Christof Pitter and Michael Zolda for their constructive comments and helpful suggestions on this thesis.

This thesis was written during my employment as research assistant for professor Herbert Grünbacher at the Institute of Computer Engineering at the Vienna University of Technology. I also received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

Contents

Preface ix

1 Introduction 1

1.1 Hard Real-Time Systems 1
1.1.1 Terminology 2
1.2 Chip Multi-Processors 2
1.3 Memory Management Techniques 3
1.3.1 Static Allocation 3
1.3.2 Explicit Memory Allocation 4
1.3.3 Stack Allocation 4
1.3.4 Scoped Memory (Real-Time Specification for Java) 5
1.3.5 Scoped Memory (Safety Critical Java) 5
1.3.6 Garbage Collection 6
1.4 Basics of Garbage Collection 6
1.4.1 Terminology 6
1.4.2 Reference Counting 8
1.4.3 Tracing Garbage Collection 8
1.5 Real-Time Garbage Collection 11
1.5.1 Scheduling 12
1.5.2 Root Scanning 13
1.5.3 Compaction 14
1.5.4 Real-Time Garbage Collection on a Chip Multi-Processor 15
1.5.5 Time and Space Bounds 16
1.6 Contributions 17

2 Non-blocking Root Scanning 23

2.1 Introduction 23
2.2 Related Work 24
2.3 Preemptible Root Scanning 25
2.3.1 Correctness 26
2.3.2 Execution Time Bounds 33
2.4 Evaluation 37
2.4.1 Results 38
2.4.2 Discussion 39
2.5 Conclusion and Outlook 40
2.6 Acknowledgement 40

3 Decoupled Root Scanning in Multi-Processor Systems 43
3.1 Introduction . . . 43
3.1.1 Garbage Collection Algorithms . . . 44
3.1.2 Real-Time Garbage Collection . . . 45
3.2 Root Cache . . . 46
3.2.1 Scheduling of Local Root Scans . . . 48
3.2.2 Garbage Collection Period . . . 48
3.2.3 Design Considerations . . . 50
3.3 Evaluation . . . 51
3.3.1 Resource Consumption . . . 52
3.3.2 Execution Time . . . 53
3.3.3 Experimental Results . . . 53
3.4 Related Work . . . 56
3.5 Conclusion and Outlook . . . 56

4 Data Caching, Garbage Collection, and the Java Memory Model 59
4.1 Introduction . . . 59
4.2 Related Work . . . 60
4.3 The Java Memory Model . . . 61
4.3.1 Terminology . . . 61
4.3.2 Happens-before . . . 62
4.3.3 Causality . . . 63
4.4 Data Cache Design . . . 64
4.4.1 Coherence and Consistency . . . 64
4.4.2 Lazy Release Consistency . . . 65
4.4.3 Consistency Implementation . . . 65
4.4.4 Analysis Friendliness . . . 66
4.5 Garbage Collection . . . 67
4.5.1 GC Algorithm . . . 68
4.5.2 Tracing the Object Graph . . . 70
4.5.3 Sliding Consensus . . . 71
4.5.4 Moving Objects . . . 74
4.5.5 Object Creation . . . 75
4.5.6 Final Variables . . . 75
4.5.7 Internal Data Structures . . . 76
4.6 Conclusion and Outlook . . . 76
4.7 Acknowledgement . . . 77

5 Worst-Case Analysis of Heap Allocations 81
5.1 Introduction . . . 81
5.2 Related Work . . . 82
5.3 WCET Analysis . . . 83
5.3.1 Loop and Recursion Bounds . . . 83
5.3.2 Data-flow Analysis . . . 84

5.3.3 Execution Time Calculation . . . 85
5.4 Heap Allocation Analysis . . . 85
5.4.1 Array Size Bounds . . . 85
5.4.2 Object Layouts . . . 86
5.4.3 Cost Functions . . . 87
5.5 Evaluation . . . 89
5.5.1 Results . . . 90
5.5.2 JVM Comparison . . . 92
5.5.3 Programming Style . . . 92
5.6 Conclusion . . . 93

6 Hard Real-Time Garbage Collection for a Java Chip Multi-Processor 97
6.1 Introduction . . . 97
6.1.1 Garbage Collection Basics . . . 98
6.2 Related Work . . . 99
6.2.1 Scheduling . . . 99
6.2.2 Stack Scanning . . . 99
6.2.3 Fragmentation . . . 100
6.2.4 Copying Hardware . . . 101
6.3 Run-Time System . . . 101
6.3.1 Memory Access . . . 101
6.3.2 Caching . . . 102
6.3.3 Synchronization . . . 102
6.3.4 Scheduling . . . 103
6.4 Hard Real-Time GC . . . 103
6.4.1 General Algorithm . . . 104
6.4.2 Stack Scanning . . . 106
6.4.3 Copying . . . 107
6.5 Evaluation . . . 109
6.5.1 Hardware . . . 109
6.5.2 System Evaluation . . . 110
6.6 Conclusion . . . 115

7 Conclusions and Outlook 119
7.1 Generalization . . . 119
7.1.1 Stack Scanning . . . 119
7.1.2 Copying . . . 120
7.1.3 Locking . . . 120
7.1.4 Handles . . . 121
7.2 Conclusions . . . 122
7.3 Future Work . . . 122
7.3.1 Exact Root Scanning . . . 123
7.3.2 Analysis . . . 123
7.3.3 Verification . . . 124

Curriculum Vitae **127**

1 Introduction

Everyday life has become dependent on computer systems in many regards. While some of those systems make our lives more convenient, other systems are critical to our safety. Modern airplanes, trains, or cars are controlled by computers, and failure to operate correctly would put human life at risk. At the same time, these systems become ever more complex, be it to provide a higher level of convenience, to increase safety, or to reduce costs.

Languages such as Java increase programmer productivity compared to low-level languages like C by eliminating many potential sources of errors. One area where Java eliminates pitfalls of low-level languages is memory management, which is fully automatic in Java. Automatic memory management, also known as garbage collection, is often considered unsuitable for safety-critical systems, especially if stringent timing requirements have to be met. However, research in the past few years has shown that—at least on uniprocessors—it is possible to provide timely guarantees even in the presence of automatic memory management.

The growing complexity of computer systems also results in higher demands on the computational power of the underlying platform. As the performance of a single processor core cannot keep up with the demand, modern systems often comprise multiple processor cores. While this helps performance, it is more challenging to provide timely guarantees on multi-processor systems than it is on uniprocessor systems.

The goal of this thesis is to develop techniques that enable garbage collection on chip multi-processors (CMPs) for safety-critical systems with stringent timing requirements. The following sections provide definitions of concepts used throughout the thesis and discuss the challenges that have to be faced to achieve the goal of the thesis. Furthermore, alternatives to garbage collection are discussed in order to provide evidence that the development of future safety-critical systems would benefit from automatic memory management.

1.1 Hard Real-Time Systems

> In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced. [32]

One important class of systems that falls under the above definition are *hard real-time* systems, where a result that is delivered too late is not only useless, but may also lead to catastrophic consequences, such as the loss of human life. Therefore, hard real-time systems must be designed rigorously, and it must be *provable* that the timing requirements are met.

A real-time system consists of one or more executable entities, which are commonly called *tasks*. The execution of a task may be triggered by the progress of time or by other events and must finish before the task's *deadline*. In hard real-time systems, it must be proven that all tasks meet their deadline even under worst-case conditions. In order to reason about this, a safe upper bound for the *worst-case execution times* (WCETs) of all tasks must be known. Furthermore, the scheduling of tasks must be analyzed to prove that with a given scheduling algorithm all tasks are able to meet their deadlines.

Examples for hard real-time systems can be found in airplanes, trains, cars, or automated assembly lines. Technological progress enables more and more functionality to be integrated in these real-time systems. As this also increases the complexity of these systems, it is necessary to provide developers with means to handle this complexity without compromising safety.

1.1.1 Terminology

Real-time tasks can be classified as *periodic*, *aperiodic* or *sporadic*. While periodic tasks are released at a fixed rate, aperiodic tasks may be triggered at any time. Sporadic tasks are similar to aperiodic tasks, but consecutive releases must at least be separated by a minimum inter-arrival time. An example for a periodic task would be a task that polls a sensor at regular intervals; an interrupt service routine that is triggered by an external source could be modeled as aperiodic task.

A *job* is a specific instance of a task, for example the fifth execution of a periodic task. The *release time* of a job is the instant at which it becomes eligible for execution. The time between a job's release and and its termination is the *response time*. In contrast to the WCET, the worst case response time also takes into account delays that are caused by preemptions. In order for a system to be schedulable, the response times of all jobs must be smaller than their deadlines.

In real systems, jobs cannot always start execution immediately when their release time arrives. On the one hand, the overhead for context switches has to be taken into account, which may vary with the processor state and the tasks that are involved in the context switch. On the other hand, a job with higher priority may be currently executing. The time between the (theoretic) release time and the instant at which a job actually starts its execution is called *release jitter*. In order to achieve predictable system behavior, interference from lower-priority tasks should be avoided for highest-priority tasks, and the release jitter should be as small as possible.

1.2 Chip Multi-Processors

The growth of uniprocessor performance has slowed down significantly in the past few years, compared to the growth between the middle of the 1980s and the early 2000s [15]. Although the number of available transistors on a single chip still increases, the returns for adding more transistors to a processor are diminishing. Also, power consumption increases with processor complexity and clock frequency. Potential increases in uniprocessor performance are therefore limited.

Multi-processors are able to overcome these limitations. Compared to an overly complex uniprocessor, a multi-processor that consists of several slightly simpler cores uses the available transistors more efficiently, while keeping the power consumption at acceptable levels. In the past few years, multi-processors have entered the mainstream market and are included in personal computers as well as in recent mobile phones.

In general, the individual cores of a multi-processor can be implemented on separate silicon chips or on a single chip. Multi-processors of the latter type are called *chip multi-processors* (CMPs). In this thesis, it is usually assumed that the underlying CMP comprises "a few" processors (up to 8 in the evaluations). There is no fixed limit on the number of cores, but processors that consist of several hundred cores might require different solutions.

Multi-processors furthermore can be classified whether the individual processor cores access a central shared memory, or whether memory is distributed. Following the programming model of Java, the thesis assumes that memory is shared between the cores. However, this does not necessarily mean that the memory latencies for all cores are equal. For example, memory arbitration can provide non-uniform latencies to reduce the WCETs of tasks that execute on a specific core.

In embedded systems, multi-processors are often heterogeneous and include specialized processors, for example for signal processing. In contrast, general purpose systems are usually homogeneous and consist of several identical processor cores. The evaluations in this thesis use a CMP that is *mostly* homogeneous. While the instruction sets of processors are identical, the processors may include specialized I/O components; this is especially true for the processor that performs garbage collection.

1.3 Memory Management Techniques

Computer programs require memory to fulfill their purpose. The allocation of memory has to be organized in order to avoid conflicts over the usage of a specific memory area and in order to detect whether a system provides enough memory. Memory management may be performed statically when a program is linked, or dynamically at run-time. Allocation may happen implicitly or explicitly; the same is true for the release of memory that is not required any more. In the following, several techniques for memory management are discussed, highlighting their advantages and drawbacks.

1.3.1 Static Allocation

With static memory allocation, the locations of memory objects are determined when linking a program. In theory it would be possible to assign more than one memory object to a location. However, it is usually assumed that memory objects are alive during the whole lifetime of a program and therefore cannot share a location. As a consequence, the run-time system must provide enough memory for all statically allocated memory objects that are used in the program, even if they are used very briefly, e.g., only during initialization.

Statically allocated memory is well suited for small systems with rather static behavior. As memory addresses are fixed and known, static memory allocation may help in achieving predictable performance and computing tight WCET bounds. However, for more dynamic systems, static memory allocation has a few drawbacks. First, as pointed out above, all memory objects remain allocated during the whole lifetime of a program, which leads to a considerable overhead. Second, data structures such a lists or trees are cumbersome to implement with static memory allocation. Instead of leaving the memory management to the well-tested run-time system, the programmer has to manage the statically allocated memory manually when implementing such data structures.

Apart from the above theoretical considerations, languages such a Java do not support static memory allocation. While Java does provide means to allocate data that appears to be statically allocated, the data is actually allocated during the execution of the program [22]. The run-time system therefore must support memory allocation at run-time, even if the programmer intends to use only statically allocated memory.

1.3.2 Explicit Memory Allocation

Feed the hungry, save the whales, free the mallocs.
— Anonymous

With explicit memory allocation, memory objects are allocated at run-time, but the reserved memory must be returned to the underlying system explicitly if it should be made available for future use. In the C programming language, memory allocation and deallocation is done with the functions `malloc()` and `free()` [1]. These functions overcome the limitations of static memory allocation. Memory objects only need to remain allocated while they are used, and dynamic data structures become considerably simpler to implement. However, these functions come along with serious pitfalls.

Many programs, and especially real-time programs, are designed to execute continuously, repeatedly allocating memory. If the programmer fails to deallocate the allocated memory objects after use, the program *leaks* memory. A continuously executing program that leaks memory will eventually run out of memory and crash. Conversely, deallocating memory that still might be used in the future leads to a different problem: memory references may point to memory that already has been allocated for a different memory object. A memory reference that points to a memory object that has been deallocated is called a *dangling pointer*.

Even in relatively simple programs, these types of bugs are difficult to find. Memory leaks often only cause problems after executing a program for a much longer period of time than can be afforded during development and testing. Dangling pointers can remain undetected for a considerable amount of time until they lead to a malfunction, and in many cases they manifest in a part of the program that is unrelated to the underlying cause. In complex programs, debugging is further complicated by the fact that allocation and deallocation may be separated by a considerable amount of code, or even take place in different modules of the program. The potential hazards of explicit memory allocation may be acceptable in general purpose computing. However, it should be used cautiously in the context of safety-critical systems [17].[1]

1.3.3 Stack Allocation

In many cases, memory objects only live during the lifetime of a function. In such cases, the required memory can be allocated in the current stack frame. Stack allocation only requires the adjustment of the stack pointer and is therefore highly efficient. Also, memory is deallocated

[1]I agree with Holzmann [17] with regard to `malloc()/free()`, but obviously do not share the skepticism towards garbage collection expressed in that paper.

automatically when the respective function returns. Unfortunately, it cannot be used for all kinds of allocations: stack allocated memory may be shared with callees of the current function, but may not be passed up to the calling function or be shared with other threads. Failing to adhere to these restrictions would lead to dangling pointers, which could result in data corruption and failure of the application.

The Java programming language does not provide means for stack allocation. However, in order to reduce the cost of memory allocations, stack allocation can be inferred automatically if it can be proven that references to an object do not escape from the current method [13, 9]. These techniques are orthogonal to the techniques presented in this thesis, and future Java Virtual Machines (JVMs) could implement both approaches.

1.3.4 Scoped Memory (Real-Time Specification for Java)

At the time when the Real-Time Specification for Java (RTSJ) [6] was drafted, real-time garbage collection was not considered mature enough to be the only memory management mechanism. In order to provide real-time programmers with a memory allocation scheme that is consistent with Java but makes it easier to provide guarantees for real-time programs, *scoped memory* was introduced. Although being conceptually similar to stack allocation, scoped memory allows threads to share data.

Threads may enter the scoped memory and allocate objects in that area. When all threads have exited the scoped memory, its contents are reclaimed. A job can therefore enter a scoped memory at the begin of its execution, allocate memory in it and exit the scoped memory at the end of its execution. When the next job of the same task enters the scoped memory, the whole memory area is available for allocations again (unless some other thread has kept its contents alive, of course). As such allocation behavior is not uncommon for real-time systems, scoped memory is suitable for a range of real-time applications.

However, although threads can share data, there are some patterns that are difficult to implement with scoped memory. One example is the producer/consumer pattern, where one thread creates (produces) objects and another task releases (consumes) these objects again. With scoped memory, it is necessary that at least one thread remains in the scoped memory if there is data for the consumer available. On the other hand, all threads must exit the scoped memory at least from time to time to clear the scoped memory and make room for new allocations.

In order to avoid problems with dangling pointers, certain rules must be followed for reference assignments when using scoped memory. A reference in a scoped memory must not point to an object in an inner (nested) scoped memory, and references on the heap must not point to objects in scoped memory. Failure to obey these rules leads to run-time errors. The possibility of triggering run-time errors, together with the problems when implementing certain programming patterns, make scoped memory probably less useful than intended by the authors of the RTSJ.

1.3.5 Scoped Memory (Safety Critical Java)

The specification for Safety Critical Java (SCJ) [23] is based on a subset the RTSJ, and aims at making Java usable for safety-critical applications. SCJ applications consist of sequences of *missions*, which in turn consist of sets of tasks. Apart from an *immortal* memory area, where memory is never reclaimed, SCJ defines two types of scoped memory: mission memory and private memory. Mission memories may be shared, and are cleared only when the respective mission finishes. The individual tasks execute in private memories, which (as the name implies) cannot be shared, but may be nested.

SCJ also provides annotations that can be used to statically check the validity of reference assignments. A successfully validated program is known not to trigger run-time exceptions related to violations of reference assignment rules. Validating this part of a program at compile time probably increases the usability of the memory model compared to the RTSJ. However, the issue of being unsuitable for certain programming patterns is still present, and the restricted memory model of SCJ potentially fits even fewer usage patterns than the RTSJ model.

1.3.6 Garbage Collection

With garbage collection, memory objects that are known not to be used in the future are reclaimed automatically by the run-time system. Garbage collection solves many of the issues mentioned for other memory allocation techniques. In contrast to static allocation, memory objects only remain allocated while they are used, and dynamic data structures are easy to implement. There is no need to scope the lifetime of objects, as it is necessary for stack allocation or scoped memory. As a garbage collector does not reclaim objects that still might be referenced by the application, dangling pointers are impossible. While it is not impossible to leak memory, it is considerably less likely than with explicit memory allocation.

Of course, garbage collection does not come for free, and introduces memory and performance overheads. However, the advantages provided by garbage collection seem to outweigh these overheads in many cases, as indicated by the success of garbage collected languages such as Java.

Garbage collection has a reputation of being unpredictable and incompatible with real-time systems. However, research has shown that it is possible to build garbage collectors for hard real-time systems. The goal of this thesis is to provide further evidence for this fact and to demonstrate that garbage collection is feasible even for hard real-time systems on CMPs.

1.4 Basics of Garbage Collection

This section provides a brief introduction to garbage collection. Readers who are interested in a more thorough introduction to garbage collection may refer to the seminal book by Jones [20] and the recent sequel to that book by the same author [19].

1.4.1 Terminology

Objects are said to be *live* if they are used in the future. The fundamental idea behind garbage collection is that objects that are not live anymore can be reclaimed. As a garbage collector cannot predict the future, liveness of objects has to be approximated. Only objects where it is impossible that they are used in the future may be reclaimed. This is the case for objects that are not referenced anywhere anymore and which hence are not *reachable*.

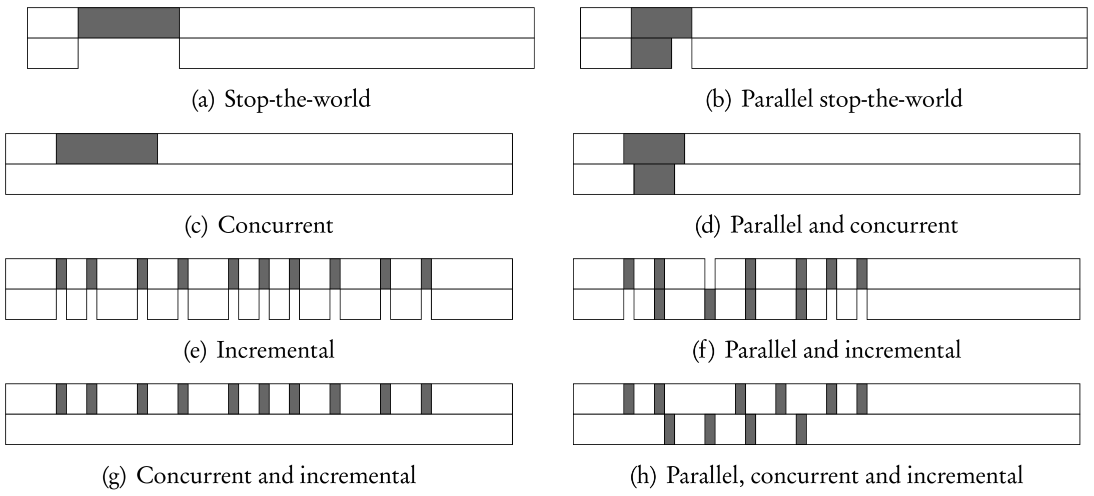

(a) Stop-the-world
(b) Parallel stop-the-world
(c) Concurrent
(d) Parallel and concurrent
(e) Incremental
(f) Parallel and incremental
(g) Concurrent and incremental
(h) Parallel, concurrent and incremental

Figure 1.1: Stop-the-world, concurrent, incremental, and parallel garbage collection

The simplest implementation of a garbage collector is a *stop-the-world* garbage collector, which stops the application, cleans up the memory, and then lets the application resume. Such behavior is not only unacceptable for real-time systems, but also annoying if the application interacts with human beings. While garbage collection is being performed, the application cannot react to input and appears to be frozen. One solution to this issue are *incremental* garbage collectors, which split the garbage collection work into smaller pieces, in between which the application can continue execution. The resulting pauses are more frequent but considerably shorter, resulting in a smaller perceived impact of the garbage collector on the application behavior.

A *concurrent* garbage collector executes in parallel to the application. In a multi-processor system, a perfectly concurrent garbage collector that can keep up with the application's allocation demands would not cause any pauses to the application. A concurrent garbage collector can utilize the parallelism of a system, but does not necessarily contain more than one thread of execution. If a garbage collector *does* contain more than one thread of execution, it is said to be *parallel*. In the context of garbage collection, application threads are sometimes referred to as *mutators*, because they change (mutate) the object graph.

Figure 1.1 visualizes the execution of garbage collection work for the various combinations of these concepts for a multi-processor with two processor cores. In that figure, white bars represent the execution of application tasks, while shaded areas show the execution of garbage collection work. As the concepts are orthogonal, any combination is possible. For example, Figure 1.1(b) shows a parallel stop-the-world garbage collector. In the context of real-time systems, the combinations shown in Figures 1.1(g) and 1.1(h) are most interesting, because they minimize disruptions for real-time tasks.

1.4.2 Reference Counting

One approach to garbage collection is reference counting, which keeps track of the number of references that point to a certain object. When assigning a reference to a variable, the count for the referenced object is incremented, while the count for the object pointed to by the overwritten reference is decremented. If that count drops to zero, the object has become unreachable and it can be reclaimed. Reference counting is naturally incremental, as garbage collection takes place whenever objects become unreachable instead of in one big pass.

However, reference counting faces problems with cyclic data structures, which would lead to memory leaks. Objects that are part of a cycle in the object graph always have a reference count greater than zero, even if they are unreachable from outside the cycle. A trivial solution to this problem is to forbid cycles in the object graph, which is however not an appropriate solution for a language such as Java, because it would preclude common data structures like double-linked lists. An alternative solution is to back up reference counting with a garbage collection algorithm that is able to handle cyclic data structures. Using a different garbage collection algorithm to collect cycles is not a reasonable solution for hard real-time systems either. If the cycle collection mechanism breaks real-time guarantees, so does the garbage collector as a whole, and any benefits from reference counting are voided. If the garbage collection algorithm that is able to collect cycles provides better guarantees, there is little point in using reference counting anyway.

Recent approaches [5, 25] tightly integrate reference counting with techniques from tracing garbage collection, which is discussed in the following section. The maximum pause times reported in [25] appear to be related to the mechanisms borrowed from tracing garbage collection. Research on hard real-time garbage collection—which among other things aims at achieving low pause times—should therefore focus on these mechanisms rather than on reference counting.

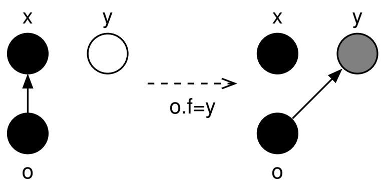
1.4.3 Tracing Garbage Collection

Tracing garbage collection determines the set of reachable objects by traversing the object graph. Objects referenced by local and global variables are always reachable by the application; the respective variables are called the *roots* of the object graph. Starting from the objects referenced by the roots, the garbage collector recursively follows the references in object fields. Once it has visited all objects that are reachable from the root set, it can reclaim the unvisited objects.

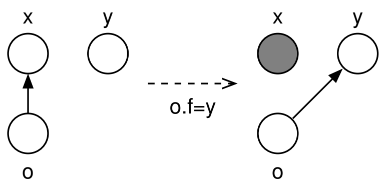
The precise algorithm for tracing the object graph differs between garbage collection algorithms. The *tri-color abstraction* [10] has proven useful to reason about the correctness of tracing garbage collectors without having to consider the actual implementation of the graph traversal algorithm.

1.4.3.1 Tri-Color Abstraction

Objects that have not been visited are said to be *white*. Objects that have been visited, but not yet scanned for references to other objects are *gray*. If an object has been visited and scanned for references, it is called *black*. Before collecting the roots and traversing the object graph,



(a) Incremental-update barrier



(b) Snapshot-at-beginning barrier

Figure 1.2: Write barriers

all objects are white. After tracing, all reachable objects are black; all unreachable objects are white and can be reclaimed.

During tracing, one of the following invariants must hold:

- No black object points to a white object (strong tri-color invariant), or
- Every white object referenced by a black object must also be reachable from a gray object through a chain of white objects (weak tri-color invariant).

These invariants can be used to formulate incremental tracing algorithms, where the application may alter the object graph while it is being traced. The key concept behind such algorithms are *write barriers*, which intercept writes to reference fields and enforce the invariants.

1.4.3.2 Write Barriers

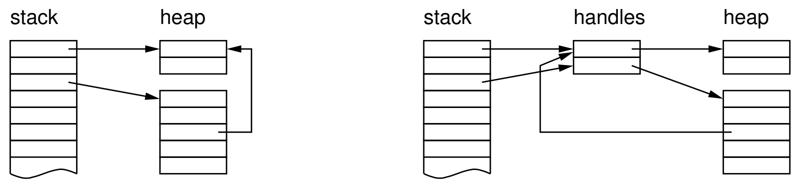
A tracing algorithm can enforce the strong tri-color invariant, by using an *incremental update* write barrier [10] (also called Dijkstraa write barrier). If a reference to a white object is stored in a black object, the referenced object is made gray. Therefore, a black object can never point to a white object. Figure 1.2(a) illustrates the effect of the incremental update write barrier when storing a reference to the white object `y` into the object field `o.f`.

A *snapshot-at-beginning* or Yuasa write barrier [33] enforces the weak tri-color invariant. Whenever a reference to a white object is overwritten, the formerly referenced object is shaded gray. Therefore, if a chain of white objects is potentially broken, a new chain that starts at a gray object is created. Objects that were reachable through the original chain remain visible to the garbage collector. In Figure 1.2(b), object `x` remains visible to the garbage collector, even if field `o.f` points to object `y` after the write.

By using write barriers, the critical sections for tracing the object graph can be kept reasonably short. Only examining the value of an object field, pushing it onto the tracing algorithm's work list, and storing the new value must be atomic with regard to modifications of the object graph. Also, writes to reference fields are relatively rare, which keeps the performance impact of the write barrier code rather small [36].

1.4.3.3 Compaction

Memory areas where objects are repeatedly allocated and reclaimed are subject to fragmentation. While there may be enough free memory in total to satisfy an allocation request, there might



(a) Direct references (b) References point to handles

Figure 1.3: Object layouts

be no contiguous region of free memory that is large enough to accommodate the request. In systems where fragmentation is unacceptable, a common solution is to relocate objects to a contiguous memory region. This *compaction* eliminates "holes" between the objects and makes the respective memory available for allocations.

Support for compaction influences the layout of objects in memory. Figure 1.3(a) shows an object layout where references directly point to the object fields. This layout is beneficial in terms of performance and memory overhead. However, it complicates object relocation considerably, because all references to that object must be updated after relocation. On the one hand, these updates can entail considerable costs. On the other hand, the garbage collector must be able to distiguish reference fields and variables from primitive-type fields and variables, as it would otherwise erroneously change primitive-type values.

An alternative to references that directly point to objects is the use of *handles*, as shown in Figure 1.3(b). With such an object layout, references point to handles, which then point to the objects. While this introduces a level of indirection, which affects performance and introduces memory overheads, it greatly simplifies object relocation. Instead of having to update all references to an object, only the handle has to be updated. In systems where heap compaction is frequent, a handle-based object layout can therefore be beneficial.

In concurrent garbage collectors, heap accesses follow a *forwarding pointer*, which points to the current object location, during compaction. This is necessary for the case where the garbage collector has already relocated an object but not yet updated all references to it. When considering the worst-case behavior of heap accesses, following the indirection through the forwarding pointer has similar costs as following the handle indirection. Therefore, an object layout where references directly point to objects cannot be expected to provide significant benefits with regard to the WCETs of application tasks.

1.4.3.4 Mark-Sweep and Copying Collectors

Tracing garbage collectors are usually organized in a number of phases. After initialization, the root set is computed in the *root scanning* phase. Afterwards, the object graph is traced in the *mark* phase. Objects that have not been visited during marking are reclaimed in the *sweep* phase. Garbage collectors that implement these phases are referred to as *mark-sweep* garbage collectors. *Mark-compact* garbage collectors implement heap compaction as a separate phase.

Copying garbage collectors integrate heap compaction with the tracing phase. In these collectors, the heap is divided into two *semi-spaces*, a *from-space* and a *to-space*. When tracing the object graph, objects are copied from the from-space to the to-space. In terms of the tri-color abstraction, objects in the from-space may be white or gray, while objects in the to-space are black. The garbage collector developed in the scope of this thesis is a copying collector. However, most of the techniques could also be applied to mark-compact garbage collectors.

1.4.3.5 Scheduling Garbage Collection

When implementing garbage collection, it has to be decided when garbage collection is triggered. If the garbage collector should execute as rarely as possible, it is reasonable to trigger garbage collection only when the system is about to run out of memory. If garbage collection pauses should be avoided, it may be more appropriate to perform garbage collection continuously or as a periodic task.

A second fundamental choice is which thread of execution performs garbage collection work. In a *work-based* approach, threads that allocate memory perform garbage collection work. In contrast, a *time-based* approach uses dedicated threads of execution for the garbage collector.

For a stop-the-world garbage collector, it would be reasonable to wait until the application runs out of memory and let the thread whose allocation request could not be met perform garbage collection. However, if timing requirements have to be taken into account, waiting until the application runs out of memory would result in unacceptable pause times. A work-based solution for this could use an incremental garbage collection algorithm and perform increments of garbage collection work whenever memory is allocated. A time-based solution could use a periodic task that performs garbage collection work, separating the execution time of the garbage collector from the execution time of application threads.

1.5 Real-Time Garbage Collection

In order to be suitable for hard real-time systems, a garbage collection algorithm must fulfill the following properties:

(a) Unreachable memory must be reclaimed within a bounded amount of time.

(b) The garbage collector must not increase the response time of application tasks excessively.

(c) The release jitter in the presence of garbage collection must be comparable to the release jitter without garbage collection.

Property (a) has two important consequences: First, garbage collection cycles must finish within a bounded amount of time. Second, there is an upper bound on the amount of allocated memory for well-behaved applications (i.e., applications that do not allocate infinitely large amounts of memory, do not execute infinitely fast, and can reach only a bounded amount of memory at any one time). Such applications can produce only a bounded amount of garbage during a bounded time interval. As both the reachable and unreachable memory is bounded, the overall memory consumption is bounded as well. Many garbage collectors fulfill Property (a) for practical purposes, but it may be impossible to prove for certain scenarios. For hard real-time garbage collectors, it must be provable that the property is fulfilled even in worst-case scenarios.

Property (b) concerns the usefulness of a garbage collector rather than its correctness. In theory, a stop-the-world garbage collector could be taken into account for schedulability analysis. However, this would render virtually any realistic real-time system unschedulable. While some overhead for garbage collection is unavoidable, the overhead should be small enough to allow tasks with only little slack to meet their deadlines.

Hard real-time systems typically aim at achieving predictable behavior. Therefore, the release jitter should be as low as possible, i.e., garbage collection should not delay the execution of tasks. Property (c) states that hard real-time garbage collection should not increase the release jitter unduly. Similarly to Property (b), it is impossible to avoid all interference, but it should be minimized to keep the garbage collector useful for realistic systems.

Traversing the object graph is a relatively expensive task and performing it atomically would result in unacceptable system behavior. Real-time garbage collectors therefore must trace the object graph incrementally and use write barriers as described in Section 1.4.3.2. Pause times caused by atomic object graph traversal are not only an issue in real-time systems; the respective techniques can also be found in garbage collectors for general-purpose systems. Implementations and correctness proofs for incremental tracing algorithms date back several decades [10, 33] and are therefore not the focus of this thesis.

Compared to garbage collectors for general-purpose systems, three areas require special attention when developing a real-time garbage collector: scheduling, root scanning, and compaction.

1.5.1 Scheduling

Hard real-time garbage collectors may use work-based or time-based approaches. For work-based approaches, the amount of garbage collection work performed within each allocation must be bounded. Otherwise, it would not be possible to compute the WCET of tasks that perform memory allocations. Time-based approaches must be tractable by schedulability analysis; the execution time of garbage collection tasks must be bounded, and their scheduling must be analyzable.

While most real-time garbage collectors use time-based approaches, the garbage collector of the Jamaica VM [30] uses a work-based approach. Newer versions of that garbage collector however combine the work-based approach with a time-based approach [31]. The garbage collector presented in [3] also integrates work- and time-based approaches.

Time-based approaches can be categorized by the priority at which garbage collection work is scheduled. One approach is to schedule small increments of garbage collection work at top priority. The scheduler must be aware of the garbage collector and regularly preempt it in order to leave sufficient computation time to the application tasks. This approach is used in the Metronome garbage collector [4, 2] and a garbage collector by Kalibera [21]. An alternative approach is to schedule garbage collection tasks at some lower priority, e.g., at a lower priority than all hard real-time tasks of an application. This approach was proposed by Henriksson [16] and is sometimes also referred to as “slack-based” garbage collector scheduling.

Scheduling garbage collection tasks at a priority other than the top priority (e.g., the priority entailed by rate-monotonic priority assignment) is used by the Schism [27] and the Java RTS [7] garbage collectors. A fundamental advantage of this scheduling approach is that the scheduling granularity does not necessarily depend on the size of the garbage collection increments. As the garbage collector developed in the scope of this thesis aims at achieving low release jitter, it also uses this approach.

Time-based garbage collectors that execute at top priority often emphasize the "minimum mutator utilization", which is defined as the minimal fraction of processor time that is available to the application in a given time window. It has to be noted that this measure is hardly useful when scheduling the garbage collector at some lower priority, because higher-priority tasks can preempt the garbage collector. The mutator utilization therefore depends on the scheduling overhead rather than the garbage collector and equals 1 when ignoring these overheads.

1.5.2 Root Scanning

Performing root scanning atomically is acceptable for many systems, where human perception is the benchmark. In hard real-time systems however, timing requirements are usually tighter, and periods in the range of milliseconds or even below are not uncommon. Therefore, it is necessary to develop techniques where such timing requirements can be provably met.

Scanning of global reference variables can be made incremental by using write barriers. While this introduces some overhead, accesses to global variables are usually rare in object-oriented languages such as Java, and the overhead is consequently small. For local variables however, the overhead for write barriers would be unacceptably high—even copying a reference between processor registers would require the execution of the write barrier code. In Java bytecode, all local variables reside on the stack; scanning local variables for references is therefore also referred to as *stack scanning*.

One approach to lower the pause times caused by stack scanning is to scan only the stacks of individual threads atomically, instead of scanning all stacks atomically [2]. With this approach, inconsistencies could be caused by references that migrate between thread stacks. This issue can be solved by guarding accesses to global reference variables and reference fields on the heap with a *double barrier*. Such a barrier combines an incremental-update and a snapshot-at-beginning barrier; the object referenced by the old value is shaded as well as the object referenced by the new value. In Chapter 2, it is shown that inconsistencies can also be avoided by allocating new objects gray.

Techniques like *stacklets* [8] and *return barriers* [34] aim at reducing blocking times through cooperation between the application threads and the garbage collector. With stacklets, the application marks activated stack frames. Only those frames that have been activated since the last scan need to be scanned by the garbage collector. With a return barrier, the garbage collector scans only the topmost stack frames atomically; other stack frames are scanned concurrently. To avoid inconsistencies, threads scan stack frames themselves when returning to frames that have not yet been scanned by the garbage collector. Stacklets and return barriers are useful to reduce stack scanning overheads. However, their behavior is difficult to predict, which results in considerable pessimism for worst-case analyses and consequently makes them unsuitable for hard real-time systems.

Instead of blocking threads to scan their stacks, it is also possible to let threads scan their own stacks. As a thread does not manipulate its own stack while doing so, atomicity with regard to this thread is ensured. Atomicity is achieved implicitly rather than via explicit synchronization.

This idea can be found in garbage collectors by Doligez, Leroy and Gonthier [12, 11]. The root scanning mechanisms presented in Chapters 2 and 6 are also based on this idea. Jobs scan their own stacks at the end of their execution, where the stack is usually shallow. This reduces the cost for stack scanning without sacrificing predictability.

The garbage collector of the Jamaica VM [30] also makes threads responsible for scanning their own stacks. That approach allows task preemptions only at certain points of the program, which must be inserted frequently in the program to keep preemption delays low. At these points, the application task saves its local roots to a dedicated memory area. Compared to the approach presented in this thesis, root scanning in the Jamaica VM introduces notable memory and performance overheads.

Making threads responsible to scan their own stacks implies that the garbage collector must wait until the threads have done so. Chapter 3 presents a hardware solution to overcome this limitation. The proposed solution also reduces the memory bandwidth required for root scanning. While it introduces a notable hardware overhead, it might be a reasonable solution for systems with very stringent timing requirements.

1.5.3 Compaction

Heap fragmentation can increase the amount of memory that is needed to satisfy all allocation requests tremendously. For hard real-time systems, it must however be proven that all allocation requests can be served. Allowing fragmentation would require much larger memories than can be afforded for realistic systems. For hard real-time garbage collectors, handling fragmentation is therefore not an optimization, but rather a requirement.

It is possible to avoid external fragmentation by splitting objects into blocks of fixed size [30]. However, this trades external fragmentation for internal fragmentation, which results in considerable (though bounded) memory overheads. Also, performance suffers, because array accesses may require several levels of indirection. A more common solution is to relocate objects to compact the heap and eliminate fragmentation.

Relocating objects while application threads are executing potentially threatens data consistency. Writing to the original object location would result in loss of data if the respective object field already has been copied. Conversely, writing to the new location would result in data loss as well if the field has not yet been copied—the copying process would overwrite the newly written data. Stopping tasks while copying whole objects is however not a viable solution for hard real-time systems, because the relocation of large arrays would result in unacceptable pause times.

The blocking time for object relocation can be reduced by organizing arrays as *arraylets* [2]. With arraylets, arrays are split into chunks at the granularity of memory pages. An array "spine" points to the actual array data, introducing one level of indirection. As a consequence, it is never necessary to allocate contiguous memory that is larger than a memory page, which reduces fragmentation. However, it is still necessary to relocate objects and and array chunks that do not occupy full memory pages atomically. While the blocking times for relocation are bounded, they may be still too large for systems with stringent timing requirements.

Atomic copying can be avoided when using other means to ensure consistency while relocating objects. One way to achieve this to write to both the old and the new object location. An obvious disadvantage of this technique is that writes become more expensive. Furthermore, reversing the order of writing to the old and the new location could result in loss of data. Either the writes to both locations have to be executed atomically, or reordering of the writes has to be precluded.

The solution proposed by Kalibera [21] avoids strict ordering by restricting the points at which tasks may be preempted. As tasks cannot be preempted between the writes to the old and the new location, they appear to be atomic to all other tasks. However, this solution is applicable only to uniprocessor systems.

It is also possible to avoid atomic copying by using mechanisms such as compare-and-swap (CAS) operations. The Sapphire garbage collector [18] retries copying if it detects an update to the copied field by a mutator, and resorts to a CAS operation if the retries exceed a threshold. For correctness, it is necessary that mutators are properly synchronized. It requires different write barriers in different phases, and relies heavily on the fact that the Java memory does not require sequential consistency.

Another example for a garbage collector that uses CAS to ensure consistency during copying is the Stopless garbage collector [26]. It first copies objects to a "wide" representation, which includes an additional status word for each word of the original object. In a second step, the object is copied to the final location. Double-word CAS operations on the wide object representation are used to detect updates by the mutators. If an update is detected, copying is retried. In general, it is impossible to bound the number of retries and consequently to bound the WCET. Therefore, this approach is unsuitable for hard real-time systems.

Chapter 6 describes a hardware unit that enables preemptible copying. Object relocation becomes transparent to the application tasks. The hardware unit presented in that chapter is an extension of a similar hardware unit presented in [29] and is suitable for use in CMPs. Hardware support for transparent object copying has also been proposed by Nielson and Schmidt [24] and is used in the SHAP processor [35]. Unlike these approaches, which implement substantial parts of the garbage collector in hardware, the hardware unit presented in this thesis aims at a small and simple implementation.

1.5.4 Real-Time Garbage Collection on a Chip Multi-Processor

Humans can process only very limited amounts of information in parallel [14]. Therefore, we find it considerably more difficult to develop and understand parallel systems than sequential systems. When executing a parallel program on a uniprocessor, the execution of the program's threads is interleaved. Between preemptions, the behavior of a thread can be analyzed sequentially, which simplifies analysis considerably. On multi-processors however, such a simplification is not possible—instead of being interleaved, execution is truly parallel.

Apart from being more difficult to understand, the development of applications for CMPs also faces technical challenges. The main reason for this is that it is more difficult to ensure correct and efficient synchronization of threads. On uniprocessors, it is possible to ensure atomic execution of a critical section by preventing the scheduler from being triggered. This can be achieved by disabling interrupts, which is usually a cheap operation. On multi-processors, threads can try to acquire a lock while it is held by another thread, even if preemption is disabled while the lock is held. Moreover, in real-time systems, there must be an ordering on the threads that are allowed to enter the lock, if multiple threads wait for the lock simultaneously. Consequently, real-time locks for multi-processors require book-keeping that complicates the lock implementation and makes lock acquisition more expensive.

On a uniprocessor, preventing preemption is sufficient to ensure atomic execution of a code fragment, even if tasks that might interfere with the critical section do not use any synchronization mechanisms. Synchronization therefore can take place asymmetrically. The equivalent on a multi-processor is to stop the execution of tasks on all other processor cores. This requires explicit communication between the cores, as the stopping of other tasks does not happen implicitly as it is the case on a uniprocessor. Asymmetric synchronization is a solution if the critical section is executed infrequently but synchronization would have to take place frequently. An example is atomic scanning of local variables by a garbage collector, where symmetric synchronization would result in considerable performance overheads for the application tasks. As explicit stopping of tasks is more expensive than simply disabling interrupts, there are probably fewer scenarios on multi-processors than on uniprocessors where asymmetric synchronization is beneficial.

An area where the implementation of garbage collectors in particular faces challenges on CMPs is root scanning. On multi-processors, it may not be possible to determine the root set of tasks that execute on other cores. While it is possible to access thread-local data that is mapped to main memory, (e.g., the stack), this may not be possible for reference variables that are located in core-local storage such as registers or scratch-pad memory. To overcome this limitation, some form of cross-core cooperation is necessary.

Multi-processors commonly use more relaxed memory models than sequential consistency. This must be taken into account for the design of a garbage collector. Chapter 4 investigates the effects of the memory model on the garbage collector and identifies parts of the garbage collection algorithm where reordering of memory accesses must be prevented.

1.5.5 Time and Space Bounds

For hard real-time systems, the garbage collector must provably be able to keep up with the application's memory allocation demands. Depending on the amount of available memory, the application tasks' allocation rates and the maximum amount of live memory, it is possible to compute a maximum garbage collection period [28]. If garbage collection cycles can be completed within that period, the garbage collector can keep up with the application.

The garbage collector's WCET can be calculated with WCET analysis techniques that are also used to compute the WCETs of application tasks. The response time can be calculated by applying scheduling theory appropriately. These areas of research are relatively mature; they apply to any hard real-time system, regardless of the memory management mechanisms used. In contrast, analyses to compute allocation rates and the maximum amount of live memory are only required in hard real-time systems that use automatic memory management. Consequently, their development goes hand in hand with the progress in research on hard real-time garbage collection. Chapter 5 presents a technique to compute the maximum allocation rate of a task by re-using the infrastructure for WCET computation. Also, it includes formulas to compute the actual memory consumption of an object for different object layouts.

1.6 Contributions

In the following, the contributions of this thesis to the state of the art in the area of hard real-time garbage collection are described. The contributions comprise theoretical considerations, software algorithms, and hardware support for garbage collection.

Hard real-time garbage collection on a CMP The probably most important contribution of this thesis is that it provides evidence that garbage collection is viable for hard real-time systems on CMPs. The results presented in Chapter 6 show that garbage collection does not necessarily lead to unpredictable behavior. The garbage collector developed in the scope of this thesis uses only on mechanisms that are analyzable. Therefore, it enables rigorous analysis of garbage-collected hard real-time systems on CMPs. In measurements, release jitter in the range of 200 μs was observed on a multi-processor with 8 cores clocked at 100 MHz. The observed jitter is only slightly higher than in a system without garbage collection.

Non-blocking stack scanning A low-overhead stack scanning scheme which does not require high-priority tasks to be blocked is proposed in Chapter 2. Tasks scan their own stacks at the end of a job's execution, which eliminates pauses for stack scanning. As the stack is usually shallow at the end of a job's execution, this also minimizes the related overhead. The correctness of the approach is proven in Chapter 2. Chapter 6 extends this scheme with events for stack scanning of tasks with long periods. This reduces the time the garbage collector has to wait for the completion of stack scanning.

Hardware support for stack scanning A novel hardware mechanism to support efficient stack scanning, which eliminates delays for the garbage collector, is presented in Chapter 3. This mechanism enables the garbage collector to start at any time, without having to wait for the tasks to scan their stacks. Additionally, it reduces the memory bandwidth required for stack scanning. While this approach requires specialized hardware, it may be an option for systems with exceptionally stringent timing requirements.

Investigation of relaxed memory models In Chapter 4, the effects of relaxed memory models on garbage collectors are investigated. Furthermore, a simple cache design that is compatible with the Java memory model is presented. The chapter highlights where the Java memory model might impair the correctness of garbage collection and identifies conditions to ensure correctness. The considerations in Chapter 4 are more general than prior research on this topic, which usually inspected only specific garbage collection algorithms on specific hardware platforms.

Maximum allocation rates A method to compute allocation rates of tasks, which is based on WCET analysis techniques, is proposed in Chapter 5. As the presented technique can reuse infrastructure for WCET analysis, the implementation of an allocation rate analysis is simplified considerably. Furthermore, Chapter 5 presents formulas to compute the actual memory consumption with different object layouts, including alignment requirements.

Hardware support for object relocation Chapter 6 presents a hardware unit that enables transparent object relocation on CMPs. Objects can be accessed at any time, even while they are being copied. Unlike prior approaches, it aims at a small and simple implementation and is targeted at CMPs. The proposed hardware unit does not impair the worst-case latencies of memory accesses and is therefore suited for use in hard-real-time systems.

Prototype implementation In the course of this thesis, a prototype implementation of the garbage collector was developed. This enables measurements on an actual system and the observation of effects that might have been abstracted away in simulations. This is particularly important for the evaluation of hardware, where the effects on the resource usage and the critical path of the overall system are difficult to estimate on a purely theoretical basis.

Bibliography

[1] American National Standards Institute. *American National Standard for information systems: programming language: C: ANSI X3.159-1989*. American National Standards Institute, New York, USA, December 1989.

[2] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *7th ACM & IEEE International Conference on Embedded Software*, pages 249–258, Salzburg, Austria, September 2007. ACM Press.

[3] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *8th ACM International Conference on Embedded Software*, pages 245–254, Atlanta, GA, 2008. ACM Press.

[4] David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM SIGPLAN Notices 38(7), pages 81–92, San Diego, CA, June 2003. ACM Press.

[5] David F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235, Budapest, Hungary, June 2001. Springer-Verlag.

[6] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[7] Eric J. Bruno and Greg Bollella. *Real-Time Java Programming: With Java RTS*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[8] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 33(5), pages 162–173, Montreal, Canada, June 1998. ACM Press.

[9] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.

[10] Edsgar W. Dijkstra, Leslie 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(11):965–975, November 1978.

[11] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 70–83, Portland, OR, January 1994. ACM Press.

[12] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *20th Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Charleston, SC, January 1993. ACM Press.

[13] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In David Watt, editor, *Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 82–93. Springer Berlin / Heidelberg, 2000.

[14] Graeme S. Halford, Rosemary Baker, Julie E. McCredden, and John D. Bain. How many variables can humans process? *Psychological Science*, 16(1):70–76, 2005.

[15] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[16] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[17] Gerard J. Holzmann. The power of 10: Rules for developing safety-critical code. *IEEE Computer*, 39(6):95–97, 2006.

[18] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3–5):223–261, 2003.

[19] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, August 2011.

[20] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[21] Tomas Kalibera. Replicating real-time garbage collector for Java. In *7th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '09, pages 100–109, Madrid, Spain, September 2009. ACM Press.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[23] Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James J. Hunt, Johan Olmütz Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Jan Vitek, and Andy Wellings. *Safety-Critical Java Technology Specification, Public draft*. 2011.

[24] Kelvin D. Nilsen and William J. Schmidt. Cost-effective object-space management for hardware-assisted real-time garbage collection. *Letters on Programming Language and Systems*, 1(4):338–354, December 1992.

[25] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. *ACM Transactions on Programming Languages and Systems*, 29(4):1–43, August 2007.

[26] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. Stopless: A real-time garbage collector for multiprocessors. In Greg Morrisett and Mooly Sagiv, editors, *6th International Symposium on Memory Management*, pages 159–172, Montréal, Canada, October 2007. ACM Press.

[27] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 45(6), pages 146–159, Toronto, Canada, June 2010. ACM Press.

[28] Martin Schoeberl. Scheduling of hard real-time garbage collection. *Real-Time Systems*, 45(3):176–213, 2010.

[29] Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *6th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '08, pages 77–84, Santa Clara, CA, September 2008. ACM Press.

[30] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas Books, 2002.

[31] Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In Jan Vitek and Doug Lea, editors, *9th International Symposium on Memory Management*, pages 11–20, Toronto, Canada, June 2010. ACM Press.

[32] John A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10 –19, oct 1988.

[33] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990.

[34] Taiichi Yuasa. Return barrier. In *International Lisp Conference*, 2002.

[35] Martin Zabel, Thomas B. Preußer, Peter Reichel, and Rainer G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 59–62. IEEE Computer Society Press, 2007.

[36] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, November 1990.

2 Non-blocking Root Scanning*

By Wolfgang Puffitsch and Martin Schoeberl

Abstract

Root scanning is a well known source of blocking times due to garbage collection. In this paper, we show that root scanning only needs to be atomic with respect to the thread whose stack is scanned. We propose two solutions to utilize this fact: (a) block only the thread whose stack is scanned, or (b) shift the responsibility for root scanning from the garbage collector to the application threads. The latter solution eliminates blocking due to root scanning completely. Furthermore, we show that a snapshot-at-beginning write barrier is sufficient to ensure the consistency of the root set even if local root sets are scanned independently of each other. The impact of solution (b) on the execution time of a garbage collector is shown for two different variants of the root scanning algorithm. Finally, we evaluate the resulting real-time garbage collector in a real system to confirm our theoretical findings.

2.1 Introduction

Tracing garbage collectors traverse the object graph to identify the set of reachable objects. The starting point for this tracing is the *root set*, a set of objects which is known to be directly accessible. On the one hand, these are references in global (`static` in Java) variables, on the other hand these are references that are local to a thread. The latter comprise the references in a thread's runtime stack and thread-local CPU registers. The garbage collector must ensure that its view of the root set is consistent before it can proceed, otherwise objects could be erroneously reclaimed.

For stop-the-world garbage collectors, the consistency of the object graph is trivially ensured. Incremental garbage collectors however require the use of *barriers*, which enforce the consistency of marking and the root set [3, 6, 19, 20]. While barriers are an efficient solution for the global root set, they are considered to be too inefficient to keep the local root sets consistent. Even frequent instructions like storing a reference to a local variable would have to be guarded by such a barrier, which would cause a considerable overhead and make it difficult if not impossible to compute tight bounds for the worst case execution time (WCET). The usual solution to this problem is to scan the stacks of all threads in a single atomic step and stall the application threads while doing so.[1] The atomicity entails that the garbage collector may not be preempted while it scans a thread's stack, which in turn causes a considerable release jitter even for high-priority threads.

*This chapter has been previously published as: Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, JTRES '08, pages 68–76, Santa Clara, California, USA, September 2008. ACM Press.

[1]The former implies the latter on uniprocessors, but not on multi-processors.

However, the atomicity is only necessary w. r. t. the thread whose stack is scanned, because a thread can only modify its own stack. If the garbage collector scans a thread's stack, the thread must not execute and atomicity has to be enforced. Other mutator threads are allowed to preempt the stack scanning thread. If a thread scans its own stack, it is not necessary to prohibit the preemption of the thread—when the thread continues to execute, the stack is still in the same state and the thread can proceed with the scanning without special action. Consequently, preemption latencies due to root scanning can be avoided. With such a strategy, it is also possible to minimize the overhead for root scanning. It can be scheduled in advance such that the local root set is small at the time of scanning.

In this paper, we show the feasibility of this approach. We present two solutions for periodic and sporadic threads, and evaluate their trade-offs. Furthermore, we show how the worst case time until all threads have scanned their stacks can be computed.

We consider a garbage collection approach that uses a separate thread for the garbage collector, as suggested by Henriksson [10]. Such an approach is also used by Robertz [12] and in IBM's Metronome garbage collector [2] to improve timely determinism of garbage collection. For our experiments, we used a concurrent-copy garbage collector as described in [15] and [17]. Our theoretical findings are however not specific to any particular tracing garbage collection algorithm.

The contributions of this paper are:

- Stack scanning performed by the garbage collection thread needs only be atomic w. r. t. to the thread whose stack is scanned.

- Delegating the stack scanning to the mutator threads completely avoids blocking time due to root scanning.

- A snapshot-at-beginning write barrier [20] is sufficient to protect against losing an object when a reference migrates from a not-yet-scanned stack to a scanned stack.

The paper is organized as follows: In the following section, we describe related work on root scanning in real-time garbage collectors. In Section 2.3 we present our approach to root scanning and establish its correctness. An implementation of the approach is evaluated in Section 2.4. Section 2.5 concludes the paper and presents an outlook on future work.

2.2 Related Work

The idea of delegating local root scans to the mutator threads is proposed by Doligez, Leroy and Gonthier in [8] and [7]. They point out that this allows for more efficient code and reduces the disruptiveness of garbage collection. Mutator threads should check an appropriate flag from time to time and then scan their local root set. However, the authors remain vague on when the mutators should check this flag and do not investigate the effect of various choices. As they aim for efficiency rather than real-time properties, they do not consider a thread model with known periods and deadlines.

The approach presented in this paper builds to some degree on an approach by Schoeberl and Vitek [17]. They propose a thread model which does not support blocking and where threads cannot retain a local state across periods. They also propose that the garbage collector runs at the lowest priority, which entails that the stacks of all threads are empty when a garbage collection cycle starts. Consequently, the garbage collector only needs to consider the global root set. Although the initial motivations differ, the work presented in this paper can be regarded as generalization of this earlier approach, especially w. r. t. the thread model.

In the JamaicaVM's garbage collector, the mutator threads are also responsible of keeping their root set up to date [18]. However, the JamaicaVM's garbage collector employs a work-based approach, which means that the garbage collector does not execute in a separate thread—garbage collection is distributed among the mutator threads. Therefore, considerations on the correctness and execution time are hardly comparable. In [18], the average overhead for root scanning is estimated as 11.8%.

Yuasa claims in [20] that the time for saving the root set of a thread can be made sufficiently short by using block transfer mechanisms. Our experience however shows that the atomic copying of the stack takes too long to keep the jitter reasonably low for tasks with sub-millisecond periods.

A later approach by Yuasa [21] introduces a *return barrier*. In a first step, the garbage collector scans the topmost stack frames of all threads atomically. Then it continues to scan one frame at a time. When a thread returns to a frame that has not yet been scanned, it scans it by itself. Return instructions consequently carry an overhead for the respective check. Furthermore, the proposed policy makes it difficult to compute tight WCET bounds, because it is difficult to predict when a scan by a thread is necessary. A critical issue is also that the topmost frames of *all* threads have to be scanned in a single atomic step. The worst case blocking time therefore increases with the number of threads. An overhead of 2 to 10% due to the necessary checks for the return barrier is reported in [21]. Depending on the configuration, 10 to 50 μs were measured as worst case blocking time on a 200 MHz Pentium Pro processor.

Cheng et al. propose a strategy for lowering the overhead and blocking of stack scanning in [5]. The mutator thread marks the activated stack frames and the garbage collector scans only those frames which have been activated since the last scan. However, this technique is only useful for the average case. In the worst case, it is still necessary to scan the whole stack atomically.

2.3 Preemptible Root Scanning

Due to the volatile nature of a thread's stack, the garbage collector and the mutator thread must cooperate for proper scanning. If a thread executes arbitrary code while its stack is scanned, the consistency of the retrieved data cannot be guaranteed. Therefore, a thread is usually suspended during a stack scan. In order to ensure the consistency of the root set, the stack is scanned atomically to avoid preemption of the garbage collector and inhibit the execution of the respective thread.

When the garbage collection thread scans a stack it is not allowed to be preempted by that thread. However, as the stack is thread local, preemption by any other mutator thread is not an issue. Therefore, a thread will only suffer blocking time due to the scanning of its own stack. A high priority thread, which has probably a shallow call tree, will not suffer from the scanning of deeper stacks of more complex tasks. The protection of the scan phase can be achieved by integrating parts of the garbage collection logic with the scheduler. During stack scanning, only the corresponding mutator thread is blocked.

We generalize this idea by moving the stack scanning task to the mutator threads. Each thread scans its own stack at the end of its period. In that case mutual exclusion is trivially enforced: the thread performs either mutator work or stack scanning. The garbage collector initializes a garbage collection period as usual. It then sets a flag to signal the threads that they shall scan their stacks at the end of their period. When all threads have acknowledged the stack scan, the garbage collector can scan the static variables and proceed with tracing the object graph. Why the static variables are scanned after the local variables is discussed in Section 2.3.1.4.

By using such a scheme, it is not necessary to enforce the atomicity of a stack scan. Furthermore, the overhead for a stack scan is low; at the end of each period, the stack is typically small if not even empty. Such a scheme also simplifies exact stack scanning, because stack scanning takes place only at a few predefined instants. Instead of determining the stack layout for every point at which a thread might be preempted, it is only necessary to compute the layout for the end of a period. The required amount of data is reduced considerably as well, which lowers the memory overhead for exact stack scanning.

2.3.1 Correctness

We identified three properties which must be fulfilled to ensure the correctness of our proposed garbage collection scheme:

- The local root set remains unmodified during scanning (Property 1).
- Modification of the object graph during scanning does not void correctness (Property 2).
- Migration of references between local root sets does not void correctness (Property 3).

Property 1 refers to the situation where a thread moves a reference from a not-yet-scanned local variable to a scanned local variable and clears the not-yet-scanned variable afterwards. In such a case, the respective root reference would be lost.

Property 2 addresses the fact that two local root sets are scanned at different times. They do not necessarily correspond to the same state of the object graph. It must be shown that this does not invalidate the correct view of the object graph at mark time. This allows threads to execute between the scanning of two local root sets.

Property 3 is similar to Property 1, but now a reference is moved between local root sets. This is also an issue if the stack of each thread is scanned atomically. The error scenario for this property is when a reference is transferred from a not-yet-scanned stack to a scanned stack and removed from the not-yet-scanned stack afterwards.

In the following, we will sketch the proofs why the Properties 1-3 are fulfilled in our proposed garbage collection scheme. The symbols we use throughout this section are described in Table 2.1. An object graph can also be written as tuple of its nodes, edges and root set: $G = \langle V(G),\ E(G),\ R(G)\rangle$. We consider only the objects that are referenced by local variables

Table 2.1: Symbol descriptions

Symbol	Description
G	Object graph
τ_i	Thread i
$V(G)$	Nodes of an object graph G (objects)
$E(G)$	Edges of an object graph G (references)
$R(G)$	Root set of an object graph G, $R(G) \subseteq V(G)$
$\rho(G)$	Reachability function, set of reachable nodes in G
$\mu(G)$	Marking function, approximation of $\rho(G)$

as part of the root set $R(G)$. Objects which are referenced by static variables are handled separately, as described in Section 2.3.1.4.

We define the operations $\cup$ and $\supseteq$ on object graphs as component-wise application of these operations to the nodes, edges and root sets of two object graphs:

$$\begin{aligned} G \cup G' &:= \langle V(G) \cup V(G'),\ E(G) \cup E(G'),\ R(G) \cup R(G') \rangle \\ G \supseteq G' &:= V(G) \supseteq V(G') \wedge E(G) \supseteq E(G') \wedge R(G) \supseteq R(G') \end{aligned}$$

We define a relation $G \bowtie G'$ ("G bow tie G'") of two graphs as follows:

$$\begin{aligned} G \bowtie G' \Leftrightarrow \forall v_1 \in V(G) \cap V(G') :& \\ \langle v_1, v_2 \rangle \in E(G) \Rightarrow \langle v_1, v_2 \rangle \in E(G'),&\ v_2 \in V(G') \wedge \\ \langle v_1, v_2 \rangle \in E(G') \Rightarrow \langle v_1, v_2 \rangle \in E(G),&\ v_2 \in V(G) \end{aligned}$$

We call two graphs G and G' *out-edge consistent* if $G \bowtie G'$. This relation holds if out-edges of shared nodes are present in both graphs. The out-edge consistency of object graphs is important when reasoning about how different threads see a shared object graph. Assuming the out-edge consistency of sub-graphs greatly simplifies formal handling.

Due to the relatively loose requirements on data consistency of the Java memory model [9], two threads do not necessarily share the same view of an object.[2] While one thread assumes that a field `obj.f` references an object `x`, another thread may assume that the same field references an object `y`. Such a situation is problematic for a garbage collector, because it would leave either object `x` or object `y` unvisited.

Such inconsistencies originate from the fact that threads are allowed to cache data locally. On uniprocessors, cache coherence is not an issue—all threads share the same cache—but thread-local registers may be used to store reference fields. As these registers are scanned for the computation of the local root set of a thread, it is ensured that references cached in registers are visited as well as references stored in the heap. It is therefore safe to assume that all threads have an out-edge consistent view of the object graph.

[2]Proper synchronization of course eliminates coherence problems, but the authors consider correctness of synchronization to be an unreasonably strong precondition. Flawed synchronization should not cause a failure of the garbage collector.

Table 2.2: Description of operations

Operation	Description
f_{new}	Create a new object
f_{load}	Add a reference to the root set
f_{kill}	Remove a reference from the root set
f_{write}	Replace an edge with a new edge

On multi-processors, cache coherence must be ensured to allow consistent tracing of the object graph. It is then also safe to assume the out-edge consistency of the threads' views of the object graph. It is beyond the scope of this paper how this cache coherence can be achieved efficiently.

The reachability function is formally defined as follows:

$$\begin{aligned}\rho(G) := \{v \in V(G) \mid \exists \text{ a chain of edges } e_1,\dots,e_n \in E(G) :\\ e_1 = \langle r, v_1\rangle, e_2 = \langle v_1, v_2\rangle, \dots, e_n = \langle v_{n-1}, v\rangle,\ r \in R(G)\}\end{aligned}$$

In other words, a node v is reachable, if there exists a path from a root node $r \in R(G)$ to v.

The reachability function ρ has three important properties:

- ρ is *monotone*:
  $G \supseteq G' \Rightarrow \rho(G) \supseteq \rho(G')$
- ρ is *closed*:
  $G' = \langle V(G),\ E(G),\ R(G) \cup \{v\}\rangle,\ v \in \rho(G) \Rightarrow \rho(G) = \rho(G')$
- ρ is *distributive* for out-edge consistent graphs:
  $G \bowtie G' \Rightarrow \rho(G \cup G') = \rho(G) \cup \rho(G')$

The monotony of ρ is a fairly intuitive property: adding a node or an edge to a graph cannot make fewer nodes reachable. ρ is closed in the sense that adding a reachable node to the root set does not change the set of reachable nodes. This property becomes clearer when considering that root nodes are just known to be reachable a priori, but are not special in any other way.

The distributivity of ρ allows to determine the set of reachable node globally or locally. Consider two threads τ and τ' and their views of the object graph, G and G'. It is possible to compute $R(G) \cup R(G')$ and start tracing from that root set; it is also possible to compute $\rho(G)$ and $\rho(G')$ independently and merge the results. Distributivity ensures that the final result is the same for both approaches.

An important observation is that a thread may only access objects that are reachable, or create new objects. This insight may seem trivial; still, it should be kept in mind for the following considerations.

There are four fundamental operations on an object graph, which are shown in Table 2.2. We use the following formal definitions for these graph operations:

$$
\begin{aligned}
f_{new}(G) &:= \langle V(G) \cup \{v\},\ E(G),\ R(G) \rangle,\ v \notin V(G) \\
f_{load}(G) &:= \langle V(G),\ E(G),\ R(G) \cup \{v\} \rangle,\ v \in V(G) \\
f_{kill}(G) &:= \langle V(G),\ E(G),\ R(G) \setminus \{v\} \rangle \\
f_{write}(G) &:= \langle V(G),\ (E(G) \setminus \{\langle v_1, v_2 \rangle\}) \cup \{\langle v_1, v_3 \rangle\},\ R(G) \rangle
\end{aligned}
$$

It must be noted that each thread can only operate on its local view of the object graph. Therefore, f_{load} and f_{kill} can only modify the local root set of a mutator thread. Furthermore, these are "minimal" definitions; they have to be extended for actual garbage collection algorithms, e. g., to model a write barrier.

We assume that any changes to the shape of the object graph retain the out-edge consistency of the local views. As already mentioned, this is a safe assumption for uniprocessors. For multi-processors, this assumption is only safe if cache coherence is ensured.

2.3.1.1 Property 1: Atomicity

The runtime stacks of any two threads are disjoint—otherwise they could not execute independently of one another. There is no way how a Java thread can access the stack of any other thread. Only JVM-internal threads like the garbage collection thread can access the stack of another thread. Therefore, the execution of a Java thread cannot modify the stack of a different thread.

If the garbage collector scans a thread's stack, it is hence not necessary to enforce absolute atomicity. It is only necessary to prevent the thread whose stack is scanned from executing—all other threads cannot modify the stack of this thread. Such a strategy needs support from the scheduler and does not eliminate blocking completely. However, blocking can be reduced considerably, because threads only need to wait while their own stack is scanned and high frequency threads typically have a shallow stack.

It also follows that a thread may be preempted while scanning its own stack without compromising the consistency of the scanned data. When the thread continues execution, the stack is unchanged and the local root set is the same as if the thread had not been preempted. It is not necessary to enforce atomicity if a thread scans its own stack.

2.3.1.2 Property 2: Independence

Due to the distributivity of ρ, it is sufficient to scan the stack of one thread at a time to determine the global set of reachable nodes. However, it needs to be shown that the actions of one thread do not interfere with the view of the object graph of another thread.

f_{new} allocates a new object, which is not yet visible to other threads; it can therefore be considered as local action. f_{load} and f_{kill} only modify the local root set and hence are local actions as well. Consequently, threads can only communicate through the f_{write} operation. This operation however retains the out-edge consistency, i.e., if one thread changes the shape of the object graph, other threads see the updated object graph. Therefore, it is not necessary for one thread to know anything about the root set or the actions of a different thread for the proper computation of ρ. The root sets can therefore be scanned independently of one another.

2.3.1.3 Property 3: Consistency

It is necessary that the local root sets and the object graph are consistent such that no less than the actually reachable nodes are marked during tracing. It must be ensured that the fundamental graph operations cannot break this consistency. Please note that the following reasoning refers to the root scanning phase and not to the marking phase. This means that no node is *black* in terms of Dijkstra's tricolor abstraction [6].

In order to keep the root set consistent for f_{new}, it is necessary to add references to new objects to the root set. Otherwise, recently created objects could erroneously appear unreachable. In terms of the tricolor abstraction, this means that new objects are allocated gray during root scanning. The f_{new} operation therefore has to be extended as follows:

```
$f_{new}(G, type) :=$ {
  $v :=$ GC.allocate($type$);
  GC.markGray($v$);
  $V(G) := V(G) \cup \{v\}$;
}
```

f_{load} can only add references to the root set which are already reachable. Due to the closed nature of ρ, this does not change the set of reachable nodes. If a snapshot of the object graph from the beginning of the garbage collection cycle is maintained, these operations have no effect.

f_{kill} removes a reference from the root set and can therefore change the set of reachable nodes. If an object is still reachable after this operation, it must be reachable through a path which starts at a different root node. As no node is black during root scanning, this path will be encountered by the marking function. If the most recent state of the object graph is traced, it is not necessary to take measures to prevent the loss of a root reference.

If it can be shown that the side effects of f_{write} allow that both the snapshot of the object graph and the current object graph are traced, both f_{load} and f_{kill} do not require any special action.

We define a *history graph* H, which subsumes changes to an object graph. With G being an object graph and $f_0 \dots f_n \in \{f_{new}, f_{load}, f_{kill}, f_{write}\}$ being a sequence of graph operations, it is defined as

$$\begin{aligned} V(H) &:= V(G \cup f_0(G) \cup f_1(f_0(G)) \cup \ldots \cup f_n(\ldots(f_0(G)))) \\ E(H) &:= E(G \cup f_0(G) \cup f_1(f_0(G)) \cup \ldots \cup f_n(\ldots(f_0(G)))) \\ R(H) &:= R(f_n(\ldots(f_0(G)))) \end{aligned}$$

Such a history graph safely approximates the shape of both the initial object graph and the current object graph. Therefore, the considerations for object graph operations above apply to a history graph. An implementation of f_{write} which allows to safely approximate a history graph is consequently sufficient to maintain the correctness of ρ.

In [1], a *double barrier*, which saves both references on an assignment, is suggested to maintain the consistency of the root set. The reasoning behind this is that references could otherwise migrate from the local root set of one thread to the root set of a different thread without being noticed. The critical operation for this assumption is when a reference migrates from a not-yet-scanned local root set to a local root set which already has been scanned. However, a thread may only access reachable objects and can therefore only add references to its root set which are already reachable. Therefore, the respective reference is also already reachable from the scanned root set. In a history graph, this reference remains visible, even if the addition of the reference to the root set is ignored.

We now show that a snapshot-at-beginning barrier as suggested in [20] is sufficient to approximate a history graph. Together with the independence of local root scans and the considerations on the graph operations above, this proves that such a double barrier is indeed not necessary.

We use the following abstract definition of f_{write}, which models the Yuasa snapshot-at-beginning write barrier:

```
$f_{write}(G, \langle v_1, v_2\rangle, \langle v_1, v_3\rangle) :=$ {
  if (color($v_2$) = white) {
    GC.markGray($v_2$);
  }
  $E(G) := (E(G) \setminus \{\langle v_1, v_2\rangle\}) \cup \{\langle v_1, v_3\rangle\}$;
}
```

This definition may look unusual when comparing it to actual implementations. It is however just a translation of such an implementation to the terminology used throughout this paper and does not add any special semantics. The barrier shades references that are overwritten gray. The overwritten reference is therefore visible for the tracing algorithm; in effect, a virtual snapshot of the object graph is retained.

The marking function μ is an algorithm to compute the set of reachable nodes. Starting from the root set, the object graph is traversed, until no new objects can be reached. It can be assumed that the marking function μ safely approximates the reachability function ρ if no graph transformations occur. This assumption is fundamental for *any* tracing garbage collection algorithm and must be proven for the correctness of any garbage collector, independent of root scanning and even for stop-the-world garbage collectors. Note that our considerations apply to the root scanning phase and not marking itself. Our goal is to show that the root set is computed correctly. Proving the correctness of concurrent marking is a different issue; such proofs can be found in [6] or [20].

f_{load} and f_{kill} change only the root set of a graph; as the root set of the history graph is the root set after any transformation, μ trivially approximates the history graph w. r. t. these operations. Shaded allocation of new objects ensures that f_{new} is handled correctly. This leaves to be shown that the write barrier indeed allows the approximation of a history graph.

Figure 2.1 shows the effect of an $f_{write}(G) = G'$ operation. The history graph includes all encountered edges, i.e., the replaced edge $\langle v_1, v_2\rangle$ and the new edge $\langle v_1, v_3\rangle$. Consequently, the following equation holds for $H = G \cup G'$:

$$v_1 \in \rho(G) \Rightarrow (v_2 \in \rho(H) \wedge v_3 \in \rho(H)) \qquad (2.1)$$

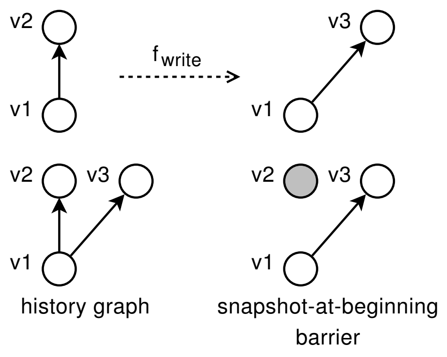


Figure 2.1: History graph and snapshot-at-beginning barrier

The snapshot-at-beginning barrier does not retain the edge $\langle v_1, v_2\rangle$, but it marks v_2 to be traced. The appropriate equation is:

$$(v_1 \in \mu(G) \Rightarrow v_3 \in \mu(G')) \wedge v_2 \in \mu(G') \tag{2.2}$$

which can be reformulated as

$$(v_1 \in \mu(G) \Rightarrow v_2 \in \mu(G') \wedge v_3 \in \mu(G')) \wedge v_3 \in \mu(G') \tag{2.3}$$

These equations entail that μ induces a superset of ρ if operating on the same graph. Therefore, a snapshot-at-beginning barrier is sufficient to ensure the correctness of incremental garbage collection.

2.3.1.4 Static Variables

The considerations above cover only the scanning of thread-local roots, but left out static variables. They can be modeled with an immutable root, which points to a virtual array that contains the static variables. This virtual array can then be handled like any other object and the scanning of static variables becomes part of the marking phase. As marking has to take place after root scanning, static variables have to be scanned after the local root sets.

This is not an arbitrary limitation—it is easy to construct an example where scanning static references before scanning local variables breaks the consistency of a garbage collector that uses a snapshot-at-beginning write barrier. Consider the case where during the scanning of static variables a reference is transferred from a local variable to a static variable which has already been scanned. The value of the local variable might be lost until it is scanned, and the new value of the static variable is not visible to the garbage collector. The variable already has been scanned, and the snapshot-at-beginning barrier retains the old value, but does not treat the new one. Therefore, the respective object may erroneously appear unreachable to the garbage collector.

In contrast, we were not able to construct an example where a reference is lost if the scanning of static variables takes place *after* the scanning of local variables. As it can then be regarded as part of the marking phase, the correctness proof of the snapshot-at-beginning algorithm [20] apply to this part of the algorithm—it is simply impossible to find a counterexample. Actually,

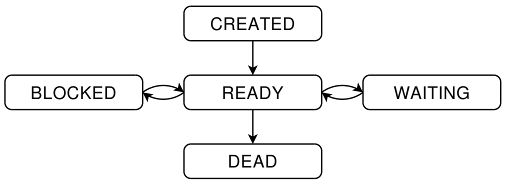


Figure 2.2: Thread Model

the theoretical findings presented above were sparked by our unavailing search for such an example.

2.3.2 Execution Time Bounds

Now that we have established the correctness of our approach, we can analyze the effects on the timing behavior of the garbage collection thread. We found two solutions to apply the theoretical results: The first solution can be applied only to periodic tasks, while the second solution can be applied to sporadic tasks as well.

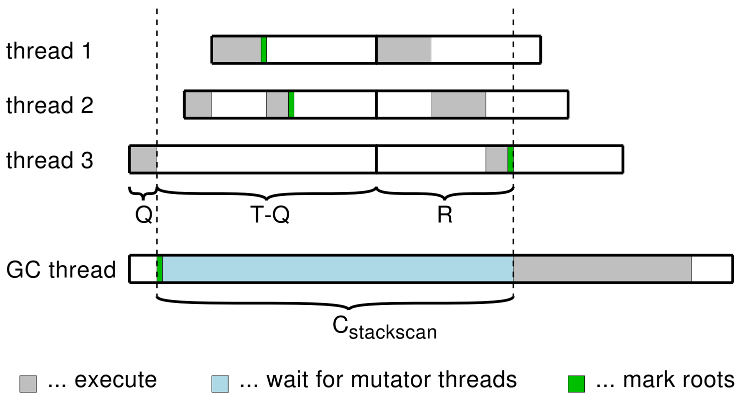
2.3.2.1 Thread Model

We assume that all threads are either periodic or have at least a known deadline. This is a reasonable assumption for real-time threads: it is impossible to decide whether a task delivers its results on time if no deadline or period is known.

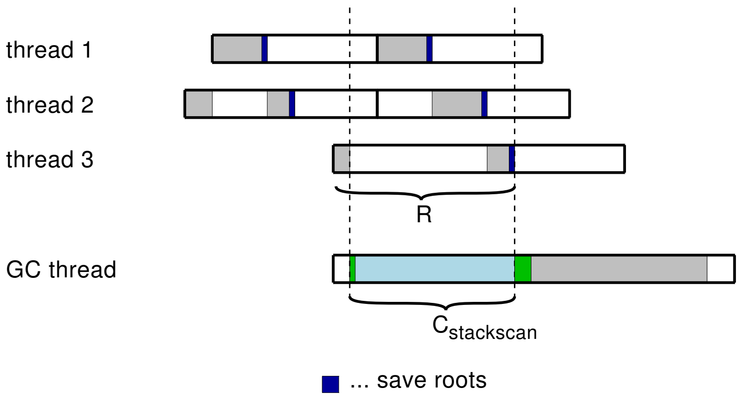
The thread model has five states: CREATED, READY, WAITING, BLOCKED and DEAD. Initially, a thread is in state CREATED. When a thread gets available for execution, it goes to the READY state. When it has finished execution for a period it becomes WAITING. At the start of the next period, it goes to the READY state again. If a thread terminates, it becomes DEAD. Threads are in state BLOCKED while they wait for locks or I/O operations. The time between the instant at which a thread becomes READY until it goes to state WAITING must be bounded—if it is not WAITING when its deadline arrives, it has missed the deadline. Figure 2.2 visualizes the possible state transitions of the thread model.

For the calculation of the execution time bounds, we assume that threads scan their stack when they become WAITING. For periodic tasks in the Real Time Specification for Java (RTSJ) [4], this can be done implicitly when `waitForNextPeriod()` is invoked. There is no need to change the application code. If no such method needs to be called by tasks, the scanning can be integrated into the scheduler. In the current version of the RTSJ, sporadic threads do not invoke such a method; their stack is however empty when they do not execute, which in turn makes root scanning trivial. The overhead for stack scanning of course has to be taken into account for calculating the WCET of tasks.

We assume that the garbage collection thread runs at the lowest priority in the system. On the one hand, a garbage collector usually has a long period (and deadline), compared to other real-time tasks. It follows from scheduling theory that it should have a low priority [11]. On the other hand, a real-time scheduler does not even need to be aware of the garbage collector to achieve full mutator utilization in such a setting.



(a) Solution for Periodic Tasks



(b) Generalized Solution

Figure 2.3: Visualization of the WCET for root scanning

2.3.2.2 Solution for Periodic Tasks

For periodic threads, the time between two releases is known and the time between two successive calls of `waitForNextPeriod()` is bounded. For this solution, the individual tasks push the references of the local root set onto the mark stack of the garbage collector if an appropriate flag is set. The garbage collector must wait until all tasks have acknowledged the scan before it can proceed. In the worst case, a task has become WAITING very early in its period when the garbage collector starts execution and becomes WAITING very late in its next period.

Let R_i be the worst case response time of a thread τ_i, Q_i its best case response time and T_i its period. The response time of a task is the time between the instant at which a thread becomes READY until it goes to the WAITING state again. $C_{stackscan}$ is the worst case time until all threads have scanned their local root set and the garbage collector may proceed. $C_{stackscan}$ can be computed as follows:

$$C_{stackscan} = \max_{i \geq 0}(T_i - Q_i + R_i) \tag{2.4}$$

Figure 2.3(a) visualizes the formula above. It shows that the worst case for two responses of a thread is $T - Q + R$. Consequently, this is the longest time the garbage collector must wait for this thread.

To avoid the computation of the best and worst case response times—especially the former is typically unknown—, this can be simplified to

$$C_{stackscan} = 2T_{max} \tag{2.5}$$

$C_{stackscan}$ has to be added to the response time of the garbage collection thread; the impact of this delay depends on the thread periods. If the minimal period of the garbage collector is far greater than the periods of the mutator threads, the relative impact is small. If there is some slack between the minimal and the actual garbage collection period, the effect can probably be hidden. If the maximum period of the mutator threads is relatively long, this may have a notable effect on the minimal garbage collection period.

2.3.2.3 Generalized Solution

The considerations for periodic tasks cannot be applied to sporadic tasks in the general case. For sporadic tasks, the minimum inter-arrival time is known, but usually not the maximum inter-arrival time. Therefore, the worst case time until the garbage collector may proceed is potentially unbounded.

The stack of a thread is only modified if the respective thread executes. Therefore, the garbage collector can reuse data from previous scans and only needs to wait for threads which may have executed since their last scan. These are—apart from the initialization and destruction of threads—the threads which are not in state WAITING.

We adapt the root scanning scheme such that threads save their stack on every call of `waitForNextPeriod()` to a *root array*. For WAITING threads, the content of the root array from their last scan is used by the garbage collector; for all other threads, the garbage collector waits until they have updated their root array. With this scheme, it is sufficient to take into account the worst case response time for the execution time of stack scanning.

$$C_{stackscan} = R_{max} \tag{2.6}$$

For schedulers which never execute lower priority threads if a higher priority thread is READY and which do not support blocking operations, the garbage collector will never encounter any threads which are not WAITING. Consequently, it is never necessary to wait for any thread to scan its stack and

$$C_{stackscan} = 0 \tag{2.7}$$

The downside of this approach is that a dedicated memory area is needed to save the roots of the individual threads. It is not possible anymore to let the mutator thread push its root set onto the mark stack. To avoid blocking in this scheme, it is necessary to use two memory areas for each thread to allow for double buffering. If the maximum number of roots is unknown, each of these areas occupies as much memory as the stack.

For this strategy, the overhead for completing the root scanning is larger on the garbage collectors side. This is due to the fact that the garbage collector itself has to push the references onto the mark stack. On the threads' side, the overhead is slightly smaller, because the content of the stack only has to be transferred to the root array, without performing any computations.

An advantage of this strategy is that $C_{stackscan}$ is considerably lower—the increased overhead for scanning is most likely far smaller than the time that is spent on waiting for the other threads. It is mandatory to use such a scheme for sporadic tasks; applying it to periodic tasks as well allows to trade off time to wait for a root scan with additional memory consumption.

Figure 2.3 compares the worst case scenarios of the solution for periodic tasks and the generalized solution. For the generalized solution, the threads save their stack in a root array at the end of each period. The garbage collector only has to wait for threads which have executed since their last scan. In Figure 2.3(b), thread 3 is BLOCKED when the garbage collector starts execution. Therefore, the garbage collector only has to wait for this thread. In the worst case, this waiting time equals the maximum response time.

2.3.2.4 Discussion

As pointed out in [12] and [15], the allowable allocation rates and the minimum garbage collection period are related. Increasing the period of the garbage collector effectively lowers the allowable allocation rates. The proposed solutions introduce a waiting time for the garbage collector and therefore may make it necessary to increase its period.

However, it is possible to mix root scanning strategies to find an optimal solution. For high frequency threads, jitter is usually very important, and the waiting time of the solution for periodic threads may be negligible. For medium frequency threads, the generalized solution with an impact in the order of one period may be a better trade-off. Low frequency threads are probably less sensitive to jitter and root scanning by the garbage collector may not hurt them. It is however not possible to propose a generic solution to this problem without knowledge about the properties of the whole system.

Table 2.3: Thread properties of the test program

Thread	Period	Deadline	Priority
τ_{hf}	100 μs	100 μs	6
τ_p	1 ms	1 ms	5
τ_c	10 ms	10 ms	4
τ_s	15 ms	15 ms	3
τ_{log}	1000 ms	100 ms	2
τ_{gc}	200 ms	200 ms	1

2.4 Evaluation

We used the Java Optimized Processor (JOP) [14] for the evaluation. The processor is implemented in an FPGA and runs at 100 MHz. The platform we used features 1 MB of SRAM with 15 ns access time. The garbage collection algorithm used by JOP [17] is an incremental garbage collector, with a snapshot-at-beginning write barrier [20]. It is based on the copying collector by Baker [3], but uses a forwarding pointer placed in an *object handle* to avoid the costly read barrier. While the absolute times reported in this section are of course platform dependent, the evaluated concepts are not specific to JOP and its garbage collector.

A special feature of our garbage collector is hardware support for interruptible copying [16]. A special hardware module translates accesses to objects and arrays that are copied, such that copying is transparent to the mutator threads. Accesses to fields that have not been copied yet are directed to the source location; accesses to already copied fields are directed to the destination location. Only single words need to be copied atomically. This allows even large arrays to be copied without introducing blocking times.

For jitter measurements, we used 6 different tasks. Deadline monotonic priority ordering is used to determine the tasks' priorities; unless otherwise stated, the deadline of a task equals its period. The task properties are described in the following and subsumed in Table 2.3. The figures presented in Table 2.4 were obtained by measuring the maximum release jitter of the highest priority thread during a run of more than one hour. For the measurements, we slightly modified the periods of the threads. We used prime numbers (e.g., 1009 μs instead of 1000 μs) to avoid a regular phasing of the threads, which could have led to too optimistic results.

The most important thread w. r. t. the measurements is the high-frequency task τ_{hf} with a period of 100 μs. It computes its own release jitter and does nothing else. This task has the highest priority of all tasks and all jitter figures in this section refer to the release jitter of this thread.

Two more threads, τ_p and τ_c, implement a producer/consumer pattern. τ_p produces one object every millisecond and τ_c consumes the available objects every 10 milliseconds. A simple list is used to pass the objects from τ_p to τ_c. These threads have the second- and third-highest priorities in the system. $\tau_{p'}$ is a slightly modified version of τ_p, which does not actually allocate an object. Instead, it emulates the blocking behavior of `new`, i.e., it contains a `synchronized` block which requires as long as the largest `synchronized` section in the implementation of new. This task is used to distinguish between the impact of synchronization of threads and the impact of garbage collection.

τ_s is a thread which occupies the stack such that it is not empty when `waitForNextPeriod()` is invoked. Consequently, a strategy as proposed by Schoeberl and Vitek in [17] cannot be used if this thread is part of the task set. The period of τ_s is 15 ms.

To record the measurements, we used a logging thread τ_{log} with a period of 1000 ms and a deadline of 100 ms. The garbage collection thread, τ_{gc}, has a period of 200 ms and is consequently the lowest-priority thread in the system.

We used various combinations of the tasks described above to evaluate the different root scanning strategies. The simplest task set comprises only the high-frequency task τ_{hf}. The jitter for this task indicates if the system is able to run this task at the requested frequency at all. The task sets $\{\tau_{hf}, \tau_{log}\}$ and $\{\tau_{hf}, \tau_s, \tau_{log}\}$ are used to measure the jitter due to task switches. As no garbage collection takes place and none of the tasks uses `synchronized` blocks, this is the only source of jitter for these two task sets. The jitter due to the combination of task switches and `synchronized` blocks are evaluated through the task sets $\{\tau_{hf}, \tau_{p'}, \tau_c, \tau_{log}\}$ and $\{\tau_{hf}, \tau_{p'}, \tau_c, \tau_s, \tau_{log}\}$. The other task sets present in Table 2.4 are used to evaluate the impact of garbage collection on τ_{hf}. τ_s has a relatively deep stack—task sets which contain τ_s therefore challenge the root scanning phase. τ_p and τ_c produce and consume objects and test the impact of the actual garbage collection on τ_{hf}.

The first row in Table 2.4 shows the trivial task set $\{\tau_{hf}\}$. The results for the task sets without garbage collection can be found in the next four rows. The bottom four rows present the results for the task sets with garbage collection.

We evaluated five different root scanning strategies. The strategy labeled *base* in Table 2.4 scans the stacks of all threads in one atomic step. The *single* strategy scans one stack at a time atomically. The strategy as proposed by Schoeberl and Vitek [17], which assumes that the thread stacks are empty at the time of root scanning, is labeled *empty*. The *scan* and *save* strategies implement the solution for periodic tasks and the generalized solution as described in Section 2.3.2. For the *scan* strategy, tasks push their local root set onto the mark stack at the end of their period. For the *save* strategy, tasks save their stack into root arrays, and the garbage collector pushes the references onto the mark stack.

2.4.1 Results

The results in Table 2.4 show that the *base*, *single* and *empty* strategies behave identical if no garbage collection takes place. This is no surprise, because the implementation of the scheduler is the same. The *scan* strategy behaves slightly worse, and the *save* strategy adds up to 27 μs of jitter (73 μs for *base*, *single* and *empty* compared to 100 μs for *save* in row 5 of Table 2.4). As we made only minimal changes to the scheduler, we had expected only a smaller deviation in the results. We observed that small changes in the application code sometimes have a considerable effect on the performance of the method cache [13] of JOP. However, further research will be necessary to find out if this is the actual source of the jitter increase. In the following, we do not attribute this jitter to garbage collection itself.

If garbage collection takes place, the *base* strategy performs worst w. r. t. the release jitter of τ_{hf}. The jitter introduced by this strategy is several times larger than for the other strategies.

Table 2.4: Jitter Measurements

Thread Set						Jitter (μs)				
τ_{hf}	τ_p	τ_c	τ_s	τ_{log}	τ_{gc}	base	single	empty	scan	save
✓						0	0	0	0	0
✓				✓		41	41	41	55	47
✓			✓	✓		68	68	68	66	57
✓	✓[a]	✓		✓		67	67	67	67	70
✓	✓[a]	✓	✓	✓		73	73	73	80	100
✓				✓	✓	321	110	57	56	56
✓			✓	✓	✓	488	180	–	70	65
✓	✓	✓		✓	✓	514	120	71	77	93
✓	✓	✓	✓	✓	✓	685	182	–	91	106

[a] $\tau_{p'}$, a slightly modified version of τ_p, was used for this measurement.

685 μs were measured as worst case jitter. The high jitter is caused by the atomic scan of all stacks; it increases with the number of threads.

The *single* strategy yields less jitter than *base*, but still more than the other strategies. The jitter for this strategy depends on the size of the largest thread stack; it is highest if τ_s is part of the thread set. Up to 182 μs jitter were observed during our experiments.

The *empty*, *scan* and *save* strategies induce a similar amount of jitter. However, *empty* cannot be applied if τ_s is part of the task set, because it cannot handle stacks which are not empty at the time of root scanning. A side effect of *scan* is that it implicitly extends the garbage collection period to 1000 ms. The reason for this is that the garbage collector cannot proceed until all other threads have run. τ_{gc} always has to wait until τ_{log} has executed once and therefore is locked to the frequency of the latter. The *save* strategy solves the problems of *empty* and *scan* at the expense of an increased memory consumption.

2.4.2 Discussion

Our goal was to minimize the impact of garbage collection on other threads. The figures in Table 2.4 show that a considerable amount of jitter is caused by scheduling and `synchronized` sections. Scheduling introduces jitter of 41 to 68 μs; scheduling and `synchronized` blocks together result in a jitter of 67 to 73 μs. For the *base* and *single* strategy, 50 to 600 μs are added to the jitter by the garbage collection thread. The *empty*, *scan* and *save* strategies perform considerably better in this regard.

The *scan* strategy adds 1 to 11 μs, when comparing the task sets with and without garbage collection. The *save* strategy adds 23 μs of jitter to the task set $\{\tau_{hf}, \tau_p, \tau_c, \tau_{log}\}$ and less than 10 μs to the other task sets. As blocking was eliminated from the root scanning phase, this can obviously not be the reason for the increased jitter. We assume three possible sources for the jitter increase: increased scheduling overhead, degraded performance of the instruction cache and imprecise measurements. On the one hand, longer measurements could have increased the measured jitter for some test cases. On the other hand, it is possible that the worst case behavior of some parts of the code only occurs if garbage collection actually takes place. Still, the jitter introduced by the two new root scanning strategies is considerably lower than the jitter introduced by scheduling and synchronization. Future work will have to show how far these two sources of jitter can be eliminated.

2.5 Conclusion and Outlook

We investigated the root scanning phase of garbage collection on a theoretical basis and could prove three important properties: First, that atomicity for stack scanning is only necessary w. r. t. the thread whose stack is scanned. Second, that atomicity is not required at all if mutator threads scan their own stack. And third, that a snapshot-at-beginning write barrier is sufficient to allow complete decoupling of local stack scans.

Furthermore, we provided two approaches how these theoretical properties can be utilized and showed the implications on the execution time of a garbage collector. The first approach can be applied only to periodic tasks and delays garbage collection by up to two times the largest task period. The second approach is more general and has a smaller impact on the execution time of the garbage collector, but has a higher memory overhead.

An evaluation of the two new approaches to root scanning confirmed the theoretical results. Jitter of high priority threads, which can be attributed to garbage collection, could be reduced considerably. The impact of the new root scanning strategies on the jitter due to scheduling and synchronization however still needs to be analyzed.

Future work will investigate if a tighter coupling of scheduling and root scanning is profitable. Merging the root arrays of the generalized solution with the memory areas for the thread contexts could lower the memory consumption without impairing the performance.

Exact stack scanning has not been handled in this paper. The proposed solutions lower the overhead for exact scanning, but tools to make use of this need to be developed.

Future work will also have to extend the proposed solutions to multi-processor systems. We are confident that the theoretical basis is applicable to such systems as well, but actual implementations may offer new obstacles as well as new opportunities.

2.6 Acknowledgement

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

Bibliography

[1] Joshua Auerbach, David F. Bacon, Florian Bömers, and Perry Cheng. Real-time music synthesis in Java using the Metronome garbage collector. In *International Computer*

Music Conference, pages 103–110, Copenhagen, Denmark, 2007. University of Michigan Library.

[2] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *30th Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 38(1), pages 285–298, New Orleans, LA, January 2003. ACM Press.

[3] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978. Also AI Laboratory Working Paper 139, 1977.

[4] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[5] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 33(5), pages 162–173, Montreal, Canada, June 1998. ACM Press.

[6] Edsgar W. Dijkstra, Leslie 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(11):965–975, November 1978.

[7] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 70–83, Portland, OR, January 1994. ACM Press.

[8] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *20th Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Charleston, SC, January 1993. ACM Press.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley Professional, Boston, Mass., 2005.

[10] Roger Henriksson. Scheduling real-time garbage collection. In *Proceedings of NWPER'94*, Lund, Sweden, 1994.

[11] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[12] Sven Gestegøard Robertz and Roger Henriksson. Time-triggered garbage collection: Robust and adaptive real-time GC scheduling for embedded systems. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM SIGPLAN Notices 38(7), pages 93–102, San Diego, CA, June 2003. ACM Press.

[13] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[14] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems.* PhD thesis, Vienna University of Technology, 2005.

[15] Martin Schoeberl. Real-time garbage collection for Java. In *9th International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 424–432, Gyeongju, Korea, April 2006. IEEE Press.

[16] Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *6th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '08, pages 77–84, Santa Clara, CA, September 2008. ACM Press.

[17] Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *5th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '07, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[18] Fridtjof Siebert. Constant-time root scanning for deterministic garbage collection. In *10th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, Genova, Italy, April 2001. Springer-Verlag.

[19] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[20] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990.

[21] Taiichi Yuasa. Return barrier. In *International Lisp Conference*, 2002.

3 Decoupled Root Scanning in Multi-Processor Systems[*]

Abstract

Garbage collection (GC) is a well known approach to simplify software development. Recently, it has started to gain acceptance also in the real-time community. Several hard real-time GC algorithms have been proposed for uniprocessors. However, the growing popularity of multi-processors for real-time systems entails that algorithms and techniques have to be developed that allow hard real-time GC on multi-processors as well.

We propose a novel root cache, which aggregates information of the processor-local root sets in multi-processor systems. It allows that the root scanning phase of the garbage collector is decoupled from the root scanning phase of working threads. Thread-local root scanning can be scheduled flexibly, without impeding the garbage collector. Additionally, the new cache lowers both the blocking time and the memory bandwidth consumption due to the root scanning phase of GC.

The proposed solution has been implemented for evaluation in a chip multi-processor system based on the Java Optimized Processor. We show how bounds on the garbage collector period can be extended to take into account the root cache. We also present experimental results, which highlight the advantages and limits of the proposed approach.

3.1 Introduction

GC reliefs the programmer from the task of manually managing memory and is part of a number of programming languages. However, simple garbage collectors cause pauses, which are hardly acceptable for interactive applications. This problem was an incentive for the development of early real-time garbage collectors, which focused on keeping interruptions small rather than achieving temporal predictability. Especially the use of Java in real-time systems has drawn new attention to this area, now also focusing on hard real-time systems, where temporal predictability is of utmost importance.

The authors of the RTSJ [4] were not convinced of the GC techniques available at the time—the specification efforts started in the late 1990s—and introduced a special programming model for memory management to Java, through the use of scoped memory. The programming model with scoped memory is however unusual to most programmers and forces the Java virtual machine (JVM) to check all assignments to references. Failing to adhere to the programming model will trigger run-time exceptions—arguably a different level of safety than most Java programmers would expect. Therefore, hard real-time GC—although not being strictly necessary—is still considered a valuable goal for designers of real-time JVMs.

[*]This chapter has been previously published as: Wolfgang Puffitsch. Decoupled root scanning in multi-processor systems. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 91–98, Atlanta, Georgia, USA, October 2008. ACM Press.

While there are hard real-time GC algorithms for uniprocessors [20, 15, 2], the known algorithms for multi-processor GC rather focus on soft real-time systems [13, 6], if considering this topic at all.

In this paper we present a novel cache, which is used for caching the references each processor in a chip multi-processor (CMP) system accesses; a garbage collector can use the information in this cache to construct its root set without interrupting the working threads. Therefore, the cache allows the temporal decoupling of the garbage collector and the working threads in a CMP system. It also lowers the number of memory accesses for each processor, removing some pressure on the memory bandwidth, a typical bottleneck in CMP systems.

This paper is organized as follows: the rest of this section highlights the challenges of real-time GC on multi-processor systems. Section 3.2 describes the proposed cache and its implications; it is evaluated in Section 3.3. Related work is covered in Section 3.4, and Section 3.5 concludes the paper.

3.1.1 Garbage Collection Algorithms

Traditionally, there are three approaches to GC: reference counting, mark-sweep algorithms and copying algorithms [10].

Reference counting counts the number of references to each object and frees them, when this count drops to zero. While this looks appealing on the first sight, the approach suffers from problems with cyclic references. The solutions to circumvent these problems rely on a certain programming model, require a considerable amount of overhead, or need a mark-sweep phase to reclaim cyclic structures. As we are not convinced that these work-arounds are feasible for real-time systems, reference counting will not be discussed further in this paper.

Mark-sweep and copying algorithms differ in a number of details, but share the same concept: First, a set of *roots* is determined, which contains all objects that are directly accessible. Starting from this root set, the object graph is traversed to find all reachable objects. The objects not discovered in this second phase are obviously garbage and can be recycled. Mark-sweep garbage collectors suffer from heap fragmentation, but can be extended with a compaction phase to circumvent this problem; they are then usually referred to as mark-compact garbage collectors.

In order to lower the blocking time of GC, incremental algorithms have been developed, which allow interleaved execution of working threads and the garbage collector. The working threads have to execute *read barriers* [3] or *write barriers* [24] when accessing memory to ensure the correctness of GC.

Figure 3.1 shows how the execution of an incremental mark-sweep or copying garbage collector may look like in a multi-processor system. One CPU is responsible for GC, while three other CPUs execute working threads (we assume one thread per CPU for simplicity here). At point A, the garbage collector initiates a new GC cycle. This has to be atomic with regard to interactions of the working threads with the memory management, namely the creation of new objects and the execution of barriers. The garbage collector then (at *B1*) scans its own stack for root references and waits until all other threads have done so as well; static variables are scanned afterwards (at *B2*). While the scanning of thread stacks has to be atomic per thread, the scanning of static variables does not necessarily have to be so. At *C*, the garbage collector starts to trace the object graph and to copy the used objects or sweep the unused objects, depending

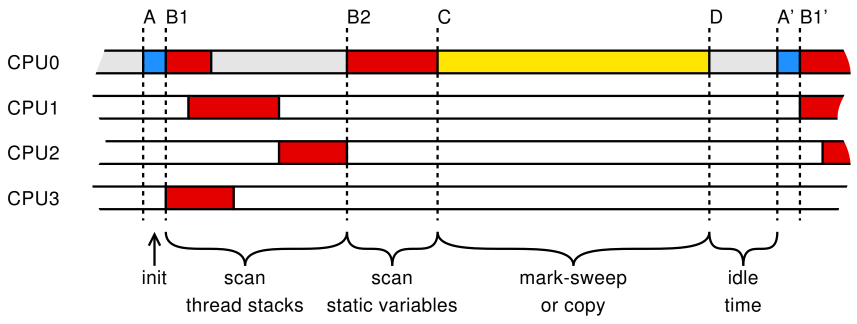


Figure 3.1: Multi-processor garbage collection

on the algorithm. When the GC is finished at *D*, the GC thread goes idle until the start of the next GC cycle at *A'*.

When using an incremental garbage collector, the working threads can continue with their work throughout all phases of GC. They only need to be halted while their local root set is scanned.

A crucial point in Figure 3.1 is that GC may not proceed until all threads have scanned their stacks. One could interrupt all threads and force them to scan their stacks; obviously this will result in a low waiting time, but means that even highest-priority threads must be interruptible. One could also signal them that they should scan their stack and wait until they have done so. From a thread's perspective, stack scanning ideally takes place at the end of its period, because its stack is usually low then and virtually no overhead has to be put into checking when to do a stack scan; such a strategy however will result in a long waiting time for the garbage collector. The implications of letting threads scan their local roots sets at the end of their periods are analyzed in [14].

3.1.2 Real-Time Garbage Collection

Real-time GC has to ensure three fundamental properties: bounded blocking times, acceptable utilization of working threads and bounded memory consumption. While the last requirement does not refer to timing in the first place, a thread that runs out of memory cannot provide its result on time—it will provide no result at all. Blocking times from GC have several reasons:

1. Retrieval of the root set
2. Copying of objects
3. Scanning of objects for graph traversal
4. Access to data structures which are shared between the working threads and the garbage collector

The access to shared data structures such as the mark stack must inevitably be synchronized to ensure these structures are not corrupted. However, the access to them appears only at certain well defined instructions (from the working threads' perspective) and the critical section is typically only a few instructions long. This blocking has to be taken into account for instructions that allocate objects (e. g., `new` in the JVM) and instructions that may execute a read or write barrier.

The issue of object scanning is a problem only for certain algorithms. Using an appropriate algorithm, the blocking time can be reduced to the access to shared data structures.

Copying of objects is an issue for copying and mark-compact garbage collectors, because copying must be atomic to ensure the consistency of the involved data. Traditional mark-sweep algorithms are not considered feasible for hard real-time systems (where memory consumption must be bounded) due to their problems with fragmentation. A mark-sweep garbage collector has been proposed in [21] to eliminate copying and external fragmentation at the expense of internal fragmentation. In [18], hardware support for non-blocking copying of objects is proposed. Extending this solution to multi-processors however remains future work.

The retrieval of the root set is not considered for the blocking time of a garbage collector in many papers. Incremental root scanning algorithms have been proposed to lower the blocking time for uniprocessor systems [5, 22, 25], but it is not clear if these can be extended to multiprocessors without introducing a considerable overhead to ensure the consistency of the root set. They are also more costly than traditional root scanning schemes; in [22], more than one million references have to be saved each second to "root arrays" and a runtime overhead of 11.8% is estimated. Furthermore, root scanning usually induces memory accesses proportional to the size of the stack, rather than to the overall memory consumption. A computationally expensive thread that rarely accesses memory can therefore cause a considerable amount of traffic, unnecessarily increasing the pressure on the memory bandwidth in a CMP system.

3.2 Root Cache

The root set for a garbage collector consists of any references that are "always" accessible (any references in `static` variables in Java) and thread-local references. The latter comprise any references in local variables, the operand stack and CPU registers. In the JVM, all this data is part of the run-time stack. As a running thread constantly modifies this data, it is impossible to get a consistent view of it without cooperation; usually, execution is stopped until all relevant data is saved. Registers of a processor may not be accessible to other processors in a CMP system, so a processor has to scan them for references by itself. The view of the root set also has to be consistent such that no reachable object appears unreachable to the garbage collector; establishing this consistency makes it necessary to use some form of synchronization.

The idea behind the proposed cache is that in Java, a processor cannot "invent" a reference. As there is no pointer arithmetic in Java, references can only point to previously allocated objects. There are only two possibilities how a reference can enter a processor: either it is read from main memory or it is created by the memory management. It is evident that in the first case the memory access is observable from outside the processor. In the latter case, the interaction between a working thread and the memory management ensures that the reference cannot remain undetected.

This fact is utilized by snooping memory accesses of running processors and marking the detected references as live in a cache. In a second step, a processor scans the thread-local data,

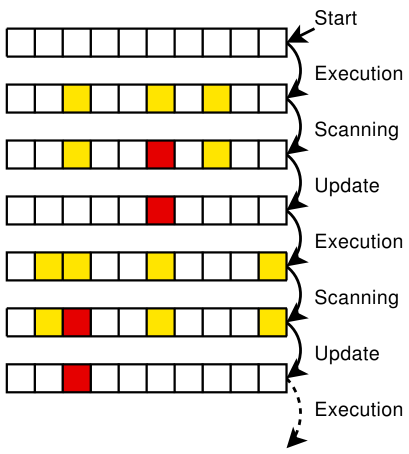


Figure 3.2: Algorithm for a single processor

and marks the references it finds in a different color. After it has marked the cache entries, the references marked only during snooping can be marked as unused again, because they are known to be dead. The *cache narrowing* by the processor ensures that the amount of marked references in the cache is bounded.

In the following, we will use the colors *white*, *yellow* and *red* for marking. These colors were chosen to avoid confusion with Dijkstra's tricolor abstraction [8], which uses white, grey and black for marking. Gray scale reproductions of Figures 3.2 and 3.3 will show yellow as light gray and red as dark gray.

Figure 3.2 shows the algorithm for a single processor. Upon start-up, all references are marked as unused (white). The processor then starts execution, and references it accesses are marked yellow. When the processor scans its stack, it marks the references it finds red, usually leaving some entries yellow. These entries are removed in the next step, where all yellow entries are reset to white. When the processor starts execution again, red entries become yellow again and, as in the previous execution phase, accessed references are marked yellow.

Note that a color always corresponds to the same bit pattern in the cache. Changing the interpretation of patterns would have resulted in a considerably larger hardware implementation than changing the values in the cache.

The algorithm ensures that a superset of all references which are contained in a root set is marked yellow or red. During execution, a reference is marked when it is accessed—as a reference has to be accessed to become live, all references which are live at some point of execution are marked yellow after that point. When the processor scans the thread-local data, it marks all live references red; after the cache narrowing, all yellow references are known to be dead and can be marked as unused again. When execution continues, red entries are reset to yellow, so they are not erroneously identified as live when the cache entries are narrowed the next time. Updating the references from red to yellow and from yellow to white is done by the hardware.

In a CMP system, the information of the caches for each processor has to be combined somehow. It also has to be ensured that the gathered information is consistent such that no

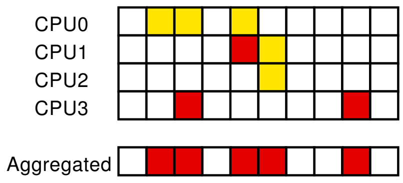


Figure 3.3: Snapshot of execution with four CPUs

reference in the root set can remain undetected. This is achieved with a memory in which the content of the per-processor caches is aggregated. In this memory, a reference is marked if it is marked yellow or red in any per-processor cache. The aggregate state computation is essentially a logical OR of the entries in the processor-local caches, which can be implemented efficiently in hardware.

In Figure 3.3, an intermediate state is shown; while the individual processors are in different phases of execution, the aggregate state reflects all references which may be part of a local root set. The aggregate state corresponds to the union of all processor-local root sets and can be used by the garbage collector to construct the global root set at any time.

3.2.1 Scheduling of Local Root Scans

There is no need to synchronize the root scanning of individual processors to achieve a consistent view of their root sets. Different strategies can be used for different threads, without changing the garbage collector. The most appropriate strategy can therefore be chosen for each thread.

A simple strategy is to use a timer interrupt to trigger a cache narrowing from time to time. This however introduces some jitter, which may not be tolerable for some applications.

Another solution is to wait until a thread waits for its next period, as proposed in [19]. As the stack is almost empty then and no overhead has to be put into checking when a stack scan should be done, this is a very convenient strategy from a threads perspective. For traditional root scanning schemes, this introduces a considerable waiting time to the garbage collector (see Figure 3.1). The proposed cache however allows one to use such a strategy without introducing waiting times.

A third solution is to scan the thread-local data upon thread switches. The state of the old thread has to be saved and the state of the new thread has to be read from main memory, so the overhead for narrowing the cache in the course of doing so can be kept small for some processors. Again, such a strategy would have caused considerable waiting times in the garbage collector for traditional root scanning schemes, but does not do so for the proposed cache.

3.2.2 Garbage Collection Period

The temporal decoupling of local and global root scanning comes at the cost of longer life times of references. A reference which is not contained in a thread's stack may still be marked in the root cache. In this section, we will show how this influences the requirements for the GC period of a copying garbage collector. The formulas can easily be extended to a mark-compact garbage collector.

An upper bound for the memory consumption is provided by the worst case memory consumption of all processors before they scan their stacks plus the amount of memory they allocate during the WCET of two GC cycles. It is necessary to take into account the GC cycle twice, because a GC cycle which started just before stack scanning cannot free the respective memory. This bound is therefore dependent on the allocation rate of individual threads and the GC period T_{GC}.

In [17], it is shown that for a copying garbage collector the following inequation must hold for situations where data is generated by one thread ("producer") and consumed by a different thread ("consumer"):

$$T_{GC} \leq \frac{H_{CC} - 2\sum_{i=1}^{n} a_i l_i - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{3.1}$$

H_{CC} is the overall heap size, T_i is the period of a thread τ_i, a_i the amount of memory allocated during each period by τ_i and l_i the "life time factor". It takes into account that data produced by one thread is consumed by another thread. It is defined as[1]

$$l_i = \begin{cases} 1 & \text{for normal threads } \tau_i \\ \left\lceil \frac{2T_c}{T_i} \right\rceil & \text{for producer } \tau_i \text{ and consumer } \tau_c \end{cases} \tag{3.2}$$

If cache narrowings take place at the end of a task's period, the lifetime of objects is not extended artificially beyond a period. Therefore, the assumptions of the equations above are fulfilled and these formulas can be applied to the proposed GC algorithm.

If cache narrowings are triggered by an interrupt, the definition of l_i has to be extended such that the time between the narrowings of the cache is taken into account. With $T_{u.i}$ as time between these cache narrowings w. r. t. a thread τ_i and C_u as WCET of such a narrowing, it can be redefined as

$$l_i' = \begin{cases} 1 + \left\lceil \frac{T_{u.i}+C_u}{T_i} \right\rceil & \text{for normal threads } \tau_i \\ \left\lceil \frac{2T_c+T_{u.c}+C_u}{T_i} \right\rceil & \text{for producer } \tau_i \text{ and consumer } \tau_c \end{cases} \tag{3.3}$$

By replacing l_i in Equation 3.1 with l_i', an upper bound for the GC period for a copying garbage collector can be computed.

Due to the limited size of the caches, we do not only have to meet restrictions on the available heap memory, but also on the number of references. A copying garbage collector

[1]This is reformulation of the original definition

$$l_i = \begin{cases} 1 & \text{for normal threads } \tau_i \\ 2\left\lceil \frac{T_c}{T_i} \right\rceil & \text{for producer } \tau_i \text{ and consumer } \tau_c \end{cases}$$

which provides tighter bounds but is still safe.

using the presented root scanning scheme also has to fulfill the following inequation:

$$T_{GC} \leq \frac{\widehat{H}_{CC} - 2\sum_{i=1}^{n} \widehat{a}_i l_i' - 2\sum_{i=1}^{n} \widehat{a}_i}{2\sum_{i=1}^{n} \frac{\widehat{a}_i}{T_i}} \tag{3.4}$$

where $\widehat{H}_{CC}$ means the maximum number of references and $\widehat{a}_i$ is the number of objects a thread τ_i allocates each period.

3.2.3 Design Considerations

It would be resource intensive and inefficient if the references had to be stored in the cache along with the colors. Therefore, it is necessary that there is a simple mapping between references and positions in the cache. This is the case if the object layout is regular, e. g., if a dedicated area for object handles is used (like in JOP [16]) or if fixed size blocks for objects are used (such as in the JamaicaVM [21]). By making use of this mapping, two bits are needed in the cache for each reference to store the colors.

It also would be very resource intensive to update the coloring of the references (red → yellow, yellow → white) and the aggregate state in a single cycle. This task hence has to be spread across several cycles. Moreover, these updates should not delay the processors' execution to keep the jitter as low as possible. The solution is time-multiplexed access to the caches. In the first cycle, the processor may access the values; in the second cycle, the update "demon" updates the coloring and computes the aggregate state. This entails that access to the caches has to be at least two times faster than access to main memory in order not to affect performance. The "demon" runs continuously and does not need to be triggered in any way. The time until a color change in a processor local cache is reflected in the aggregate state is therefore dependent on the number of entries in the caches, the number of entries updated in one step of the "demon" and the time for one step.

Splitting updates into several cycles is not problematic with regard to references which move from one processor to the other, because the state for a reference is always updated within a single cycle. What *could* pose problems is the fact that the aggregate state may not reflect a recently created object. This may however be circumvented by either allocating new objects in an area which is not garbage collected in this GC cycle or by coloring objects upon allocation. After the start of a GC cycle, the garbage collector has to wait until the update "demon" has computed the aggregate state for all references, i. e., until all objects created before the start of the GC cycle are reflected in the aggregate state. Otherwise, objects that were created just before the start of the GC cycle could be garbage collected even though they are live.

A different issue with the split update is that processors have to wait after scanning their stack, until the update has unmarked all yellow entries. If it does not wait long enough, the update will only be partial, and stale objects may be considered live. It is however safe to use this data, as no live root object is marked as unused. If it can be guaranteed that the time between successful updates is bounded, the amount of stale objects is bounded as well.

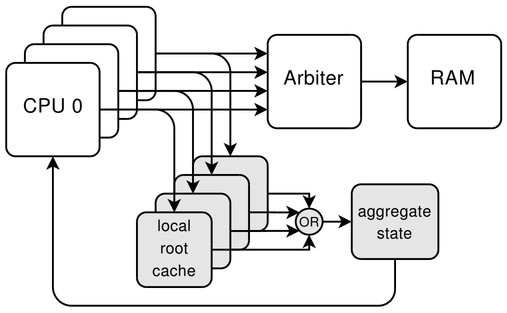


Figure 3.4: Block diagram

3.3 Evaluation

We implemented the root caches in a CMP system based on JOP [16, 11, 12]. While the principal algorithm is not tied to any specific processor, the results presented in this paper were obtained with this implementation and will be different in some places for other processors.

In our implementation, a single CPU takes care of garbage collection. While it would be possible to distribute the work load to more than one processor, we do not expect any gains from doing so, due to the synchronization overhead and the fact that access to shared memory rather than computation power appears to be the main limiting factor for GC in a CMP system.

Apart from the CPU dedicated to GC, all other CPUs are free to execute real-time threads. Figure 3.4 shows a block diagram of the individual components. The components we added to the standard configuration are shaded. While each CPU updates its state privately, one CPU can access the aggregate state to perform garbage collection. Accessing the per-processor caches and the main memory can be done in parallel.

We used a CMP based on JOP with 8 cores for evaluation. The CMP comprises 8 JOP cores, an arbiter for access to the shared memory, and synchronization and I/O modules. The board we used features an Altera Cyclone-II field programmable gate array (FPGA) and 512 KB SRAM with a 16-bit interface. The processor is clocked at 100 MHz; 32-bit accesses to the SRAM take 6 cycles. Each core includes 1 KB of on-chip memory for the method cache (a special instruction cache) and 1 KB for the stack cache. The root cache for each processor is also 1 KB in size, allowing for 4096 objects to be under the control of the garbage collector.

The GC algorithm used by JOP [19] is an incremental garbage collector, with a snapshot-at-beginning write barrier [24]. It is based on the copying collector by Baker [3], but uses a forwarding pointer placed in *object handles* to avoid the costly read barrier.

Memory arbitration is done by a time-sliced arbiter, which makes it possible to reason about the worst case latency of memory accesses. We chose a slot size of 6 cycles, which equals the time for one 32-bit memory access. As copying of objects must be atomic, the worst case memory access latency is however not only dependent on the slot size. Rather, the worst case memory latency for a CPU depends on the length of the atomic operations of all other CPUs. For the CPU that does GC, the longest atomic operation is the copying of the largest object in the system. For all other CPUs, it is the access to arrays, which includes three back-to-back memory accesses. Hardware support to eliminate the blocking due to copying is proposed in [18]. Future work will have to extend this solution to multi-processors.

3.3.1 Resource Consumption

Two bits are needed for each reference to realize the three possible colors. The number of objects depends on the application and is constrained by the overall heap size. JOP uses a dedicated area for object headers, a forwarding pointer and other GC information; each of these object handles is 8 words in size. Therefore, even if only objects that do not carry any data are created, no more than $\frac{heapSize}{4*8}$ objects can exist. This number will typically be lower, because objects consume memory as well, decreasing the maximum possible number of objects. Note that not all of the main memory is heap memory, because it contains also the bytecodes and class descriptions. Even very small Java programs occupy several dozen KB of main memory with such data.

With 1 MB of heap memory, 32768 objects can exist at most; furthermore, the payload of objects has to be taken into account, and—for a copying collector—that only one semi-space can contain live objects. An average object size of 33 Bytes is reported in [9], so a reasonable estimate for the number of handles would be $\frac{heapSize}{4*(8+8.25*2)}$, yielding 10700 handles for 1 MB of heap memory.

A safe limit can however only be determined by proper memory analysis. Such an analysis is necessary in any case to confirm that the heap is large enough to satisfy all allocation requests. Determining the maximum number of objects is even simpler than this analysis—using an object size of 1 for all objects, the "memory consumption" equals the number of objects.

As a concrete example, consider the system we used for evaluation, with eight CPUs and 512 KB main memory. Allowing 4096 handles, this yields a memory consumption of $\frac{2*4096}{1024*8}$ =1 KB for each per-CPU cache. The aggregate state cache is 0.5 KB in size. For the whole system, 8.5 KB of on-chip memory are required for the root caches, which is an overhead of 1.66% when comparing it to the size of the main memory. In the JOP-based CMP system, other on-chip memories (method cache, stack cache and microcode ROM) occupy 4.625 KB per processor. The overall memory consumption is 45.5 KB, of which the root caches occupy 18.7%.

The system in consideration is balanced with regard to the number of handles and the available heap memory for average object sizes of 8 to 9 words for medium sized applications (about 100 KB of non-heap data). These figures are consistent with the average object sizes reported in [9].

For the configuration presented above, 32 entries can be updated in one cycle without inferring additional memory blocks (one cache consists of two 4 Kbit memory blocks, which together allow 64 bit access in one cycle). An update consequently takes 256 cycles, or 2.56 μs at 100 MHz. This time has to be taken into account for the blocking time of a processor for root scanning.

The whole CMP system occupies 26546 logic cells (LCs). Each processor core consumes about 2600 LCs; arbitration, synchronization and I/O modules consume about 3150 LCs. The

Table 3.1: Measured execution times

	BCET	WCET
Traditional Root Scan	23 μs	87 μs
Cache Narrowing	13 μs	37 μs

update logic occupies 2588 LCs, which is 9.7% of the total LC count. This number could have been lowered by updating fewer entries in one cycle at the expense of a longer update period.

3.3.2 Execution Time

In the presented scheme, every processor only has to access the cache memory during stack scanning, which is not shared with other processors. The processors do not have to wait for their memory slot and stack scanning therefore has the same WCET regardless of the number of processors (i. e., it is as fast as on a single processor system). The presented scheme however infers a delay for waiting until the update is finished; the length of the delay depends on the configuration, but a value of 2.56 μs (see Section 3.3.1) is reasonable. This overhead is an order of magnitude smaller than the WCET for stack scanning with time-multiplexed access to the main memory. A system consisting of 8 CPUs running at 100 MHz with 256 stack entries and a memory with 6 cycles access time needs at least 122.88 μs until all processors have written the contents of their stacks to main memory. Manual analysis of the bytecodes for cache narrowing resulted in a WCET of around 39 μs on JOP. Note that the figures presented in Section 3.3.3 are lower, because 64 words of the stack cache are not used for the run-time stack.

3.3.3 Experimental Results

For our measurements, we used the already mentioned CMP based on JOP, with 8 cores and 512 KB main memory. While one core was responsible for GC, the other seven cores executed the working threads.

Table 3.1 compares measurements for the best case execution time (BCET) and WCET for traditional root scanning and the proposed solution. These were obtained by triggering a stack scan at a period of 1 ms, while a task that occupied the stack was executed at various periods. The traditional root scanning was emulated by writing the stack contents to a dummy memory location. While the execution time of traditional root scanning depends on the access to main memory, the proposed scheme accesses only local memory and does not need to rival with other processors for this resource. The WCET for this task is therefore less than half for the proposed solution, compared to traditional stack scanning. The difference might be even bigger for arbitration schemes where some processors are preferred over others with regard to memory requests. While the preferred processors may have a lower WCET for cache narrowings, the memory access latencies (and hence the WCET) for other processors are inevitably increased.

Tables 3.2, 3.3 and 3.4 show our experiments with different cache narrowing policies. In the experiments for Tables 3.2 and 3.3, the cache narrowing is triggered periodically, with periods

Table 3.2: Measurements with narrowing period of 1 ms

Test	Min. Period (ms)	Allocations (Objects/s)	Jitter (μs)
`ObjectTest`	0.25	26977	90
`ListTest`	27.5	25416	52
`TreeTest`	27.5	23954	51
`ArrayTest`	1.2	5654	206

of 1 and 10 ms, respectively. For Table 3.4, the root cache is narrowed at the end of each period of a working thread.

Throughout all tests, the garbage collector is executed in a loop with a gap of 1 ms between iterations to allow for logging to be done. The working threads allocate various data structures, to test different aspects of the garbage collector. They also check the integrity of the allocated data to provide evidence for the correctness of the garbage collector.

The minimum periods were determined in two ways: on the one hand we measured the WCET, on the other hand we lowered the task periods every 100 seconds, until the system ran out of memory. The lowest period at which the system operated correctly, without missing any deadlines, is shown in the tables. The allocation rate shown is the rate reported by the test for this minimum period. The jitter displayed is the maximum release jitter we measured throughout the full run of a test.

`ObjectTest` is a "best case" example: The working threads allocate one simple object in each period. The objects do not need to be scanned and the tracing of the object graph is effectively a non-issue. `ObjectTest` is the only test which reaches the empirically determined WCET. For `ListTest`, the working threads allocate a linked list with 100 elements each period. For `TreeTest`, a tree structure comprising 94 elements is allocated. Obviously, the object graph for these structures is more complex than for `ObjectTest`. `ArrayTest` allocates integer arrays with 256 elements, resulting in a memory consumption of 1 KB for each array. While the object graph for this test is simple, it challenges the garbage collector through the size of the objects which must be copied.

`ObjectTest`, `ListTest` and `TreeTest` achieve similar allocation rates. `ObjectTest` is however bounded by its WCET, while `ListTest` and `TreeTest` run out of memory at higher execution frequencies. Further tests showed that the garbage collector can keep up with an allocation rate of 34884 objects per second and a minimal period of 160 μs for `ObjectTest` if the WCET is ignored. `ArrayTest` has a low allocation rate in terms of objects. This is not surprising, taking into account that it allocates large objects and thus the limiting factor is its allocation rate in terms of bytes rather than in terms of objects.

The release jitter for `ListTest` and `TreeTest` is lower than for `ObjectTest`. The sources for the release jitter are however the same. Consequently, we have to assume that the measurement is too optimistic for `ListTest` and `TreeTest`. The release jitter `ArrayTest` is considerably worse, compared to the other tests. This is due to the atomic copying of objects and arrays, which causes a release jitter proportional to the size of the largest object or array.

Table 3.3: Measurements with narrowing period of 10 ms

Test	Min. Period (ms)	Allocations (Objects/s)	Jitter (μs)
`ObjectTest`	0.25	26428	79
`ListTest`	37.5	18665	40
`TreeTest`	30.0	21970	49
`ArrayTest`	1.7	3870	259

Table 3.4: Measurements with narrowing at end of task period

Test	Min. Period (ms)	Allocations (Objects/s)	Jitter (μs)
`ObjectTest`	0.25	27325	26
`ListTest`	30.0	23365	26
`TreeTest`	30.0	21952	26
`ArrayTest`	1.1	6229	62

Table 3.3 shows the effect of a longer narrowing period: the achievable allocation rate is lowered, while on average, the jitter remains the same. The lower allocation rate can be explained by the extended life time of objects in the root cache. As more objects are live at the same time, the system runs out of free memory earlier. The effect is not the same for all tests: while the minimum period for `ListTest` and `ArrayTest` is considerably increased, there is only a small increase for `TreeTest`. `ObjectTest` again reaches its WCET. An advantage of the longer narrowing period is however that the relative overhead for narrowing the cache is decreased.

Narrowing the root cache at the end of the task period yields similar minimal periods as narrowing the cache at a period of 1 ms. While for `ListTest` and `TreeTest`, the minimal period is only slightly higher, it is slightly lower for `ArrayTest`. For `ObjectTest` the WCET is reached again. The advantage of this policy is however, that the release jitter is lowered considerably. This is due to the fact that the thread stacks are almost empty at the end of a period. The measurement for `ArrayTest` seems to be too optimistic; we assume that this is caused by advantageous thread phasings. While the jitter may seem to be negligible when comparing it to the task periods of `ListTest` and `TreeTest`, it is an issue for a task with sub-millisecond periods such as `ObjectTest`. The relative overhead for narrowing the cache increases for smaller periods with this strategy.

The figures in Tables 3.2, 3.3 and 3.4 indicate that there is a trade-off between the needed task period/allocation rate and the allowable jitter and overhead. This trade-off is influenced by the allocation behavior of a task, which therefore must be considered to find an optimal solution.

3.4 Related Work

While there are numerous papers on GC, only few take blocking times due to root scanning into consideration [22, 25, 2]. The work we are aware of in this area covers only uniprocessors. A more recent paper [1] identifies problems with the consistency of the root set in multi-processor systems, and suggests a special write barrier to overcome these. It still requires that the stack of a single thread is saved to main memory atomically.

The presented algorithm shares some concepts with reference counting techniques like the Deutsch-Bobrow algorithm [7] and one-bit reference counts as proposed in [23]. These algorithms use an approximation of the reference count, which is made precise on certain occasions. While the Deutsch-Bobrow however caches which references are probably dead in a "zero count table", the algorithm presented in this paper caches which references are probably live. One-bit reference counts are also used to remember the live references, but a full mark/sweep run is needed to overcome the problems of reference counting. A fundamental difference between reference counting approaches and the proposed root cache is however that the latter only "counts" the roots of the object graph, while the former apply reference counting to all objects.

3.5 Conclusion and Outlook

We presented a novel root cache, which allows that thread-local root scanning can be decoupled from the root scanning phase of a garbage collector in a CMP system. This enables flexible scheduling of thread-local stack scans without impairing the garbage collector. The proposed cache also lowers the WCET of stack scans compared to traditional root scanning techniques. Thread-local root scanning does not occupy memory bandwidth, which is a typical bottleneck in CMP systems.

The evaluation of the root cache demonstrated that the jitter caused by GC is reasonably low to allow even sub-millisecond task periods if only objects and small arrays are used. Further research is however necessary to avoid the atomic copying of larger arrays, which causes considerable jitter.

Acknowledgement

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

Bibliography

[1] Joshua Auerbach, David F. Bacon, Florian Bömers, and Perry Cheng. Real-time music synthesis in Java using the Metronome garbage collector. In *International Computer Music Conference*, pages 103–110, Copenhagen, Denmark, 2007. University of Michigan Library.

[2] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *30th Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 38(1), pages 285–298, New Orleans, LA, January 2003. ACM Press.

[3] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978. Also AI Laboratory Working Paper 139, 1977.

[4] Greg Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[5] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 33(5), pages 162–173, Montreal, Canada, June 1998. ACM Press.

[6] Cliff Click, Gil Tene, and Michael Wolf. The Pauseless GC algorithm. In Michael Hind and Jan Vitek, editors, *1st ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 46–56, Chicago, IL, June 2005. ACM Press.

[7] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[8] Edsgar W. Dijkstra, Leslie 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(11):965–975, November 1978.

[9] M. Teresa Higuera, Valerie Issarny, Michel Banatre, Gilbert Cabillic, Jean-Philippe Lesot, and Frederic Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[10] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[11] Christof Pitter and Martin Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.

[12] Christof Pitter and Martin Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the IEEE Third Symposium on Industrial Embedded Systems (SIES 2008)*, Montpellier, France, June 2008.

[13] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. Stopless: A real-time garbage collector for multiprocessors. In Greg Morrisett and Mooly Sagiv, editors, *6th International Symposium on Memory Management*, pages 159–172, Montréal, Canada, October 2007. ACM Press.

[14] Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *6th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '08, pages 68–76, Santa Clara, CA, September 2008. ACM Press.

[15] Sven Gestegøard Robertz and Roger Henriksson. Time-triggered garbage collection: Robust and adaptive real-time GC scheduling for embedded systems. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM SIGPLAN Notices 38(7), pages 93–102, San Diego, CA, June 2003. ACM Press.

[16] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[17] Martin Schoeberl. Real-time garbage collection for Java. In *9th International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 424–432, Gyeongju, Korea, April 2006. IEEE Press.

[18] Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *6th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '08, pages 77–84, Santa Clara, CA, September 2008. ACM Press.

[19] Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *5th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '07, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[20] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *6th International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, pages 96–102, Hong Kong, 1999. IEEE Press, IEEE Computer Society Press.

[21] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 9–17, San Jose, CA, November 2000. ACM Press.

[22] Fridtjof Siebert. Constant-time root scanning for deterministic garbage collection. In *10th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, Genova, Italy, April 2001. Springer-Verlag.

[23] David S. Wise and Daniel P. Friedman. The one-bit reference count. *BIT*, 17(3):351–359, 1977.

[24] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990.

[25] Taiichi Yuasa. Return barrier. In *International Lisp Conference*, 2002.

4 Data Caching, Garbage Collection, and the Java Memory Model*

Abstract

Multiprocessors often feature weaker memory models than sequential consistency. Relaxed memory models can be implemented more efficiently and allow more optimizations. The Java memory model is a relaxed memory model that is a natural choice for a Java processor. In this paper, we show that a data cache that obeys the restrictions of the Java memory model can be implemented efficiently. The proposed implementation requires only local actions to ensure data consistency and is therefore timing analysis friendly.
Furthermore, we investigate the impact of the proposed data cache design on the correctness of real-time garbage collection. Relaxed memory models must to be taken into account when initializing and moving objects during heap compaction. Also, write barriers and the start of a garbage collection cycle require careful design under the Java memory model.

4.1 Introduction

Sequential consistency constrains the possible optimizations and requires complex cache coherence protocols. Therefore, multiprocessors often features weaker memory models. The Java memory model (JMM) [18, 12] is a memory model that is relatively loose and can be implemented efficiently. Many garbage collection (GC) algorithms are formulated with uniprocessors in mind and do not take into account weak memory models. Some multiprocessor GC algorithms do take into account the requirements of the underlying memory model (e.g., [17, 3, 5]), but these papers provide ad-hoc solutions, tailored to the needs of the algorithm and the underlying hardware.

Common processors implement their own memory model, which may provide stronger guarantees than the JMM. For a Java processor, there is no apparent need to provide a stronger memory model than the JMM. However, the JMM relies on programs being properly synchronized for inter-thread communication; GC acts on a different level, which the application is not aware of. Therefore we investigate the implications of the JMM on GC algorithms.

Furthermore, we propose an implementation of the JMM in the context of the Java Optimized Processor (JOP) [23, 19]. We show that the JMM can be implemented efficiently while retaining temporal predictability for data caches. This implementation is therefore a good match for hard real-time systems, where worst-case execution time (WCET) analysis is required. The proposed data cache is a write-through cache; it is invalidated upon `monitorenter` and reads from volatile variables.

*This chapter has been previously published as: Wolfgang Puffitsch. Data caching, garbage collection, and the Java memory model. In *Proceedings of the 7th international workshop on Java technologies for real-time and embedded systems*, JTRES '09, pages 90–99, Madrid, Spain, September 2009. ACM Press.

This paper is organized as follows: The next section provides an overview about related work. Section 4.3 provides a short introduction to the JMM. Section 4.4 describes the cache design and explains why the proposed implementation of the JMM is advantageous for hard real-time systems. The interactions between the JMM and GC algorithms are described in Section 4.5. Section 4.6 concludes the paper and provides an outlook on future work.

4.2 Related Work

There are numerous memory models, and even more descriptions of these models. A good introduction to various memory models is provided in [1]; starting from the strongest memory model commonly used, sequential consistency, more relaxed memory models are described.

Doug Lea's "Cookbook for Compiler Writers" [16] provides an overview of how the JMM can be implemented on modern platforms such as the Intel IA-32 architecture [14]. Such architectures usually feature *fence* instructions, which act as memory barriers[1] that can be used to enforce the required ordering. They are however not directly related to higher-level operations that are often used to formally describe memory models. A compiler writer has to map the higher-level constructs to these low-level barrier operations. Our approach is different, because we do not fit the implementation onto a given memory model, but rather try to design the processor's memory system such that it obeys the rules of the JMM.

In [17] and [3], the impact of weak memory models is considered for the respective GC algorithms. They provide details about which parts of the algorithms require sequential consistency. Ordering requirements in the write barrier code are enforced by adding fence instructions. The interaction between the write barrier code and the GC algorithm proposed in [3] requires to run tracing in several phases that are repeated until tracing is done. Furthermore, additional fence instructions are required to correctly handle object updates. Levanoni and Petrank [17] exploit the fact that many processors are sequentially consistent for accesses within the same cache line. Their algorithm thereby avoids synchronization in some cases. The correct start of a GC cycle is ensured by additional handshakes between the mutators and the garbage collector. It is remarkable that the authors did not integrate the proposed solutions into their actual implementations.

Barabash et al. [5] describe the modifications to their algorithm that were necessary to correctly run on processors with relaxed memory models. Synchronization for accessing the shared mark stack is reduced by inserting fence instructions for groups of objects rather than each individual object. The second issue they consider is the fact that object tracing could see objects that are not fully initialized. They suggest to insert a fence instruction immediately after the creation of each object; for small objects, execution of the fence is batched. They note that a future revision of the JMM (i.e., JSR-133, which led to the now current specification) would deprecate parts of their solution.

Cheng [7] considers synchronization issues in the context of a replicating garbage collector. In such a collector it must be ensured that writes to the original object and to the replica object are performed in the same order. Otherwise, updates may leave reachable objects invisible to the GC algorithm.

[1]Memory barriers are unrelated to barriers in the context of GC.

4.3 The Java Memory Model

In the case of single-threaded programs, a memory model is not important. As long as the illusion is retained that the program is executed as stated by the application code, any optimization is allowed. This is unfortunately not true for multi-threaded programs, especially when running on multiprocessors. Thread-local optimizations can lead to surprising program behavior, and caching may keep updates of one thread invisible to another thread. A memory model has to specify which program behaviors are allowed w. r. t. memory accesses. Depending on the memory model, programmers can rely on different guarantees. This makes it possible to reason about the correctness of multi-threaded applications.

4.3.1 Terminology

Memory models can be described from several points of view. While some descriptions (e.g., in [1]) describe the allowed reorderings, other memory models like the JMM describe which writes a read may see. In order to allow for sound reasoning, this section presents the definitions used throughout this paper.

visible A value is *visible* to a processor if a read operation by that processor returns that value.

available A value is *available* to a processor if a read operation may return that value. A value in main memory is available to all processors, but not necessarily visible to all processors.

happens-before A partial order ("x happens-before y" is denoted by $x \leq_{hb} y$) that describes the allowed orderings of memory accesses. For example, a read operation r must not return the value of a write operation w if $r \leq_{hb} w$.

The JMM [18, 12] describes which values threads may read from memory that is shared between threads. It does so by specifying which write operations may be seen by a read operation. Memory locations on the heap are simply referred to as variables, in contrast to local variables, which are part of the run-time stack. A central definition of the JMM is the *happens-before* relation, which determines which actions have a defined ordering.

Several types of actions are distinguished by the JMM: volatile read, volatile write, non-volatile read, non-volatile write, lock, unlock, special synchronization actions, external actions, and thread divergence actions. Special synchronization actions include starting and stopping of threads. External actions interact with the environment, e.g., by printing a value. Thread divergence actions occur if a thread is stuck in an infinite loop in which no memory, synchronization, or external actions are executed. For the scope of this paper, the read/write and lock/unlock actions are most important. The volatile read/write, lock/unlock, and the special synchronization actions are referred to as synchronization actions.

Apart from the happens-before order, the JMM uses a *synchronization order*, which is a total order over all synchronization actions and is denoted by $x \leq_{so} y$. This order is used to describe the ordering of synchronization actions in an actual execution.

4.3.2 Happens-before

The JMM uses the following definition for the happens-before partial order:[2]

(1) If x and y are actions of the same thread and x comes before y in program order, then $x \leq_{hb} y$.

(2) An unlock action u on monitor m happens-before all subsequent lock actions l on m, i.e., $u(m) \leq_{so} l(m) \Rightarrow u(m) \leq_{hb} l(m)$.

(3) A write w to a volatile variable v happens-before all subsequent reads r of v, i.e., $w(v) \leq_{so} r(v) \Rightarrow w(v) \leq_{hb} r(v)$.

(4) The happens-before order is transitively closed, i.e., if $x \leq_{hb} y$ and $y \leq_{hb} z$ then $x \leq_{hb} z$.

(5) The write of the initial value (zero, `false` or `null`) to each variable happens-before the first action of every thread.

(6) An action that starts a thread happens-before the first action of the started thread.

(7) The final action in a thread happens-before any action in another thread that detects that the thread has terminated.

(8) If a thread interrupts a different thread, the interrupt happens-before any point where any other thread detects that the thread was interrupted.

(9) The end of a constructor happens-before the invocation of the finalizer for an object.

The basic rules for a programmer to understand the JMM are the first four rules in the definition above. The first rule entails that the program order is obeyed. The second and third rule establish an ordering between threads. The fourth rule causes the happens-before order to span across threads in case they synchronize.

The JMM poses two basic restrictions onto the visibility of values. Let r be a read operation that returns the value written by a write w. Then it must not be the case that $r \leq_{hb} w$. Additionally, there must be no write w' that writes to the same variable as w with $w \leq_{hb} w' \leq_{hb} r$. These definitions allow that the write a read sees is ambiguous for programs that are not properly synchronized. Proper synchronization however removes such ambiguities.

The synchronization order also implies visibility constraints. Let r be a volatile read that returns the value written by a write w. It then must not be the case that $r \leq_{so} w$ and there must be no write w' to the same variable with $w \leq_{so} w' \leq_{so} r$.

Consider the small Example 1. The initial writes of the default values to `x` and `b` happen-before the first actions of each thread. The synchronization order entails that the volatile read `!b` in T2 is totally ordered with the write `b=true` in T1, so either `!b` $\leq_{so}$ `b=true` or `b=true` $\leq_{so}$ `!b`. In the first case, the visibility rules entail that T2 can only see the initial value for `b` and is hence stuck in the loop `while(!b) {}`. When the second case becomes true, T2 must not see the initial write to `b` anymore, because `b=false` $\leq_{hb}$ `b=true` $\leq_{hb}$ `!b`. Transitivity of the happens-before relation also entails that `x=0` $\leq_{hb}$ `x=1` $\leq_{hb}$ `r1=x`. Therefore, the final read in T2 cannot see the initial write to `x` and `r1` must be 1 at the end of the execution.

[2]The definition is presented in a slightly simplified form here.

`int x = 0; volatile boolean b = false;`

Thread T1	Thread T2
`x = 1;`	`while(!b) {}`
`b = true;`	`int r1 = x;`

Example 1: Local variable `r1` is guaranteed to see `x` with value 1.[3]

`int x = 0; int y = 0;`

Thread T1	Thread T2
`int r1 = x;`	`int r2 = y;`
`if (r1 != 0)`	`if (r2 != 0)`
`  y = 1;`	`  x = 1;`

Example 2: Happens-before model allows `r1 == r2 == 1`.[3]

`int x = 0;`

Thread T1	Thread T2
`int r1 = x;`	`int r2 = x;`
`x = 1;`	`x = 2;`

Example 3: Java memory model allows `r1 == 2`, `r2 == 1`.[3]

4.3.3 Causality

Using only the happens-before relation to define the JMM is unfortunately not sufficient. Most notably, it allows values to appear out of thin air through causal loops. Consider Example 2: The reads of `x` and `y` in threads T1 and T2 are not ordered with the writes of the other thread through a happens-before relation. Even though a "first cause" is missing, each thread may see the write of the other thread, leading to `r1 == r2 == 1`. As this behavior is highly counter-intuitive, the JMM forbids such violations of causality.

The JMM defines causality requirements to validate executions that basically follow a happens-before memory model. They prohibit optimizations that could break causality such as speculative writes while still allowing optimizations such as reordering. Explaining the details of these requirements here would go beyond the scope of this paper. They are defined in terms of committing actions from an execution. If all actions can be committed when following a number of rules, the execution is valid. To show that an implementation follows the rules of the JMM, it has to be shown that the possible executions are valid under the causality requirements.

Despite the causality requirements, the JMM allows some surprising behaviors, as Example 3 shows. The read and write within each thread may not be reordered. However, the write to `x` in T2 is not ordered w. r. t. the read in T1 and vice versa. Therefore it is legal that the threads see each others writes, but none of them sees the initial write.

[3]Example taken from [18].

There is evidence [6, 25] that the causality requirements do not fully achieve what is claimed by the JMM. For example, there is a counterexample for the claim that the JMM allows the reordering of independent statements. However, it is unlikely that corrections to repair the causality requirements will impact our cache design.

4.4 Data Cache Design

Memory bandwidth is clearly the bottleneck for chip multiprocessors (CMPs). Memory access times are relatively long, especially when considering worst-case behavior. Therefore, caches are necessary to achieve acceptable performance. The Java Optimized Processor (JOP) [23] and its CMP version JopCMP [19] are designed to be timing-analysis friendly. A cache for these processors has to be timing-analysis friendly as well, otherwise one of their major design goals would be voided. Furthermore, these processors are designed to require only little hardware resources, so they can be implemented in low-cost field-programmable gate arrays (FPGAs). As the JMM is a natural memory model for a Java processor, we investigated how the JMM can be implemented efficiently while still being timing-analysis friendly.

4.4.1 Coherence and Consistency

In a multiprocessor system, two cores do not necessarily share the same view of the memory; memory accesses may appear in a different order to different processors. It is necessary to either enforce a consistent ordering in hardware, or to provide means to do so in software.

Reordering of memory accesses can appear at different levels. One level is reordering of accesses by an optimizing compiler. These reorderings do not change the intra-thread semantics, but other threads obviously observe a different ordering than specified by the program code. A second level of reordering appears in many modern processors where memory accesses can bypass each other. For example, while a read operation waits for the result from main memory, values can be written to the cache. A third, more subtle form of reordering can occur in the presence of caches without a coherence protocol. If a core issues two write operations, they can appear to a different core in reversed order if one of the values is present in the cache and becomes visible only after it has been evicted from the cache, while the other value is read from main memory.

Example 4 shows how a cache can cause writes to appear in reversed order. On the right-hand side of the column for each thread the values for the fields `A.f` and `B.f` are displayed. The two leftmost columns show which values are available for `A.f` and `B.f`, i.e., which values are in main memory. Thread T2 first reads `A.f`, which is consequently cached. Thread T1 writes to `A.f` and `B.f`. Thread T2 now reads `B.f` and `A.f`. While it reads `1` for `B.f`, `A.f` still appears to have a value of `0`. Only after T2 has invalidated its cache (or the respective cache line has been evicted) it sees the value `1` for `A.f`. The writes to `A.f` and `B.f` therefore appear to T2 in reversed order.

The definition of the *happens-before* relationship between memory accesses in the JMM is very similar to the definitions for *lazy release consistency* [15]. This memory model has some properties that make it a good match for a time-predictable multiprocessor. In the following,

Step	Thread T1	`A.f`	`B.f`	Thread T2	`A.f`	`B.f`	global `A.f`	global `B.f`
1:		0	0	`r1 = A.f;`	0	0	0	0
2:	`A.f = 1;`	1	0		0	0	1	0
3:	`B.f = 1;`	1	1		0	1	1	1
4:		1	1	`r2 = B.f;`	0	1	1	1
5:		1	1	`r3 = A.f;`	0	1	1	1
6:		1	1	`invalidateCache();`	1	1	1	1
7:		1	1	`r4 = A.f;`	1	1	1	1

Example 4: The writes to `A.f` and `B.f` appear in reversed order to Thread T2.

we will explain the concept of lazy release consistency and why our design is consistent with the JMM.

4.4.2 Lazy Release Consistency

Lazy release consistency [15] divides memory accesses into ordinary accesses (*read*, *write*) and special accesses, where especially *acquire* and *release* are important. These accesses relate to acquiring and releasing a lock. In a JVM, they appear as `monitorenter` and `monitorexit` and when accessing volatile variables. Memory updates from other processors are guaranteed to become visible to a processor when it performs an *acquire*. It is guaranteed that all memory updates until the last *release* on the same location (lock) become visible upon that *acquire*.

The primary work on lazy release consistency considered it for software distributed shared memory, which has different trade-offs than shared memory for CMPs. The original definitions for lazy release consistency include considerations for paging and a protocol to propagate updates. Our implementation can be considered as coarse-grain implementation of the original lazy release consistency protocol. Most notably, our implementation does not determine which writes actually happened and always assumes that the whole memory has been updated.

4.4.3 Consistency Implementation

We consider a write-through cache; the processor stalls until the data is actually written to main memory. Furthermore, reads stall the processor until a value is returned. Reads and writes of a processor cannot overlap. Memory arbitration guarantees that all writes to main memory are ordered. A processor may see some stale value, but it cannot "go back in time". Once it sees a memory update, it can never see a memory update on the same location that happened earlier. According to the definitions in [11], the processor is *cache coherent*.[4] However, there is no global store order, which would imply that writes to different memory locations are seen consistently by all processors. Example 4 shows how caching can violate a global store order although retaining the order for writes to an individual field.

[4]In informal contexts the term "cache coherence" is often used to describe whether inconsistent views of the memory are allowed. This usage differs considerably from some definitions of this term in the memory model literature.

The write-through cache trivially ensures that all writes are available (though maybe not yet visible) to other processors after a *release*. Therefore, all memory updates become visible to a processor when invalidating its own cache upon an *acquire*. Invalidating the cache upon an *acquire* is the only action that needs to be taken to implement lazy release consistency in the proposed infrastructure. The invalidation is part of the implementation of the `monitorenter` bytecode. For read accesses to volatile variables, we chose to insert a special `invalidate` bytecode, instead of introducing special bytecodes for each possible read access to a volatile variable. This simplifies the implementation without impairing the principal concept. Potential hazards from thread preemption between invalidation and the actual read can be avoided by invalidating the cache before resuming a thread. Cache invalidation is always a local action, which does not depend on the behavior of other cores.

The actions discussed above are not the only sources of happens-before relations. However, synchronization actions such as starting a thread are not directly visible to the hardware. Typically, the implementation of these actions implicitly establishes the required ordering. Where this is not the case, the `invalidate` bytecode can be used to enforce the appropriate ordering.

Lazy release consistency uses a happens-before relation that is consistent with the happens-before relation of the JMM. As the proposed consistency implementation is compliant to lazy release consistency, it is also consistent with the JMM. The JMM causality requirements prohibit optimizations that could violate causality. Our hardware executes memory accesses strictly in-order and does not speculatively execute any accesses. Therefore, the proposed implementation cannot violate causality.

Invalidating the whole cache upon `monitorenter` and volatile reads is costly, compared to other consistency implementations. As volatile reads usually vastly outnumber writes, a scheme that enforces consistency upon writes could be more efficient. However, such schemes provide only poor temporal predictability. The solution presented in this section enables temporal predictability, while still providing the benefits of data caching.

A potential optimization to the proposed consistency implementation is to check whether a cache invalidation is actually necessary. If no other processor performed a release since a processor performed its most recent acquire, the cache invalidation can be omitted. The hardware to enable such checks would require relatively few resources. However, this is an optimization for the average case rather than for the worst-case performance. As we are mainly interested in worst-case performance, we did not implement this optimization.

4.4.4 Analysis Friendliness

For WCET analysis, write-through caches are considerably easier to handle than write-back caches. Write-back caches require the analysis also to model the write buffer of the cache. This in turn introduces a whole new dimension into the analysis, because the analysis has to consider which words already have been written back. While this new dimension requires the analysis to be aware of the timing, the analysis for a write-through cache only requires knowledge about the possible memory access traces. The timing information can be delegated to the actual WCET computation.

The consistency implementation only requires local actions. Therefore, WCET analysis is aware of these actions without making assumptions about the behavior of other threads. This helps to minimize the pessimism for WCET bounds. Furthermore, no hardware cache coherence protocol is required, which would increase the indeterminism for memory access times. To solve the issue of unpredictable memory arbitration, a time-predictable memory arbiter [19] can be used.

As shown in [24], the ideal heap data cache for analysis should be fully associative. This is due to the fact that an access to an unknown address affects all cache sets in the analysis. Especially for a data cache, it is not always predictable which addresses are accessed—the addresses of heap allocated data are unknown until the data has been allocated.

Furthermore, the ideal replacement strategy for an analyzable cache is *least recently used* (LRU) [13, 21]. For all other replacement strategies, only a fraction of the cache can be analyzed. The downside of this is that LRU is complex to implement for highly associative caches, which is one of the reasons why it is hardly used in mainstream processors for general caches.

One way to avoid the burden of a large, fully associative cache is to split data caches for different memory areas [24]. While a small, fully associative cache with LRU replacement is used for heap-allocated data, other memory areas may use a simpler cache, e.g., a direct-mapped cache. Split caches also provide the advantage that only caches that contain shared data need to be invalidated. Caches that contain read-only data, such as the constant pool, do not need to be invalidated, because they are implicitly cache coherent.

A split data cache following the considerations in this section has been prototyped in an FPGA and integrated into JOP and JopCMP. Evaluating the cache with respect to its resource consumption, performance impact, and worst-case behavior is ongoing work.

4.5 Garbage Collection

While the previous section considered whether the memory model complies to the requirements on the Java language level, the question remains what it implies for the design of the underlying JVM. At the Java language level, synchronization needs to be symmetric: unless a program is properly synchronized, only few guarantees can be made. A processor that implements only what is necessary to comply with the JMM can trouble garbage collectors. Application threads interact with the garbage collector through barriers. For efficiency reasons, GC algorithms are designed to require as little interaction as possible. State-of-the-art real-time garbage collectors [4, 26, 22, 2] require interaction at writes to reference fields and often some small overhead for other memory accesses, e.g., to follow forwarding pointers. Reads from objects typically do not require any synchronization. The synchronization therefore becomes asymmetric and the garbage collector actively has to ensure that its actions are seen by the mutator threads.

For a runtime-system in general, a second issue arises: the runtime-system acts on a lower level than the application code, but needs to be consistent with the JMM. Starting a thread, for example, infers a happens-before edge in the JMM. The runtime code has to take this into account and synchronization at the runtime- and application-level must be coordinated.

```
void runGC() {
  // initiate new GC cycle
  startCycle();
  // retrieve roots
  gatherRoots();
  // trace the object graph
  traceObjectGraph();
  // clear objects that are still white
  sweepUnusedObjects();
  // optional memory defragmentation
  defragment();
}
```

Listing 4.1: GC cycle

For our considerations, we use the cache design described in Section 4.4 as a basis. The cache is a write-through cache, which is invalidated upon `monitorenter` and reads from volatile variables. Furthermore, memory accesses of a processor cannot overlap. Where appropriate we also describe alternative solutions and issues for other implementations.

4.5.1 GC Algorithm

In this section, we present a rather generic GC algorithm, which we assume for further reasoning. While details for specific implementations may of course vary, the listings represent the basic approach of common real-time GC algorithms.

We use Dijkstra's tricolor abstraction [8] to classify the state of objects during garbage collection. *Black* objects have been visited by the garbage collector and do not need to be visited again. Objects are *gray* if the garbage collector is aware that they need to be visited during object graph tracing. *White* objects are unvisited and considered garbage after object graph tracing has finished.

Listing 4.1 shows the basic top-level algorithm for a garbage collector. In the first step, `startCycle()`, a new GC cycle is initiated. Usually, this includes exchanging the meaning of black and white, so all objects that were marked in the previous cycle become unmarked. Some algorithms also include a handshake with the mutators such that they are aware of the start of the GC cycle. In `gatherRoots()`, the references in local and static variables are collected. The garbage collector may scan the stacks of the mutators by itself, or it may delegate this to the mutators. The latter solution requires a handshake to ensure the completeness of the roots. After the roots have been gathered, the object graph is traced and all reachable objects are marked. In `sweepUnusedObjects()`, the objects that have not been marked are recycled. The final `defragment()` step is optional and not needed for copying collectors.

A generic tracing algorithm is shown in Listing 4.2. While there are any gray objects left, an object is taken from the set of gray objects. The algorithm then iterates through all reference fields of this object and marks all white objects gray. The object itself is marked

```
void traceObjectGraph() {
  // while there are still gray objects
  while (!grayObjects.isEmpty()) {
    // get a gray object
    Object obj = grayObjects.removeFirst();
    // iterate over all reference fields
    for (Field f in getRefFields(obj)) {
      Object fieldVal = getField(obj, f);
      // mark referenced objects as needed
      if (color(fieldVal) == white) {
        markGray(fieldVal);
      }
    }
    // mark object black
    markBlack(obj);
  }
}
```

Listing 4.2: Object Graph tracing

```
void putFieldRef(Object obj, Field f, Object newVal) {
  // snapshot-at-beginning barrier
  Object oldVal = getField(obj, f);
  if (color(oldVal) == white) {
    markGray(oldVal);
  }
  // write new value to field
  putField(obj, f, newVal);
}
```

Listing 4.3: Snapshot-at-beginning write barrier

black afterwards. For a copying garbage collector, `markBlack()` would also include copying the object.

Listing 4.3 shows pseudo-code for a snapshot-at-beginning write barrier [27]. First, the old value of the field to be overwritten is read. If the reference to be overwritten is white, it is marked gray and added to the set of gray objects via `markGray()`. Afterwards, the actual update takes place.

We assume that `markGray()`, `markBlack()`, `color()` and all operations regarding `grayObjects` are properly synchronized, albeit not necessarily following the JMM. For more detailed consideration regarding the internal correctness of GC algorithms, see Section 4.5.7. The `getField()` and `putField()` methods follow the semantics of the respective bytecodes.

4.5.2 Tracing the Object Graph

Threads may have different views of the heap. A garbage collector must take this into account, so it can correctly trace the object graph without leaving any reachable objects unmarked.

Definition 1. Object graph roots*: References in local and static variables (which are directly accessible to mutator threads) are called* roots*.*

Definition 2. Correctness of tracing*: A garbage collector correctly traces the object graph if at the end of the tracing no less than all reachable objects are marked black, assuming that all roots were marked gray at the start of tracing.*

Definition 3. Consensus*: A* consensus *is a state of the object graph where each processor core and each thread has the same view of the object graph.*

When starting from a consensus and inhibiting modifications of the object graph by stopping all mutator threads, a tracing algorithm as given in Listing 4.2 is trivially correct. In such a case, the algorithm visits exactly those objects that are reachable from the root set. The situation however becomes more complex if mutator threads are allowed to modify the object graph during tracing.

Theorem 1. *The object graph can be traced correctly if*

(a) a snapshot-at-beginning write barrier is used, and

(b) new objects are allocated non-white, and

(c) a consensus is established at the beginning of tracing

Proof. When starting from a consensus, any differences in the object graph views of threads (including GC threads) must stem from modifications of the object graph. Any such update triggers the write barrier. Within the write barrier, the current value is read, marked grey, and afterwards replaced by the new value. This ordering within the write barrier is enforced by the JMM (executions must obey program order).

However, the JMM allows the situation that two concurrent field updates see the updated values of each other, but none of them sees the original value (see Example 3). In the proposed cache design, memory accesses of a processor cannot overlap. The update is not issued before the read of the old reference has returned a value. Therefore, the proposed cache guarantees that one of the threads has to see the value in the consensus. Consequently, objects that are reachable in the consensus remain visible to the garbage collector.

When storing a reference, the referenced object must be either already reachable and hence exist in the snapshot of the consensus, or it must have been newly allocated. Newly allocated objects are non-white and are thus not garbage collected in the ongoing GC cycle. All reachable objects are therefore visible to the garbage collector. □

The preconditions for Theorem 1 are not unreasonable to achieve. Most garbage collectors that are written for stronger memory models will perform this GC phase correctly also on weaker memory models than sequential consistency, as long as situations as shown in Example 3 are precluded.

```
void putFieldRef(Object obj, Field f, Object newVal) {
  // get meaning of ''white'' once
  Color w = white;
  // mark old value
  Object oldVal = getField(obj, f);
  if (color(oldVal) == w) {
    markGray(oldVal);
  }
  // mark new value
  if (color(newVal) == w) {
    markGray(newVal);
  }
  // write new value to field
  putField(obj, f, newVal);
}
```

Listing 4.4: Double barrier

4.5.3 Sliding Consensus

While conditions (a) and (b) of Theorem 1 can be easily fulfilled, establishing a consensus can provide a challenge. A straightforward way to create a consensus is by invalidating all caches at once. Unfortunately, this creates timely indeterminism for the application threads. However, such an invalidation is only necessary once per GC cycle, and the indeterminism can be taken into account without introducing severe pessimism. Still, using an algorithm that is similar to *sliding view* root scanning [10, 17, 20], would remove the need for the hardware overhead to invalidate the caches all at once.

Sliding view root scanning is a technique where each thread scans its own stack, instead of being stopped while the garbage collector scans its stack. By doing so, root scanning can be scheduled to take place when there is little data to scan and no blocking of threads is necessary. We propose that each core invalidates its own cache before scanning its local root set. By doing so, a *sliding consensus* is created.

To ensure the integrity of GC during the phase where some stacks are scanned while others stacks are not, a *double barrier* is used. Such a double barrier marks both the reference that is overwritten and the new reference. It has been shown in [20], that it is also possible to only use a snapshot-at-beginning barrier, if objects are allocated grey and the write barrier is atomic. In the following, we consider a double barrier, because it simplifies reasoning for the scope of this paper.

Listing 4.4 show pseudo-code for a double barrier. We assume here that the start of a GC cycle flips the meaning of black and white instead of re-coloring individual objects. If `white` is read immediately before the start of a GC cycle (when all reachable objects are black), neither the old nor the new value are shaded. If `white` is read immediately after the start of a GC cycle, both values are marked gray. More precisely, we say that a field update started *before* the start

Step	Thread T1	`A.f`	Thread T2	`A.f`	Thread T3	`A.f`	global `A.f`
1:	`A.f = Y;`	`X`		`X`		`X`	`X`
2:	`// A.f == white?;`	`X`		`X`		`X`	`X`
3:	`// A.f ← Y;`	`Y`	`A.f = Z;`	`Y`		`Y`	`Y`
4:		`Y`	`// A.f == white?;`	`Y`		`Y`	`Y`
				start GC cycle			
5:	`A.f = null;`	`Y`		`Y`	`scanRoots();`	`Y`	`Y`
6:	`// A.f == white?;`	`Y`		`Y`		`Y`	`Y`
7:	`// mark(Y);`	`Y`		`Y`		`Y`	`Y`
8:		`Z`	`// A.f ← Z;`	`Z`		`Z`	`Z`
9:		`Z`		`Z`	`r1 = A.f;`	`Z`	`Z`
10:	`// A.f ← null;`	`null`		`null`	`// r1 == Z;`	`null`	`null`

Example 5: Overlap of update and write barrier leaves object invisible.

of a GC cycle, if it sees the old notion of white, and that it started *after* the start of a GC cycle if it sees the updated notion.

Even without inconsistent views of the heap, it is possible to leave reachable objects invisible to the garbage collector in some cases. This is demonstrated by Example 5; a similar example was presented by Doligez and Gonthier [9]. Thread T1 writes `Y` to field `A.f`. As at the end of the GC cycle all objects are black, no object is marked. Thread T2 then writes `Z` to `A.f`; after it has checked whether anything should be marked, a new GC cycle is initiated. Thread T1 then overwrites `A.f` and marks `Y`. Before T1 actually writes to `A.f`, T2 executes the update of `A.f`, which started before the new GC cycle was initiated. Thread T3, which already has scanned its root set, reads `Z` from `A.f`. Thread T1 now actually updates `A.f`. Thread T3 has a local variable pointing to `Z`, which is however not visible to the garbage collector. Obviously, such behavior is not acceptable.

We identified the following requirements for the correctness of the GC algorithm:

Lemma 1. *The correctness of object graph tracing can be ensured if*

(a) Field updates that started before the start of a GC cycle must be visible to write barriers for the same field that start after the start of this GC cycle, unless they already have been marked by such a barrier.

(b) All field updates that started before the start of a GC cycle must be visible for local root scanning, unless such an update already has been marked by a write barrier that started after the start of this GC cycle.

(c) Field updates that started before the start of a GC cycle must be perceived consistently by write barriers and local root scanning.

Example 6 shows that the respective write must be visible as stated by Condition (a) of Lemma 1. Thread T2 writes `Y` to field `A.f`; although the write is made available to all threads, it is not visible to Thread T1. Thread T2 clears the local variable pointing to `Y`. A new GC cycle is initiated, and thread T2 scans its local root set. Thread T1 overwrites `A.f` with `Z`; as it sees `A.f` to point to `X`, it marks both `X` and `Z`, but not `Y`. Before `Z` is actually written, thread T2 reads

Step	Thread T1	`A.f`	Thread T2	`A.f`	global `A.f`
1:		X	`// r1 == Y`	X	X
2:		X	`A.f = r1;`	Y	Y
3:		X	`// A.f ← Y;`	Y	Y
4:		X	`r1 = null;`	Y	Y
			start GC cycle		
5:		X	`scanRoots();`	Y	Y
6:	`A.f = Z;`	X	`r1 = A.f;`	Y	Y
7:	`// mark(X);`	X	`// r1 == Y`	Y	Y
8:	`// mark(Z);`	X		Y	Y
9:	`// A.f ← Z;`	Z		Y	Z
10:	`scanRoots();`	Z	`invalidateCache();`	Z	Z

Example 6: Object `Y` is reachable from T2 but remains invisible to the GC.

Step	Thread T1	`A.f`	Thread T2	`A.f`	global `A.f`
1:	`// r1 == X`		`// r2 == Y`		
2:	`A.f = r1;`	X	`A.f = r2;`	Y	
3:	`// A.f ← X;`	X	`// A.f ← Y;`	Y	X
4:	`r1 = null;`	Y	`r2 = null;`	X	X
			start GC cycle		
5:		Y	`scanRoots();`	X	X
6:	`A.f = Z;`	Y	`r2 = A.f;`	X	X
7:	`// mark(Y);`	Y	`// r2 == X`	X	X
8:	`// mark(Z);`	Y		X	X
9:	`// A.f ← Z;`	Z		X	Z
10:	`scanRoots();`	Z	`invalidateCache();`	Z	Z

Example 7: Object `X` is reachable from T2 but remains invisible to the garbage collector.

`Y` from `A.f`. Finally, thread T1 scans its root set, and T2 (not necessarily on purpose) invalidates its cache. `Y` is now reachable from the local root set of T2, but not visible to the GC. Lemma 1 is violated, because the write of `Y` by T2 is not visible to T1 when its update of `A.f` begins.

Lemma 1 also requires that updates that started before the start of a GC cycle are perceived consistently by write barriers and local root scanning. Example 7 shows a case where the updates are visible to another processor, but not seen consistently. Thread T1 writes `X` to field `A.f`, Thread T2 writes `Y` to the same field. As these writes are not ordered, one of the writes "wins" the race to main memory, but a cache coherence protocol may make the write of the other thread visible. A new GC cycle is initiated, and thread T2 scans its local root set. Thread T1 overwrites `A.f` with `Z`; as it sees `A.f` to point to `Y`, it marks both `Y` and `Z`, but not `X`. Before `Z` is actually written, thread T2 reads `X` from `A.f`. Finally, thread T1 scans its root set, and T2 (not necessarily on purpose) invalidates its cache. `X` is now reachable from the local root set of T2, but not visible to the GC. As there is no ordering between the writes of `X` and `Y` to `A.f`, such behavior is legal under the JMM, but of course it voids the correctness of GC. Although such behavior cannot occur in the cache design as described in Section 4.4, future optimizations could enable this. A general GC algorithm therefore has to take into account such behavior.

We justify (though not strictly prove) the correctness of Lemma 1 with the following considerations:

- If a processor adds a reference to its local root set before root scanning, one of the following must be true:
  - The reference is still contained in the local root set at the time of root scanning. In that case, root scanning obviously makes that reference visible to the garbage collector.
  - The reference is stored to an object field and removed from the local root set afterwards. In that case, a write barrier is executed, which marks both the old and the new value. If the old value was written by an update that started before the start of the GC cycle, the value must be seen consistently. Otherwise, the overwritten value must have been written by an update that executed a write barrier and is thus visible to the garbage collector.
  - The reference is not written to any object and not contained in the local root set any more. In that case, the reference is unreachable and hence irrelevant.
- If a processor adds a reference to its local root set after it has scanned its root set, the reference must already be reachable through the local root set or newly allocated and hence visible to the garbage collector. As root scanning invalidates the cache, any divergence between available and visible values must stem from an update, which in turn has made the reference visible to the garbage collector through the write barrier.

One consequence of Lemma 1 is that threads should not be preempted while executing a write barrier. Otherwise, preempted threads would have to be resumed before garbage collection can continue. Depending on the scheduler implementation, preemption during execution of the write barrier may be avoided by making the write barrier non-interruptible or by introducing “safe points”. Threads are only be preempted at such safe points; correct placement of these safe points ensures that threads are only preempted at appropriate points of the execution.

Ensuring that updates that started before the start of the GC cycle are visible is only necessary upon root scanning and field updates. For root scanning, the cost is small, because it is executed only once per GC cycle. Visibility in the write barrier can be achieved by bypassing or invalidating the cache. The overhead for this is greater, because it appears for each write barrier. Depending on the general overhead for the write barrier code, this overhead may or may not be acceptable. For the current write barrier implementation in JOP, the overhead seems to be small enough not to notably affect performance.

4.5.4 Moving Objects

If a garbage collector moves an object, it has to take measures to ensure that the new location of the object becomes visible to the application threads. Otherwise, the application may never become aware of the change.

In both object layouts with handles and with forwarding pointers, threads must be forced to see the new location at some point. Handle-based layouts have a slight advantage, because only one location per object has to be updated, the reference to the actual object in the handle area. The interference of the garbage collector and the mutator threads is therefore limited. For object layouts with forwarding pointers, the stack and fields that reference a moved object must be patched at some point. It is therefore necessary to keep a far greater number of locations consistent.

A straightforward extension of the consistency implementation described in Section 4.4 is to allow cores to (atomically) invalidate the caches of all cores. Obviously, such a feature allows the garbage collector to make the new location of objects visible to all cores. The downside of this is that the cache behavior is not fully local anymore. WCET analysis has to take into account the effects of these invalidation. If N objects are moved in the worst case, N cache invalidations and corresponding cache fills have considered as worst-case overhead. More sophisticated cache architectures may lower the overhead—such architectures however effectively result in cache coherence protocols, which are known to be complex and introduce an unwanted amount of indeterminism to the execution time of memory accesses.

One way to keep the cache behavior fully local is to avoid moving objects. A fixed-block memory layout like in the JamaicaVM's garbage collector [26] not only eliminates moving objects, but also bounds the fragmentation. Future work will have to investigate whether it is more efficient in terms of worst-case performance to avoid moving objects or to invalidate the caches of other processors.

4.5.5 Object Creation

The JMM assumes that objects are initialized with default values before they are ever used. When the default values are written upon object allocation, it is trivial to ensure that the thread that allocates the object sees the correct values. For other threads, this is not so trivial—object fields may (in theory) still be cached by other cores. Therefore, it would be (in theory) necessary to explicitly enforce visibility initialization for all potential uses. However, this can be relaxed in some cases.

Consider that the default values are written in the context of the thread that allocates the object. The addresses where the object is allocated cannot have been accessed since the last GC cycle (otherwise, the garbage collector could not have reclaimed the space). If caches are invalidated during root scanning, no thread can have the respective memory area consciously cached. However, there is a potential pitfall: if a newly allocated object happens to be in the same cache line as a live object, fields of the new object may be cached despite not having been explicitly accessed. In such a case the allocating thread would have to invalidate or update the caches of other processors. Such situations can be avoided if objects are aligned to cache line boundaries.

4.5.6 Final Variables

Not directly related to GC, but somewhat similar to the initialization during allocation is the issue of final values. If an object reference is obtained through correct publication of the reference, threads are guaranteed to see the proper final values. As the constructor is invoked immediately after the allocation of an object, the same considerations apply as for initial field values.

If a reference is published incorrectly (i.e., before the constructor exits), threads may see the initial or the proper final field values. The implementation must guarantee that reordering does not cause references to be incorrectly published. Writes to final fields must be made available before the reference to the respective object may be used.

4.5.7 Internal Data Structures

One issue that has been left out so far are internal data structures. One example are flags to indicate phases of the GC algorithm or the color of an object. Such structures are necessary to exchange information between GC and mutator threads. Of course, it is possible to use the same mechanisms for consistency and synchronization as for the mutator threads. However, this merges the happens-before relations of these two layers and induces stronger orderings than required by the application code.

Consider a flag that is used to indicate the color of an object. This flag is read in each write to a reference field to check whether the write barrier must shade the object. Using volatile accesses for each such read not only induces a ordering on accesses to the flag, but effectively a happens-before relation between all writes to reference fields, that transitively extends to the application code. Depending on the architecture, it may be cheaper to simply bypass the cache for accessing this flag.

The GC implementor must decide how to establish the required consistency. On some architectures, it may be cheaper to bypass the cache than to wait until the cache coherence protocol has settled and a fence instruction is completed.

4.6 Conclusion and Outlook

In this paper, we presented the design of a timing analysis friendly data cache. We showed that this cache is consistent with the JMM while avoiding complex cache coherence protocols. The cache avoids non-local modifications of the cache state, which usually lead to unpredictable cache states.

We also investigated the impact of such a cache on GC algorithms. Object creation and tracing of the object graph are not affected by the cache design. However, other phases of GC cannot be implemented in a straight-forward manner. When moving objects, the garbage collector has to ensure that the new location of an object is visible to all threads. This can be achieved by either allowing global invalidation/updating of caches or by avoiding to move objects at all.

A more complex challenge occurred for the start of a GC cycle. We found that reaching a consensus by invalidating the local caches in a way that is similar to sliding view root scanning requires some overhead in the write barriers. As this overhead seems to be small enough for our current write barrier implementation, a global cache invalidation is not necessary.

Future work will on the one hand concentrate to implement an analysis for the proposed cache to classify accesses as hits or misses. This will enable a decision whether the analysis-friendliness actually pays off in terms of WCET performance. Furthermore, it will be investigated whether the consistency implementation can be optimized w. r. t. worst-case performance. On the other hand, future work will investigate the trade-off between invalidating caches to enable object moving and avoiding to move objects. Conclusive evidence on this topic will have to include performance as well as memory consumption metrics.

4.7 Acknowledgement

The author would like to thank Martin Schöberl, who provided feedback on early drafts, and Doug Lea, who helped to prepare the final version of this paper.

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *8th ACM International Conference on Embedded Software*, pages 245–254, Atlanta, GA, 2008. ACM Press.

[3] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 38(11), pages 269–281, Anaheim, CA, November 2003. ACM Press.

[4] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *30th Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 38(1), pages 285–298, New Orleans, LA, January 2003. ACM Press.

[5] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, November 2005.

[6] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. *Lecture Notes in Computer Science*, 4421:331, 2007.

[7] Perry Sze-Din Cheng. *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnegie Mellon University, September 2001. SCS Technical Report CMU-CS-01-174.

[8] Edsgar W. Dijkstra, Leslie 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(11):965–975, November 1978.

[9] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 70–83, Portland, OR, January 1994. ACM Press.

[10] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *20th Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Charleston, SC, January 1993. ACM Press.

[11] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(3a):15–26, 1990.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley Professional, Boston, Mass., 2005.

[13] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A: System Programming Guide, Part 1*, March 2009.

[15] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 13–21, New York, NY, USA, 1992. ACM.

[16] Doug Lea. The JSR-133 cookbook for compiler writers. Available at `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`. Accessed June 24, 2009.

[17] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 36(11), pages 367–380, Tampa, FL, November 2001. ACM Press.

[18] Jeremy Manson. *The Java memory model*. PhD thesis, University of Maryland, College Park, MD, USA, 2004.

[19] Christof Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.

[20] Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *6th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '08, pages 68–76, Santa Clara, CA, September 2008. ACM Press.

[21] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007.

[22] Martin Schoeberl. Real-time garbage collection for Java. In *9th International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 424–432, Gyeongju, Korea, April 2006. IEEE Press.

[23] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[24] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

[25] J. Sevcik and D. Aspinall. On validity of program transformations in the Java memory model. In *Ecoop 2008-Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008, Proceedings*, page 27. Springer, 2008.

[26] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas Books, 2002.

[27] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990.

5 Worst-Case Analysis of Heap Allocations[*]

By Wolfgang Puffitsch, Benedikt Huber and Martin Schoeberl

Abstract

In object oriented languages, dynamic memory allocation is a fundamental concept. When using such a language in hard real-time systems, it becomes important to bound both the worst-case execution time and the worst-case memory consumption. In this paper, we present an analysis to determine the worst-case heap allocations of tasks. The analysis builds upon techniques that are well established for worst-case execution time analysis. The difference is that the cost function is not the execution time of instructions in clock cycles, but the allocation in bytes. In contrast to worst-case execution time analysis, worst-case heap allocation analysis is not processor dependent. However, the cost function depends on the object layout of the runtime system. The analysis is evaluated with several real-time benchmarks to establish the usefulness of the analysis, and to compare the memory consumption of different object layouts.

5.1 Introduction

In hard real-time systems, failing to deliver results in time may lead to catastrophic consequences. Deadlines must be met even in worst-case scenarios. As such situations cannot be reliably provoked through measurements, hard real-time systems must be statically analyzed to ensure that all deadlines will be met. Scheduling analysis determines whether all tasks can meet their deadlines. The input for this analysis are the worst-case execution times (WCETs) of the individual tasks and their respective deadlines. However, it is not only important to consider the tasks' timing. An application that runs out of memory cannot deliver its result on time either. Therefore, it is important to bound not only the WCET, but also the worst-case heap allocations (WCHAs). With the known WCHAs, the memory management system, be it scoped memory or a real-time garbage collector (GC), can be correctly dimensioned.

Allocations are often "hidden" behind syntactic features or in libraries. In Java, even innocent-looking expressions such as `println("info:  "+i)` allocate several objects. The expression `"info:  "+i` is processed as follows: a `StringBuilder` object is allocated, which contains an array to store the actual characters. Then, `"info:  "` is appended to that object, potentially allocating a larger character array. The integer `i` is also appended to that object; converting a number to its decimal representation requires another character array. Finally, the `StringBuilder` is converted to a `String`, allocating yet another object. In total, two objects and two to four arrays are allocated. Considering the simplicity of the above expression, manual analysis of realistic programs is clearly not an option.

[*]This chapter has been previously published as: Wolfgang Puffitsch, Benedikt Huber and Martin Schoeberl. Worst-case analysis of heap allocations. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation (ISoLA '10)*, LNCS 6416, pages 464–478, Heraclion, Greece, October 2010. Springer-Verlag.

Manual analysis is also complicated by the fact that the requested amount of memory is not always the same as the allocated amount. In object-oriented languages, objects include some meta-information about the type of an object. Some real-time GCs (RTGCs) split objects, either to avoid [19] or to overcome [5] fragmentation issues. Programmers must have intimate knowledge about the runtime system to find out how much memory is actually allocated.

Knowledge about heap allocations is useful both when using a RTGC and when using scoped memory. Scoped memory was introduced in the real-time specification for Java RTSJ [3] to eliminate the need for garbage collection. As the size of the scoped memory area has to be provided when the area is created, it is important to know how much memory will be allocated in that scoped memory. The analysis helps to size the scoped memory area such that allocation demands can be met even in worst-case scenarios.

RTGCs have gained more acceptance since the RTSJ was formulated, and the use of scoped memory often can be avoided. Hard real-time GCs require knowledge about the application for correct operation. A RTGC must be paced correctly, otherwise it cannot keep up with the allocations from the application. In such a case the system would fail, either because it runs out of memory or because tasks are delayed beyond by the GC. The allocation rate alone is not enough to determine an upper bound for the period of the GC thread [14, 15]. However, the allocation rate is a necessary prerequisite for pacing the RTGC.

We propose to use the existing technologies for WCET analysis for the analysis of heap allocations and provide cost formulas for several object layouts. While we focus on Java, we believe that the presented ideas could also be applied to other languages. We evaluate the tightness of our approach by comparing the analysis results to measurements, and identify idioms that introduce pessimism.

The following section provides an overview of work related to this paper. Background on WCET analysis is given in Section 5.3. In Section 5.4, we present the analysis to automatically determine the WCHAs of tasks, which is evaluated in Section 5.5. Section 5.6 concludes the paper and provides an outlook on future work.

5.2 Related Work

An early attempt at automated computation of upper bounds for different performance measures was presented by Wegbreit [21]. The analyses are formulated for Lisp and computation takes place on a symbolic level. Aggregation of worst-case results must be conservative and assume that all program fragments exhibit their worst-case behavior at the same time. In contrast, our analysis uses a more powerful approach based on integer linear programming, which allows expressing cases where the execution of program fragments is not independent.

Unnikrishnan et al. presented analyses for both allocations and live memory [20], which are to some degree similar to the analyses by Wegbreit. However, recursion is handled implicitly rather than through explicit equations. The analyses are formulated for a first-order functional language and assume that the input programs are purely functional. It is not clear whether their findings can be directly applied to imperative languages such as Java.

Albert et al. [1] developed an analysis framework where a sub-set of Java bytecodes is transformed to a rule-based procedural representation. Loops are transformed into recursions, and recurrence equations are generated to characterize the program. The solution to these equations provides parametric bounds for the memory consumption.

The analysis presented by Braberman et al. [4] computes the memory usage of "regions" within a program. The memory consumptions of these regions are combined to derive symbolic bounds on the minimum memory consumption and the total amount of allocated memory.

A type-based heap-space analysis is proposed by Hofmann and Jost [8]. Programs are typed with a special type system that is used to establish bounds on the memory consumption. While this approach seems to work reasonably well for bounds that depend on the *size* of the input, it remains unclear whether bounds that depend on input *values* can be handled efficiently.

Mann et al. [11] developed a data-flow analysis to determine worst-case allocation rates. They use an instruction "window", and determine how much data is potentially allocated within such a window. Clustered allocations, as they are common at the start of a task's period, can lead to considerable pessimism when using an instruction window. Considering the whole execution of a task, as our analysis does, levels out such allocation spikes.

5.3 WCET Analysis

WCET analysis, similar to many other program analyses, is performed on a program abstraction, the control flow graph (CFG). In a CFG the basic blocks are represented by vertices and the directed edges represent possible control flows. The cost of an edge is set to the maximum time needed to execute the basic block it originates from. WCET analysis needs to find the most expensive execution path between the program's entry and exit node. In order to bound the execution time of a task or method, an upper bound for the number of times loops are executed and for recursion depths has to be known. Additionally, one aims to exclude infeasible paths, which are never taken but are part of the CFG abstraction.

A common technique to find the WCET is implicit path enumeration (IPET) [13, 10]. The problem of finding the most expensive execution path is transformed to a network flow problem. The variables of the problem correspond to the execution frequency of CFG edges. Unique start and end vertices are created with a single outgoing and a single incoming edge with execution frequency one. For all other vertices, representing basic blocks in the CFG, the flow into the vertex is equal to the flow out of the vertex. Furthermore, linear constraints to bound the maximum number of loop iterations and to exclude infeasible paths are added. Each edge is assigned a constant execution cost. The problem of finding the WCET now amounts to finding the flow with maximal cost. The solution to the resulting integer linear programming (ILP) problem can be found by a standard ILP solver, such as `lp_solve`.[1]

5.3.1 Loop and Recursion Bounds

In order to bound the WCET, it is necessary to bound the maximum number of loop iterations and the maximum depths of recursions. Simple loop bounds can be automatically extracted

[1]`http://lpsolve.sourceforge.net/5.5/`

```
for (int i = 2; i <= 10; i++) { // outer loop
  for (int j = i; a[j] < a[j - 1]; j--) {
  // @WCA loop <= 9
  // @WCA loop <= 45 outer
    swap(a,j,j-1);
  }
}
```

Listing 5.1: Loop bound annotation example

from the program source. For this purpose, a data-flow analysis (DFA) framework providing a loop bound analysis is integrated in our WCET analysis tool [17].

However, if a bound cannot be determined automatically, programmers must provide annotations. An example annotation is shown in Listing 5.1. The annotation `@WCA loop<=9` tells the analysis that the loop body is executed at most 9 times whenever the loop is entered. The annotation `@WCA loop<=45 outer` further restricts the number of times the body is executed by stating that it is executed at most 45 times whenever the *outer* loop is entered. The analysis can therefore compute tight bounds for triangular and other non-rectangular loops.

5.3.2 Data-flow Analysis

The data flow analyses are run prior to the WCET calculation, and provide information to deal with dynamic dispatch (receiver type analysis) and cycles in the CFG (loop bound analysis). Both analyses are based on the techniques described in [12], and operate directly on Java bytecode.

5.3.2.1 Receiver Types

A receiver type analysis computes which types an object may actually have. This is useful to reduce the pessimism that is introduced to the WCET/WCHA analysis by virtual method calls. The term *receiver* refers to the object which *receives* a message through the method call.

Our receiver type analysis takes call strings into account and is similar to k-CFA ("k^{th}-order control-flow analysis") [18]; a detailed description of the analysis can be found in [17]. We acknowledge that techniques like the ones described in [2] are more efficient than our approach. However, these techniques trade precision for analysis time; the amount of pessimism introduced by this loss in precision remains to be evaluated.

5.3.2.2 Loop Bounds

The loop bound analysis is based on an interval analysis that computes an upper bound for the values integer variables may hold. It is augmented with information whether a variable is only incremented or decremented. From the value range of a loop variable and its possible increments/decrements, it can be deduced how often a loop may be executed. As this analysis is not the focus of this paper, please refer to [17] for details of the analysis.

A by-product of the loop bound analysis is that it also computes ranges for array sizes. As the analysis computes ranges for all integer variables, it also computes ranges for the values that are passed to `newarray`, `anewarray`, and `multianewarray`. The only necessary change in the analysis was to keep those ranges available for further processing.

5.3.3 Execution Time Calculation

In addition to the high-level program analysis described above, the construction of a low-level timing model is necessary before calculating the WCET. The low-level analysis provides a bound on the execution time of basic blocks, and depends on the particular target platform.

Given the results of the low-level and high-level program analysis, an ILP instance is generated, with variables corresponding to the edges in the CFGs. The objective function for the ILP problem is obtained by summing up the cost of all edges. The cost of an edge is the product of the edge's frequency (a variable) and the cost of executing the basic block the edge originates from. Finally, the model is passed to an ILP solver, which calculates the edge's execution frequency on the worst-case path, as well as the WCET itself.

5.4 Heap Allocation Analysis

The WCHA analysis we propose is based on the WCET analysis described in Section 5.3. Instead of using the execution time as cost function for the analysis, we use the amount of memory a bytecode allocates. Apart from the cost function, the problem to be solved for the WCHA is identical to the problem to be solved for the WCET calculation. This implies that the infrastructure for calculating the WCET can be reused for calculating the WCHA. All path information obtained from value and loop bound analysis and the information extracted from annotations is available to the allocation analysis as well.

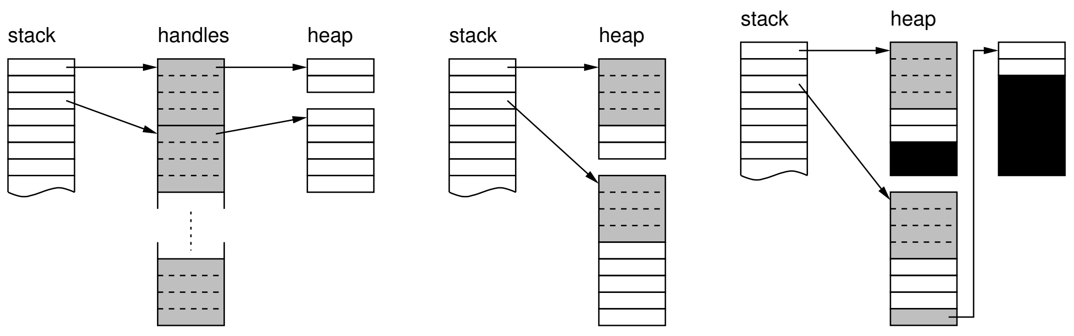
Obviously, most bytecodes do not allocate any memory. The amount of memory a `new` bytecode allocates is determined by the type it allocates. The size of instances of that type are known at compile time, and can be obtained by adding the sizes of all fields for a given class and its superclasses. When allocating arrays, the allocation size is determined at runtime. The cost functions for bytecodes that allocate data are detailed in Section 5.4.2. All other bytecodes have zero cost.

Note that we do not take into account the effects of garbage collection. Especially for multi-threaded applications, modeling the lifetime of data would require taking into account also the timely behavior of the application and runtime system, which is beyond the capabilities of the proposed technique.

In our analysis, we assume a "closed world", i.e., the application must not change during execution, and features like dynamic class loading are precluded. This assumption is necessary, because we obviously cannot analyse unknown code. However, our tool can handle virtual method calls; the cost of a call is the maximum over all possibly invoked methods.

5.4.1 Array Size Bounds

To bound the maximum size of allocated arrays, the DFA has been extended to provide array sizes at the allocation site. During the loop bound analysis, ranges for all integer values have to be computed. As this also includes values on the stack, the analysis has been extended to



(a) Handles (b) Headers (c) Fixed Blocks

Figure 5.1: Comparison of object layouts

record bounds for array allocations. When encountering a `newarray` or `anewarray` instruction, a mapping between the allocation site and the value range for the array size is added to an allocation bound table. For `multianewarray` instructions, the analysis must record multiple ranges, to bound every level of the multidimensional array. When computing the cost of an array allocation, the appropriate mapping is retrieved from the allocation bound table.

If the analysis does not succeed in computing a bound, annotations have to be provided by the programmer. An example for an array size annotation would be `arr = new int[n]; // @WCA size<=100`, where the array is annotated to occupy no more than 100 elements.

5.4.2 Object Layouts

When analyzing the WCET it is also necessary to include low-level details of the processor in the analysis. Some of the low-level architecture details, such as caches and branch predictors, are irrelevant for the WCHA analysis. However, the analysis must take into account the object layout of the underlying JVM.

Figure 5.1 shows object layout variants that are used in JVMs with RTGCs. White cells contain user data, while gray cells contain meta-information required by the runtime system. Black cells denote memory which cannot be used due to the object layout. The black cells are the result of internal fragmentation.

In a RTGC, it is necessary to either defragment memory (by relocating objects) or to avoid unbounded fragmentation. In a handle-based layout (Figure 5.1(a)), a handle in a separate handle area points to the actual object. The handle area does not need compaction, because the fixed length of the handles eliminates fragmentation. Object relocation is simple, because only the indirection pointer in the handle has to be updated. While the cost for an object is similar to a header-based layout, it has to be taken into account that also the number of allocated objects has to be bounded in order to fit the handle area.

A layout that incorporates header data (Figure 5.1(b)) into the object eliminates the indirection. However, when relocating objects, forwarding pointers have to be followed. On average, such a layout speeds up object accesses, but in the worst case, access times are similar to a handle-based layout due to the indirection through the forwarding pointer during object relocation. Such a layout also complicates defragmentation, because all references in the objects and on the thread stacks have to be updated to point to the new object location.

When using fixed block sizes for all allocations, external fragmentation can be eliminated, albeit at the expense of internal fragmentation. Such a layout is shown in Figure 5.1(c). Objects that are too large to fit into a single block are organized as linked list. Arrays are organized as a tree to achieve logarithmic costs for accesses. Individual accesses may be more expensive than with the other object layouts. However, when considering the whole system, this is alleviated by the fact that no defragmentation is necessary.

5.4.3 Cost Functions

Different object layouts lead to different cost functions for the heap allocation analysis. Object fields can be allocated packed or at word boundaries. Object meta-data (e.g., object type, array size, ...) can be organized differently to optimize for size or for speed. Furthermore, large objects can be split into constant sized blocks to avoid heap compaction or to make object relocation interruptible.

In the following, $\mathcal{F}(o)$ are the fields of an object o, and $\mathcal{F}_k(o)$ is the k-th field of object o (with indices starting at 1). With $s(f)$ we denote the size of a field f in bytes, and with $a(f)$ the required alignment for the field f. For the total memory usage of an object o we write $mu(o)$. To handle alignment requirements, we define $P(n,m)$ such that it pads the address n to a multiple of m.

$$P(n,m) = \left\lceil \frac{n}{m} \right\rceil m$$

$S(n,f)$ returns the memory usage of an object after adding a field f to that object at relative address n. For example, $S(3,f)$ would evaluate to 8 for a 4-byte field f that requires alignment to 4-byte boundaries.

$$S(n,f) = P(n,a(f)) + s(f)$$

5.4.3.1 Handle-based Layout.

In a handle-based layout, header data is stored at the handle site, while the payload is located in the remaining heap space. Together, the handle and the payload must fit the total available memory. Furthermore, it is necessary that the handle area is large enough for the handles, and the rest of the heap is large enough for the object data.

The memory usage of an object can be computed with the following formulas:

$$\begin{aligned}
mu_h(o) &= s(handle) \\
mu_f^0(o) &= 0 \\
mu_f^k(o) &= S(mu_f^{k-1}(o), \mathcal{F}_k(o)) && k > 0 \\
mu_f(o) &= P(mu_f^{|\mathcal{F}(o)|}(o), A_{field}) \\
mu(o) &= m_h(o) + mu_f(o)
\end{aligned}$$

We assume that handles are always aligned, and that any required padding or unused fields are part of $s(handle)$. Furthermore, the formulas assume that object fields start at an address that never requires padding. The maximum alignment for object fields is denoted by A_{field}. By padding the end of the user data to such an alignment boundary, we ensure that the next object starts at this boundary and its first field indeed does not require padding.

The equations for arrays are the same as for objects, except that arrays use an $array\_handle$ instead of a $handle$. The handle types can differ, because the $array\_handle$ must accommodate the size of the array and type information may be treated differently.

When being interested in the overall memory consumption, $mu(o)$ is the appropriate cost function. When considering the handle area and the remaining heap space separately, $mu_h(o)$ and $mu_f(o)$ provide the respective cost functions.

5.4.3.2 Layout with Header Data.

Header data is located in the same place as the payload. The memory usage can be computed as follows:

$$\begin{aligned} mu^0(o) &= s(header) \\ mu^k(o) &= S(mu^{k-1}(o), \mathcal{F}_k(o)) && k > 0 \\ mu(o) &= P(mu^{|\mathcal{F}(o)|}(o), A_{header}) \end{aligned}$$

Again, we assume that objects start appropriately aligned and add padding at the end of the object as required. The equations for arrays are the same, except that an $array\_header$ is used instead of $header$.

5.4.3.3 Fixed-Block Layout.

In the fixed-block object layout, the header data and the start of the object are in the same block. If the header data and the payload exceed the size of a single block, the object is split across several blocks. The link to the next block may be located at the start or the end of a block, which leads to slightly different equations for the memory usage computation. With B we denote the size of a block; we assume that fields never require padding at the beginning of a block.

Next Pointer at End of Block If the pointer to the next block is located at the end of a block, it must be checked whether the field and the $next$ pointer fit the current block when adding a field to an oject. The function S_{next} returns a value greater than B if these two fields do not fit the current block.

$$S_{next}(n, f) = S(S(n \bmod B, f), next)$$

The function S_{block} uses S_{next} to determine the actual memory usage when adding field f at position n.

$$S_{block}(n, f) = \begin{cases} S(n, f) & \text{if } S_{next}(n, f) \leq B \\ P(n, B) + S(0, f) & \text{if } S_{next}(n, f) > B \end{cases}$$

Next Pointer at Beginning of Block If the *next* pointer is located at the beginning of a block, S_{next} and S_{block} have a slightly different definition:

$$S_{next}(n,f) = S(n \bmod B, f)$$

$$S_{block}(n,f) = \begin{cases} S(n,f) & \text{if } S_{next}(n,f) \leq B \\ P(n,B) + S(s(next),f) & \text{if } S_{next}(n,f) > B \end{cases}$$

In this flavor of the fixed-block object layout, the *next* pointer must taken into account for all blocks. This is especially true for the first block; the *next* pointer for this block is considered as part of the object header.

Memory Usage of Objects The memory usage for objects can be computed with the following formulas:

$$\begin{aligned} mu^0(o) &= s(header) \\ mu^k(o) &= S_{block}(mu^{k-1}(o), \mathcal{F}_k(o)) && k > 0 \\ mu(o) &= P(mu^{|\mathcal{F}(o)|}(o), B) \end{aligned}$$

The formulas are the same for both flavors of the fixed-block object layout; the placement of the *next* pointer is already taken into account by S_{block}.

Memory Usage of Arrays Arrays have a special header, that includes the size of the array and depth of the tree representation. As all fields are the same size and have the same padding requirements, we do not need to use a recursive definition for the memory consumption. $L(a)$ captures how many array fields fit into a single block. $N(a)$ is the number of blocks the array elements occupy. M is the number of *next* pointers within an inner node of the tree representation.

$$L(a) = \left\lfloor \frac{B}{s(\mathcal{F}_1(a))} \right\rfloor \qquad N(a) = \left\lceil \frac{|\mathcal{F}(a)|}{L(a)} \right\rceil \qquad M = \left\lfloor \frac{B}{s(next)} \right\rfloor$$

$$\begin{aligned} depth(a) &= \lceil \log_M(N(a)) \rceil \\ mu^0(a) &= s(array\_header) \\ mu(a) &= \begin{cases} P(mu^0(a), B) & \text{if } |\mathcal{F}(a)| = 0 \\ P(mu^0(a), B) + \sum_{i=0}^{depth(a)} B \left\lceil \frac{N(a)}{M^i} \right\rceil & \text{if } |\mathcal{F}(a)| > 0 \end{cases} \end{aligned}$$

5.5 Evaluation

To evaluate the heap allocation analysis, three benchmarks and two applications are analyzed and the memory consumption is compared to measurements of the five applications on the target JVM. Measurements cannot reliably capture the the worst-case behavior; comparing the analysis results with measurements only hints at bounds that might be overly pessimistic.

In order to compare our work with existing memory allocation analyses, we use the `JOlden` benchmark suite [6], which was also used in [4, 1]. We use the subset of the benchmarks that does not require recursion and hence can be analysed by our tool. The benchmarks were modified such that they do not get their parameters via the command line arguments. We cannot compute a worst-case bound for unknown input values and hence initialize the appropriate variables internally. Where the DFA could not find loop bounds, we provided manual annotations.

The first application we evaluate is based on the demo application presented in [5].[2] It emulates a multi-threaded financial transaction system, which must react to market changes within a bounded amount of time. For the evaluation, we chose the methods `MarketManager.onMessage()` and `OrderManager.checkForTrade()`, both of which perform core functionality of the respective thread.

The application was adapted in three ways: First, the execution model of our execution platform is closer to the thread model of safety-critical Java [7], than the thread model of the RTSJ. The thread management therefore had to be reorganized. Second, our platform does not support the libraries for receiving and transmitting messages. This part had to be rewritten such that messages are read from and sent to standard in- and output. Third, we used string buffers without automatic resizing where suitable. The reasoning behind this is discussed in Section 5.5.3.

The second application used for evaluation was extracted from the CD_x benchmark for RTSJ [9]. The source code of the original benchmark is freely available.[3] We analyze the real-time thread responsible for collision detection of airplanes. The benchmark in its original form is unsuitable for WCET analysis, as it makes heavy use of hash tables, which have poor worst-case performance. For heap allocation analysis on the other hand, the benchmark is both challenging and, with a few modifications, within the capabilities of our tool.

We first adopted the benchmark to meet the requirements of our target platform. The recursive voxel intersection procedure, which needs large amounts of stack space, was replaced by an efficient iterative version. The number of planes and other constants were reduced to meet the memory restrictions of our embedded system. For the analysis it was necessary to annotate some loops that use iterator objects, as these are beyond the capabilities of our data flow analysis.

5.5.1 Results

The analysis results for the six benchmark methods is shown in Table 5.1. In order to keep the pessimism within reasonable bounds, we used a modified version of the Java development kit (JDK) suitable for real-time applications. It requires to pass a maximum capacity for lists and maps in the constructor, and prohibits on-demand reallocations, which cannot be handled by the analysis (see Section 5.5.3 for a discussion of the respective issues). Furthermore, the results

[2]We thank Eric Bruno and Greg Bollella for open-sourcing this demo application. It is available at `http://www.ericbruno.com`.

[3]`http://adam.lille.inria.fr/soleil/rcd/`

Table 5.1: Analysis and measurement results

		Allocated Objects		Allocated Words	
Benchmark	Method	Analysed	Measured	Analysed	Measured
MST	`MST.main()`	242	221	501	459
Em3d	`Em3d.main()`	814	805	11627	7298
BH	`Tree.createTestData()`	464	464	1954	1954
Trading	`MarketManager.onMessage()`	8	8	4104	2004
Trading	`OrderManager.checkForTrade()`	35	29	7876	741
CD_x	`Main.run()`	197936	22907	590150	73841

for the trading engine application were obtained under the assumption that input messages are at most 1024 characters long. This is enforced by the input routines, but not a constraint that is visible at the application level. For unbounded input messages, the memory consumption of the trading engine would not be boundable.

Table 5.1 compares the results of the analysis with the results of a measurement; the numbers in the last two columns of this table refer to the raw amount of memory allocated by the application, excluding object meta-data. Results that take into account the overhead of the object layout are discussed in the following section.

The figures in Table 5.1 show that the analysis yields relatively tight results for some benchmarks, while introducing a considerable pessimism for others. One reason for this pessimism is the fact that the measurement is not guaranteed to actually trigger the worst case. Some of the pessimism is however introduced by the analysis itself.

The three benchmarks from the `JOlden` benchmark suite are relatively simple and their execution is independent from input data. The analysis can therefore find reasonably tight bounds. The figures for the object count are tighter than the figures for the memory consumption because array sizes are overestimated at a few occasions. The pessimism for the object count is similar to the pessimism reported in [4] for these benchmarks.

The analysis results for the `MarketManager.onMessage()` method are off by a factor of around two. The method parses the input string for a name and a price and updates the market price of a traded item accordingly. Although the measurement was performed with a message that was designed to trigger as much memory allocation as possible, the analysis fails to find tight bounds on the string variables and assumes that all strings are 1024 characters long.

`OrderManager.checkForTrade()` shows considerably more pessimism. This is mainly caused by conversions from numbers to strings. Within these conversions, the analysis is not able to bound the length of the result string and assumes that such strings are 1024 characters in size. It is notable that the number of objects is overestimated by only about 20%, but the number of allocated words by about an order of magnitude.

The heap allocations reported for the collision detector thread of the CD_x benchmark are relatively high. The main reason for the overestimation is that it is difficult to find tight bounds for all collection sizes and loops in the benchmark, and that our tool does not yet support context dependent manual annotations. On the other hand, the benchmark showed that the

Table 5.2: Analysis results for different object layouts

			Allocated Words		
Benchmark	Method	Objects	Raw	Handles/Headers	Blocks
MST	`MST.main()`	242	501	1469	2040
Em3d	`Em3d.main()`	814	11627	14883	22776
BH	`Tree.createTestData()`	464	1954	3810	5768
Trading	`MarketManager.onMessage()`	8	4104	4136	4768
Trading	`OrderManager.checkForTrade()`	35	7876	8016	9528
CD_x	`Main.run()`	197936	590150	1381894	1915336

analysis scales up for larger programs, and helped us to identify many problematic language constructs which complicate the analysis.

5.5.2 JVM Comparison

To compare the effects of different object layouts, we variated the cost function for the WCHA analysis as described in Section 5.4.2. The results are shown in Table 5.2. We assume 4 words for header data, and a blocks size of 8 words. We also include the number of allocated objects in Table 5.2, as this number is crucial to correctly dimension the handle area for a handle-based object layout.

As it is the case on our evaluation platform, the Java Optimized Processor (JOP) [16], fields are always stored at word boundaries and do not require any further padding. Due to the simple model for alignment requirements, a handle-based layout and a header-based layout consume the same amount of memory. The heap has to be large enough to fit the number of words given in the “Handles/Headers” column. Similarly, the total memory consumption of a fixed-block layout is provided in the “Blocks” column.

For the handle-based layout, not only the total amount of memory must fit the heap, but also the handle area has to be dimensioned correctly. The number in the “Object Count” column times the handle size has to fit the handle area, and the rest of the heap has to be large enough to fit the number of words given in the “Raw” column.

When relating the total amount of allocated memory to the number of allocated objects, the trading engine benchmarks differ considerably from the other benchmarks. While the former allocate a few relatively large objects (mostly arrays), the latter allocate many small objects. The overhead for the header data is therefore considerably higher for these benchmarks than for the trading engine benchmarks.

Using a fixed-block layout increases the memory consumption by 15 to around 50%, when comparing it to a simple header layout. A GC that uses such a layout must be considerably more efficient in other areas to make up for this increased memory consumption.

5.5.3 Programming Style

During the evaluation of the analysis, we encountered several times that relatively simple operations resulted in seemingly excessive memory allocations. A closer look at these operations revealed that this was due to the automatic resizing of data structures. Except for a few special cases, the analysis assumed that such a resizing would always occur. For example, when appending characters to a `StringBuffer`, the analysis assumed that the array to hold the actual characters would be resized for each invocation of `append()`. Converting a `float` to a `String` was reported to allocate several megabytes of memory, instead of a just few dozen words. Similar effects were observed for other common data structures of the Java library, such as `ArrayLists`.

Such situations can be circumvented in two ways. On the one hand, some data structures exhibit more analysis-friendly behavior than others. For example, adding an element to a `LinkedList` requires only the allocation of a single list element. The drawback of this solution is that such data structures do not always have the desired performance characteristics. On the other hand, it is sometimes possible to size the data structure upon allocation such that no resizing is necessary. However, this solution requires programmers to correctly predict the sizes of data structures. We believe that further research is necessary to find a suitable tradeoff between analyzable memory allocation and ease of use for the respective data structures.

5.6 Conclusion

Bounds for the WCHAs of real-time tasks are needed to correctly size scoped memories or to calculate the maximum period of the GC task. We have adapted technologies from the WCET analysis field to analyze the heap allocations of tasks. Instructions that allocate memory get a cost equivalent to the the size of the allocated data structure. All other instructions have zero cost. Analyzing the program with those costs gives the maximum memory allocation for a task instead of its maximum execution time.

We have shown that our analysis can find reasonably tight bounds for moderately complex programs. However, more realistic Java programs that are not explicitly designed for real-time systems are hard to analyze and result in considerable pessimism for the WCHA bounds. As future work we plan to investigate the right Java based programming style for real-time applications. Furthermore, we will investigate better analyzable replacements for library elements of the JDK.

Bibliography

[1] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa Gil. Live heap space analysis for languages with garbage collection. In *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, pages 129–138, New York, NY, USA, 2009. ACM.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM.

[3] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[4] Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 141–150, New York, NY, USA, 2008. ACM.

[5] Eric J. Bruno and Greg Bollella. *Real-Time Java Programming: With Java RTS*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[6] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, Mar. 2009.

[8] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[9] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. Cdx: a family of real-time java benchmarks. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 41–50, New York, NY, USA, 2009. ACM.

[10] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[11] Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. *SIGPLAN Not.*, 40(7):193–202, 2005.

[12] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[13] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

[14] Sven Gestegøard Robertz and Roger Henriksson. Time-triggered garbage collection: Robust and adaptive real-time GC scheduling for embedded systems. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM SIGPLAN Notices 38(7), pages 93–102, San Diego, CA, June 2003. ACM Press.

[15] Martin Schoeberl. Real-time garbage collection for Java. In *9th International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 424–432, Gyeongju, Korea, April 2006. IEEE Press.

[16] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[17] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[18] Olin Shivers. The semantics of scheme control-flow analysis. In *PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 190–198, New York, NY, USA, 1991. ACM.

[19] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas Books, 2002.

[20] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Automatic accurate stack space and heap space analysis for high-level languages. Technical Report 538, Indiana University, April 2000.

[21] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

6 Hard Real-Time Garbage Collection for a Java Chip Multi-Processor[*]

Abstract

Garbage collection is a well known technique to increase program safety and developer productivity. Within the past few years, it has also become feasible for uniprocessor hard real-time systems. However, garbage collection for multi-processors does not yet meet the requirements of hard real-time systems. In this paper, we present a hard real-time garbage collector for a Java chip multi-processor that provides non-disruptive and analyzable behavior. For retrieving the references in local variables of threads, we propose a protocol that minimizes disruptions for high-priority tasks while still providing good bounds on the time until stack scanning finishes. Also, we developed a hardware unit that enables transparent, preemptible copying of objects, which eliminates the need to block tasks while copying objects.

Evaluation of the hardware shows that the copy unit introduces only little overhead and does not limit the critical path. Measurements resulted in release jitter for high-priority tasks of 224 μs or less on an embedded multi-processor with 8 cores clocked at 100 MHz. This indicates that with the proposed garbage collector, high scheduling quality and garbage collection do not contradict each other on chip multi-processors.

6.1 Introduction

Real-time systems keep growing more and more complex. One means to fight this complexity and increase the productivity of developers is the usage of safe high-level languages such as Java. The initial revision of the real-time specification for Java (RTSJ) [5] introduced "scoped memory" so real-time tasks can avoid the uncertainties of garbage collection. However, in the past years, real-time garbage collection has become a mature technique, at least for uniprocessors. While there exist real-time garbage collectors for multi-processor systems, most of them at some point rely on mechanisms that are not suitable for hard real-time systems. In this paper, we present a garbage collector for a chip multi-processor that truly qualifies as hard real-time. All mechanisms provide predictable behavior with bounded execution times, from the garbage collector itself down to the arbitration of individual memory accesses.

There are two key contributions of this paper: First, we describe a protocol for the stack scanning phase of the garbage collector that minimizes disturbance of high-priority tasks while at the same time has low overhead and limits the time until stack scanning finishes. Second, we present a hardware unit that enables transparent, preemptible copying of objects during the compaction phase of the garbage collector. These features make garbage collection feasible for

[*]This chapter has been previously published as: Wolfgang Puffitsch. Hard real-time garbage collection for a Java chip multi-processor. In *Proceedings of the 9th international workshop on Java technologies for real-time and embedded systems*, JTRES '11, pages 64-73, York, United Kindom, September 2011. ACM Press.

hard real-time systems that require high scheduling quality, especially with regard to release jitter.

This paper is organized as follows: The rest of this section briefly describes the key concepts of garbage collection and clarifies terminology. Section 6.2 discusses related work in the area of real-time garbage collection, focussing on scheduling of the garbage collector, the stack scanning phase and fragmentation handling. The run-time system the garbage collector is targeted at is described in Section 6.3, while the garbage collection algorithm itself is presented in Section 6.4. The implementation of that algorithm is evaluated in Section 6.5, with regard to the hardware overhead and its scheduling quality. Finally, Section 6.6 concludes the paper and provides an outlook on future work.

6.1.1 Garbage Collection Basics

Garbage collection automatically frees memory that is known not to be accessible to the application any more. Objects that are referenced by local or global variables always may be accessed by the application. These references form the *roots* of the object graph; the phase in which the garbage collector determines this root set is called *root scanning*. The part of this phase where local variables are scanned for references is called *stack scanning*. Starting from the roots, the garbage collector then *traces*, or *marks*, the object graph by following the references in the fields of objects. Afterwards, it frees the objects it has not visited during tracing in the *sweep* phase.

A garbage collector that implements these three phases is commonly called *mark-sweep* garbage collector. If it uses an additional phase to compact the heap and eliminate fragmentation, it is called *mark-compact*. In a *copying* garbage collector, tracing and heap compaction is integrated. Such a garbage collector divides the heap into two *semi spaces*; during tracing, objects are copied from one semi space to the other. Upon the start of a garbage collection cycle, the notion of the two semi spaces is exchanged, or *flipped*.

There are several possible implementations for tracing the object graph. However, the *tricolor abstraction* [9] enables reasoning about tracing without being concerned with the details of the implementation. Objects that have been visited during tracing and do not need to be visited again are *black*. Objects are *gray* if the garbage collector is aware that they need to be visited. Unvisited objects are called *white*. At the end of tracing, the objects that are unreachable from the root set are white and can be reclaimed.

A garbage collector is *concurrent* if it can execute in parallel to the application. A *parallel* garbage collector can use more than one thread simultaneously. Note that a concurrent garbage collector is not necessarily parallel and vice versa. An *incremental* garbage collector breaks down the garbage collection work into smaller steps and can be interrupted between these steps. A *stop-the-world* garbage collector pauses all application threads to perform a whole garbage collection cycle; it is therefore neither concurrent nor incremental, but still potentially parallel. In the context of concurrent or incremental garbage collection, application threads are often also referred to as *mutator* threads, because they may modify the object graph during garbage collection.

Work-based garbage collectors perform garbage collection work when mutator threads allocate memory. The simplest implementation is to perform a full garbage collection cycle if an allocation request cannot be satisfied. Garbage collection can also be performed in a separate thread; this approach is called *time-based*, because time rather than the amount of memory allocations drives the execution of garbage collection work. These approaches can also be combined, by performing garbage collection work during allocations as well as in a separate thread that uses spare computing capacities. Another combination could be to trigger a garbage collection cycle of a time-based garbage collector only after a certain amount of memory has been allocated.

Please note that we use the term *thread* when referring to a thread of execution in general, and *task* when also taking into account a thread's scheduling properties, such as its period and priority. For a specific instance of a task we use the term *job*.

6.2 Related Work

There is a wealth of literature on garbage collection in general and real-time garbage collection in particular. In this section, we try to outline at least the most important principles and techniques in this area.

6.2.1 Scheduling

Most real-time garbage collectors use a time-based approach. One notable exception is Siebert's garbage collector [29], which uses a work-based approach. However, more recent work on that garbage collector also integrates a time-based approach [31]. A combination of a work- and a time-based approach is also presented in [2].

Time-based approaches can be divided into two categories: approaches that schedule small quantums of garbage collection work at top priority, and approaches that schedule the garbage collection task at some lower priority. The latter approach is sometimes referred to as "slack-based".

The policy to schedule small quantums of garbage collection work at top priority can be found in the Metronome garbage collector [3, 1] and in a garbage collector by Kalibera [16].

Scheduling garbage collection work at a priority between high-priority real-time tasks and other tasks was proposed by Henriksson [13]. The Schism [21] and the Java RTS garbage collectors [6] use this approach. Additionally, these collectors also provide means to control the scheduling parameters to enable adaption of the scheduling policy to the application. We also follow this approach. Most importantly, it avoids the problem with scheduling at top priority that the size of garbage collection quantums limits the granularity for scheduling.

6.2.2 Stack Scanning

While scanning the thread stacks for root references, care has to be taken to avoid inconsistencies. A straight forward approach is to stop all threads and perform stack scanning atomically with regard to all threads. As this may result in unacceptably long pauses, more sophisticated mechanisms have been devised.

The approach in [1] performs stack scanning atomically only with regard to individual threads. To avoid inconsistencies, writes to references in the heap have to be guarded by a *double barrier*. In a double barrier, both the old value of a reference field and the reference to be written are marked gray, unless they are gray or black already. In [24], it has been shown that this can also be achieved by marking the old value and allocating new objects gray.

Doligez, Leroy and Gonthier [11, 10] introduced the idea of making mutator threads responsible for stack scanning. This idea was adapted for real-time systems in [24, 27]; it is proposed that tasks scan their stack at the end of a job's execution, where it is usually shallow. In this paper, we also follow this approach, but refine it to achieve tighter bounds on the time needed to complete stack scanning.

A different approach is used in Siebert's garbage collectors [29, 31]. When a thread reaches a "safe point", i.e., a point in the execution where it may be preempted, it scans its stack for references and saves them to a *root array*. After the start of a garbage collection cycle, the garbage collector has to wait until all threads have reached such a safe point and can then use the root arrays to construct the root set. This approach has the drawbacks that it introduces overhead to update the root arrays and requires additional memory for them.

Other approaches to reduce the blocking time related to stack scanning, such as stacklets [7] or return barriers [33] are sometimes discussed within the scope of real-time garbage collection. Due to their unpredictability we do not consider these mechanisms suitable for hard real-time systems.

6.2.3 Fragmentation

A hard real-time garbage collector must not void the real-time properties of the system as a whole. Obviously, it must not keep tasks from meeting their deadlines. However, it must also be possible to reason about the maximum memory consumption of a system. As fragmentation would make such reasoning difficult if not impossible, hard real-time garbage collectors must take it into account.

A solution to cope with fragmentation is to entirely avoid external fragmentation. This can be achieved by organizing the heap in blocks of a fixed size [29]. This eliminates external fragmentation at the expense of a considerable amount of internal fragmentation. Also, arrays have to be organized as trees to achieve at least logarithmic access time.

The Java RTS garbage collector [6] overcomes fragmentation by allocating objects in small fragments if necessary. The organization of fragmented objects is somewhat similar to the fixed block organization. Accesses to fragmented objects have a considerably higher overhead than accesses to non-fragmented objects. Therefore, average-case performance and worst-case behavior differ significantly.

Fragmentation can also be eliminated by defragmenting the heap, which requires the relocation of objects. During relocation, writes to the object that is about to be copied can potentially be lost or cause inconsistent data. In order to avoid consistency issues, objects are often copied atomically. Especially for large arrays, this introduces unacceptably high blocking times. In the Metronome garbage collector [1], arrays are therefore organized as arraylets, which are similar to arrays in a fixed-block layout, but at the granularity of memory pages. Fragmentation is however not completely eliminated and objects and smaller array chunks still need to be copied atomically. The approach of the Schism garbage collector [21] combines a fixed-block layout for objects with arraylets to cope with fragmentation and achieve constant access times for arrays.

Consistency during object relocation can also be achieved by writing both the old and the new location of an object. Apart from the obvious drawback of making writes more expensive, care must be taken that these writes are not reordered arbitrarily. The approach in the Sapphire garbage collector [15] exploits the fact that the Java memory model does not require sequential consistency of data. A garbage collector by Kalibera [16] is targeted at uniprocessors with green threads and therefore can avoid preemption between the two writes, but is not applicable to more general systems.

The Stopless garbage collector [20] ensures consistency during copying by clever use of compare-and-swap (CAS) operations. Unfortunately, copying may not terminate in adverse situations, which makes the approach unsuitable for hard real-time systems.

6.2.4 Copying Hardware

Using hardware to assist transparent relocation of objects has been proposed by Nielsen and Schmidt [19] and is also used in the SHAP processor [34]. The basic principle, namely to use hardware to redirect memory accesses during copying to the appropriate location, is the same as in our approach. However, these approaches implement substantial parts or all of the garbage collector in hardware. In contrast, our approach aims at a small and simple hardware implementation, while leaving most of the garbage colleciton logic in software. This paper also focusses more on the integration of the copy unit with the arbitration logic in a chip multi-processor than previous descriptions.

6.3 Run-Time System

It is pointless to strive for a predictable garbage collector on top of an unpredictable run-time system. Just as a chain is only as strong as its weakest link, a component that does not meet the requirements for hard real-time systems spoils the properties of the whole system.

The target platform for the garbage collector described in this paper is a multi-processor version of the Java Optimized Processor (JOP) [25]. It is targeted at embedded hard real-time systems and aims at simplifying worst-case execution time (WCET) analysis. This means that provable execution time bounds must be available from execution of an individual instruction up to the scheduler and the garbage collector. In the following, we describe the key components of the run-time system that enable time-predictability in our multi-processor platform.

6.3.1 Memory Access

The time to access memory must be bounded. While this is usually simple to achieve on a uniprocessor, it is less so on a multi-processor. The arbitration unit must guarantee a worst-case latency. In many systems, bandwidth is more important than latency, as long as requests are served eventually. For WCET analysis however, it is more important to achieve a low latency in order to minimize overestimations.

JOP supports both a time-division multiple access (TDMA) and a round-robin arbiter. While the former provides better predictability, the latter improves average-case performance. A round-robin arbiter is especially useful if there is a lot of locking, because the bandwidth of blocked cores can be used by running cores. However, we have not yet found a way to integrate this advantage into WCET analysis.

6.3.2 Caching

Even with very fast backing memory, memory access times in CMPs often make caching worthwhile. Although there is work towards time-predictable data caches [28, 14], our WCET analysis tool does not yet take data cache hits into account. Therefore, we assume in the following that data memory is not cached. Future versions will include caching of data memory; as we decided to implement a moving garbage collector, the caching solution must allow either cross-core cache invalidations or implement cache coherence [22].

JOP does however contain a method cache and a stack cache. As methods are constant and stacks are thread-local, these caches do not require cache coherence protocols. Accesses to the stack are known to be cache hits on JOP and are therefore trivial to analyze. Cache hits and misses on the method cache can only occur on invoking or returning from a method, and an appropriate analysis is part of the WCET analysis tool.

6.3.3 Synchronization

Synchronization takes place on more than one level. Low-level synchronization mechanisms operate on the hardware level. While they are very useful for fine-grained synchronization, they are often difficult to use both correctly and efficiently. High-level synchronization mechanisms such as software locks are usually simpler to use; their implementation however requires some form of low-level synchronization.

6.3.3.1 Low-Level Synchronization

Modern processors usually provide support for synchronization, e.g., CAS operations. Algorithms that are based on CAS or similar operations usually guarantee progress of *some* thread. In hard real-time systems, it is however usually more important to ensure progress of a *specific* thread.

On JOP, the means for low-level synchronization is a single global hardware lock. Due to the implementation in hardware, the overhead for acquiring/releasing the lock is reduced to a few cycles and comparable to the execution time of a CAS operation. The hardware implementation also ensures that threads are served round-robin; in an n-way CMP, a core must wait at worst until the $n - 1$ other cores have finished their critical section.

A single global lock severely limits parallelism. Therefore, it is only used to guard the critical sections that are required for the implementation of higher-level synchronization mechanisms. As it has little overhead, it can be used for fine-grained locking with very small critical sections without being overly expensive.

6.3.3.2 Language-Level Locking

Language level synchronization uses per-object locks, as usual in Java. When a task tries to acquire the lock, its priority is raised to a ceiling priority, which is higher than the priorities of all other application tasks and makes it effectively non-preemptible. If the lock is already held by another task, the acquiring task spins until it becomes the head of the FIFO queue and the lock is released by its current holder. The task then acquires the lock and removes itself from the queue. After having released all (nested) locks, the task returns to its normal priority.

The synchronization algorithm can be seen as special case of the MSRP protocol described in [12], with all locks being considered global. However, we do allow nesting of global locks, as we would otherwise have to preclude all nesting. The protocol therefore does not avoid deadlocks. Deadlock freedom has to be proven by the application designer.

The time between trying to acquire a lock and actually acquiring it is bounded, because a task has to wait until at most $n-1$ other requests have been served. As we assume that all tasks have a bounded execution time, critical sections also must have a bounded execution time. The $n-1$ requests therefore can be served within a bounded time, provided that deadlock freedom can be proven.

6.3.4 Scheduling

Real-time scheduling for CMPs is considerably more complex than for uniprocessors [8]. A scheduler on a CMP does not only have to decide *when* something is executed, but also *where*. On the one hand, this additional dimension makes formal reasoning more challenging. On the other hand, practical issues like the cost of thread migration further complicate things.

Our approach on this issue is straight forward: We do not allow thread migration. Every thread can be executed on exactly one processor core. Within a core, we use static priority scheduling. For the scope of this paper, we assume that a rate-monotonic priority assignment is used.

The strict partitioning of threads reduces the number of schedulable task sets compared to more flexible scheduling algorithms. However, it provides the benefit that schedulability analysis is reduced to uniprocessor schedulability analysis in many regards. One aspect that can unfortunately not be simplified is the analysis of blocking times, which must still consider that synchronization may take place across cores.

6.4 Hard Real-Time GC

The usage of the term "real-time" for garbage collectors dates back to the late 1970s [4]. However, this term has been used in very different ways since then. In this paper, we talk about hard real-time systems, i.e., systems where failure to meet a deadline may have catastrophic consequences. This means that a number of techniques for garbage collectors that are casually referred to as real-time collectors are not suitable in this context. The probably most prominent technique is generational garbage collection, which eliminates garbage collection *most* of the time, but still would cause deadline violations from time to time.

Please note that our understanding of hard real-time entails that bounds must be *provable*, and violation of these bounds must be impossible, not merely very unlikely. As example consider an algorithm that loops until a compare-and-swap operation succeeds. For usage in a hard real-time system, it must be proven that the loop terminates after a finite number of steps. Showing that it usually does so in practice is not sufficient.

A hard real-time garbage collector must be predictable with regard to both time and memory. On the one hand, it must not void the system's abilities to meet its hard real-time requirements. On the other hand, it must be possible to reason about the maximum memory consumption of a system. The execution time of the garbage collection algorithm must be bounded, and it must be provable that garbage will be eventually collected.

6.4.1 General Algorithm

According to the classification presented in Section 6.1.1, the garbage collection algorithm presented in this paper is concurrent and incremental. This means that it executes while mutator threads are executing and that it may be preempted. However, apart from stack scanning, it does not qualify as parallel garbage collector.

The decision not to parallelize the garbage collector was taken consciously. Garbage collection is a memory-bound task. On usual CMPs, parallelization also provides more bandwidth to the garbage collector and consequently provides a speed-up. On JOP in contrast, we can control the arbitration policy for the individual processor cores, and can provide the garbage collector with more bandwidth without the overhead for parallelization. Also, by avoiding parallel marking, the execution time of the marking phase does not depend on the structure of the object graph—for example, a simple linked list is naturally sequential and cannot be traced in parallel. For a discussion of limits of parallel marking, please refer to [30].

The fundamental algorithm behind our garbage collector is copying garbage collection, with extensions for making it incremental and concurrent. The initial design and the considerations behind it are best described in Chapter 7 of the JOP handbook [25]. Many of the design decisions of the initial algorithm can be found also in the algorithm described in this paper. However, the stack scanning phase has changed considerably to reduce blocking and ensure correctness on CMPs. Also, the copying unit described in Section 6.4.3, which is necessary to achieve concurrent copying on CMPs, has been developed since. The extensions described in this paper do not rely on a copying garbage collector and may be also applied to mark-compact garbage collectors.

Our garbage collector uses a handle-based object layout. In such a layout, meta data like the type of an object is located in a separate memory area instead of being located in a header to the actual object data. As references point to the handle, accesses to object fields have to follow an indirection to the actual object data. Figure 6.1 shows a handle-based layout, with references on the stack pointing to the handles and indirection pointers in the handles pointing to the actual object data in the heap.

An advantage of a handle-based object layout is that only a single location, the indirection pointer, needs to be updated when an object is relocated. As all handles are the same size, the handle area itself is not subject to fragmentation and handles do not need to be relocated.[1] In contrast, a layout with object headers, where references directly point to the object data, would require to patch the object graph and the thread stacks when moving objects. An alternative would be a fixed-block layout [29]; however, it is not yet clear whether such an approach would provide substantial benefits over our current approach. A fair comparison would at

[1]The fact that the part of the heap that is reserved for handles cannot be used by objects and vice versa may be regarded as a form of fragmentation, though with different implications.

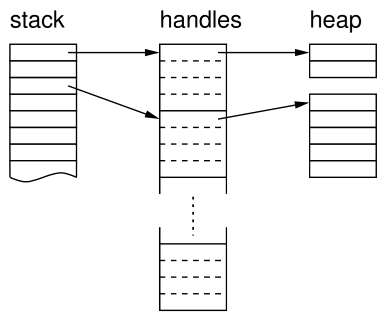


Figure 6.1: Handle-based object layout

least have to include a study on how the results for space overheads of a copying collector [26] and a fixed-block layout [23] compare in practice and whether the logarithmic access time for arrays in a fixed-block layout can be compensated by avoiding copying. Contrary to popular belief, a copying collector does not require twice the amount of memory of a mark-compact collector. Rather, the overhead depends on the amount of live memory and the allocation rates of mutator threads, which makes a comparison more complex than it might appear at first sight.

The garbage collector described in this paper uses eight word handles. While this may seem rather large compared to other garbage collectors, the handle includes apart from the indirection pointer also the object type and size, data structures for garbage collection, and a pointer to a lock. The size of the handles could be reduced to six words, though with some impact on the performance. At the moment, we do not think that the benefits from reduced memory consumption grant the performance costs, but future developments might change this.

The correctness of incremental marking is ensured by a snapshot-at-beginning barrier [32]. In order to allow modification of the object graph during stack scanning, our garbage collector supports both double barriers [1] and anthracite allocation of new objects (i.e., new objects do not need to be copied, but are pushed onto the work list of the tracing algorithm and scanned for references). The proof of correctness for the latter option can be found in [24].

As pointed out in Section 6.2, the garbage collector presented in this paper is time-based; garbage collection takes place in a separate, periodic task. In line with traditional scheduling theory, the priority for the garbage collection task is preferably chosen using rate-monotonic priority assignment. The maximum period for the garbage collection task depends on the heap size, the allocation rates, and the amount of live memory. Details on how such a bound can be computed can be found in [26].

A garbage collector usually consumes considerably more execution time per release than other real-time tasks. In order to be able to meet its deadline, its period will typically also be larger than the periods of other tasks. In the following, we therefore assume that the garbage collector executes at lowest priority, as entailed by rate-monotonic priority assignment for the task with the longest period. However, the proposed solutions do not necessarily require this assumption, and tasks with longer periods or non-real-time tasks may execute at lower priorities than the garbage collector.

6.4.2 Stack Scanning

The garbage collector presented in this paper uses the approach from [24] that tasks scan their stack at the end of a job's execution, where it is usually shallow. While scanning the stack at the end of execution reduces the effort, it introduces a delay for the garbage collector—garbage collection cannot proceed until stack scanning has completed. In general, this delay depends on the period or the response time of the tasks (depending on the implementation). Tasks with long response times can slow down garbage collection to the point where it cannot keep up with the allocations any more.

To overcome this limitation, we propose a strategy that does not block high-frequency tasks while still providing reasonable bounds for the time to complete stack scanning.

- High-frequency tasks scan their own stack. They can do so when their stack is shallow, which minimizes overhead. As they execute at a high frequency, the time until up-to-date roots are available to the garbage collector remains short.
- The stacks of low-frequency tasks are scanned by stack scanning events. These events have a priority that is higher than the priority of the tasks whose stacks it scans, but lower than the priority of high-frequency tasks. Therefore, stack scanning of these events can be accounted for with usual schedulability analysis.

There is usually one stack scanning event per core, which scans the stacks of all lower-priority tasks. On cores where all tasks scan their own stack, the stack scanning event may be omitted. In theory, it would be possible to have more than one stack scanning event. However, we believe that situations where this could provide benefits are rare in practice. Therefore, and for the sake of simplicity, we assume that there is at most one stack scanning event per core in the following.

The firing of stack scanning events is done via cross-core interrupts, which trigger the scheduler. As this interrupt is masked during the execution of high-priority tasks, we achieve low latency without disturbing the execution of high-priority tasks.

Figure 6.2 exemplifies the proposed stack scanning scheme. Tasks τ_1 to τ_6 are application tasks, τ_{s1} and τ_{s2} are stack scanning events, and τ_{gc} is the garbage collection task. Shaded areas indicate stack scanning, while white areas denote regular execution. The task set executes on two processor cores, as indicated by the line between τ_3 and τ_4. Tasks τ_1 and τ_4 have quite tight deadlines; they can afford to scan their own stacks, but any preemption or delay would cause them to miss their deadlines.

At time 0, the garbage collector starts a new cycle by flipping the notion of the semi spaces, and subsequently triggers the stack scanning events for both cores; the dashed arrow represents the firing of the event. Execution of τ_{s1} is delayed by the execution of τ_2. Task τ_{s2} starts its execution immediately, just to be preempted by τ_4 at time 5. Tasks τ_1, τ_2 and τ_4 scan their own stacks and are not disturbed by stack scanning. At time 15, τ_1, τ_2 and τ_4 have scanned their own stack, and the stacks of tasks τ_3, τ_5 and τ_6 have been scanned by the stack scanning events. At this point, τ_{gc} can proceed and begin to traverse the object graph.

As usual, we use T_i for the period of task τ_i and R_i for its response time. Furthermore, σ is the set of tasks that scan their own stack, and ρ is the set of stack scanning events. For a task $\tau_i \in \sigma$, the time to scan its own stacks is part of the WCET and therefore included in R_i. For

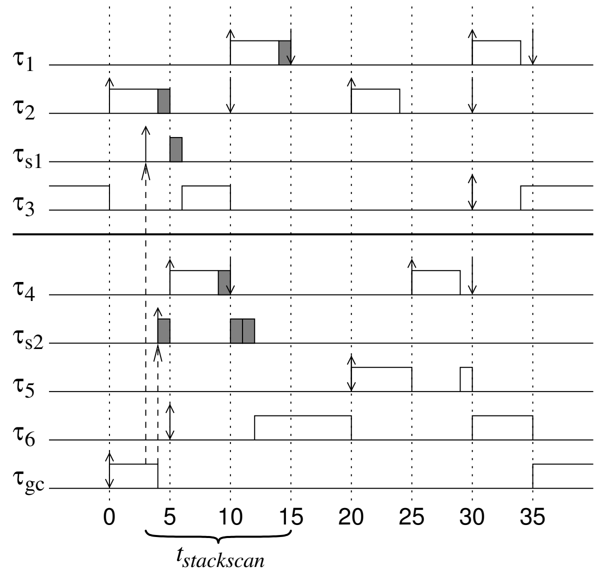


Figure 6.2: Stack scanning example

the events in ρ, the WCET includes the time to scan the stacks of all lower-priority tasks on the same core. An upper bound for the maximum time between triggering stack scanning and its completion, $t_{stackscan}$, is given by

$$t_{stackscan} \leq \max(\max_{\tau_i \in \sigma}(T_i + R_i), \max_{\tau_i \in \rho} R_i)$$

For the tasks in σ, we must take into account both their period and their response time. In the worst case, the garbage collector notifies a task τ_i that it should scan its stack just after the task has decided not to do so. The garbage collector then has to wait for T_i until the task is released again, and R_i until the task has finally scanned its stack. As the events in ρ are triggered by the garbage collector, it is sufficient to take their response time into account.

The garbage collector needs to wait until stack scanning has finished until it may proceed. Therefore, the choice which tasks scan their own stack and which have their stack scanned scanned by a stack scanning event influences its response time. This choice also influences the response time of the stack scanning events. Finding an optimal solution and integrating the above results with schedulability analysis however go beyond the scope of this paper and remain future work.

6.4.3 Copying

When relocating objects during garbage collection, care has to be taken to keep data consistent. A write to an object that is about to be copied could be missed if the write goes to the old location of the object, or might be overwritten if it goes to the new location of the object.

The solution presented in this paper builds on a hardware unit to support transparent, preemptible copying of objects. A uniprocessor version of such a unit was presented in [27]. The unit presented in that paper is part of a core's memory unit and redirects memory accesses

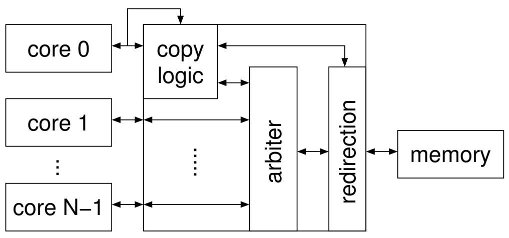


Figure 6.3: Block diagram of copy unit

to fields that already have been copied to the new location. However, as this unit is local to a core, it can do the translation only for a single core.

For a CMP copy unit, we face the following requirements: First, copying must be preemptible, transparent and must retain data consistency. Second, copying must not influence the arbitration of memory requests of other cores. Third, it should be possible to reuse existing arbiters. The first requirement basically states what we want to achieve. The second requirement entails that we cannot simply lock the arbiter in order to achieve consistency, because doing so would influence the timing of memory accesses for other cores. Consistency must be achieved by taking into account memory access from other cores between reading and writing the data to be copied rather than preventing the other cores from accessing memory. Also, the copy unit should either have its own slot for arbitration or use the slots of the core that issues the copying command rather than stealing bandwidth from other cores. The third requirement takes into account the fact that arbiters are non-trivial pieces of hardware and rewriting an arbiter is not a task to be taken lightly. However, the requirement slightly complicates copying, as even a TDMA arbiter reorders memory accesses and it might not be possible to tell copying related memory accesses from ordinary memory accesses after the arbitration.

Figure 6.3 shows a block diagram of the copy unit. While the connections from cores 1 to $n-1$ are forwarded directly to the arbiter, core `0` is special and the only core on which the garbage collector may execute. Normally, accesses from core `0` are also just forwarded to the arbiter. However, if core `0` triggers a copying step, the copy logic uses the arbitration slot of core `0` to read from the source and write to the destination location. After arbitration, the redirection logic changes the addresses of memory accesses if necessary. It is connected to the copy logic in order to communicate whether copying is in progress, the source, destination and offset of the current copying process. As a location that is about to be copied may be written to, the redirection unit must also detect such writes and update the internal buffer accordingly. While this is to some degree similar to cache coherence, it is fortunately simpler to retain consistency on a single word than on a whole cache.

Listing 6.1 shows how the copy unit is utilized by the garbage collector. As the ports of the copy unit are part of the local I/O space, accesses to them are not subject to arbitration. In the first lines, the source and destination for copying are stored in the copy unit, and the unit is activated. The actual copying is issued by writing the position to be copied to address `Const.IO_CCCP_POS`. After updating the indirection pointer, it is necessary to wait before turning off the copy unit again. Otherwise, other cores might use the old location for accesses without redirection, resulting in inconsistent data. The value of the redirection pointer

```
// copy object
Native.wr(addr, Const.IO_CCCP_SRC);
Native.wr(dest, Const.IO_CCCP_DST);
Native.wr(1, Const.IO_CCCP_ACT);

for (i = 0; i < size; i++) {
  // @WCA loop <= MAX_SEMI_SIZE outer
  Native.wr(i, Const.IO_CCCP_POS);
}

// update object pointer to the new location
Native.wr(dest, ref+OFF_PTR);
// wait until everybody uses the new location
for (i = 10; i > 0; --i); // @WCA loop = 10
// turn off address translation
Native.wr(0, Const.IO_CCCP_ACT);
```

Listing 6.1: Copying code

is stored locally only in a few spots in the run-time system. Analysis of the respective code fragments showed that the time between reading and using an indirection pointer is always bounded. Therefore, a simple delay loop is sufficient to ensure correctness.[2]

The listing also shows annotations for WCET analysis, `@WCA`. The delay loop towards the end of Listing 6.1 is obviously executed 10 times. The actual copy loop is only bounded by the size of a semi space—in the worst case, a single array that occupies the whole semi space has to be copied. However, no more than a semi space can be copied during the whole tracing phase. The annotation therefore relates the loop bound to the bound of the `outer` loop, stating that at most `MAX_SEMI_SIZE` words are copied in total.

6.5 Evaluation

6.5.1 Hardware

For the evaluation, we use the CMP version of JOP with 8 cores, which runs at 100 Mhz on an Altera DE2-70 board. This board contains 2 MB synchronous SRAM; both reads and writes take 3 cycles.

In order to assess the overhead for the copy unit, we synthesized a plain variant of the processor and a variant of the processor that includes the copy unit. In order to be useful, the size of the copy unit must not slow down the critical path, which would require to run the processor at a lower frequency. Also, it must be reasonably small so the associated cost remains low.

[2]Of course, it must be guarded appropriately against being eliminated by optimizations.

Table 6.1: Synthesis results

Configuration	LCs	Registers	f_{max} (MHz)
zero-cycle	45 077	14 079	93.51
registered	45 308	13 677	99.86
copy unit	45 525	14 275	100.03

The original version of the TDMA arbiter allows zero-cycle arbitration, i.e., cores that issue a memory access are granted access immediately if they happen to hit their slot. In a configuration with 8 cores, 3-cycle memory accesses and uniform 3-cycle slots, this results in a worst-case memory access time of 26 cycles. Two cycles have to be taken into account, because accesses can only be issued in the first cycle of the slot—otherwise the access would occupy parts of the next core's slot. It then takes 21 cycles until the slots of the other cores have passed, and further 3 cycles for the actual memory access to be completed.

Unfortunately, zero-cycle arbitration results in a relatively long path from the core to the memory interface. Inserting more logic into that path would have lowered the maximum frequency considerably. Therefore, we decided to make the arbiter fully registered. This results in a slightly higher worst-case memory latency of 27 cycles instead of 26 cycles—one cycle being added by the registers—but does not reduce the available bandwidth and allows the processor to run at the maximum frequency the core provides.

Table 6.1 shows the results of the hardware synthesis for three variants of JOP. The line labeled "zero-cycle" shows the results for the original version of JOP. It is notable that the maximum frequency f_{max} is only 93.51 MHz; the critical path is between the cores and the memory interface. Breaking the critical path by replacing the zero-cycle arbiter with a registered arbiter increases the maximum frequency to 99.86 MHz. The reduced pressure on the paths between the cores and the memory interface leads to different optimizations being applied by the synthesis tool. Therefore, this version requires more logic cells (LCs) to be implemented, but fewer registers.

Optimization effects also lead to the paradoxical effect that adding the copy unit to JOP increases the maximum frequency to 100.03 MHz. As the critical paths are in a part of the processor that did not change between the two versions, we assume that the difference is caused by the heuristic nature of fitting, place and route algorithms, where even minimal changes to the hardware design can cause random variations of the results.

The size of the copy unit is not exactly the difference between the "registered" and the "copy unit" configurations. The actual size of the copy unit is 347 LCs, of which 178 are registers. Compared to the overall size, we consider this overhead negligible.

6.5.2 System Evaluation

As benchmark we use a Java port of the Papabench [18] benchmark, jPapabench [17]. The benchmark implements an autopilot for an unmanned aerial vehicle and was ported to Java in order to provide a realistic benchmark for real-time Java implementations.

Table 6.2: jPapabench tasks

#	Task	Period (ms)
A1	RadioControl	25
A2	Stabilization	50
A3	LinkFBWSend	50
A4	Reporting	100
A5	Navigation	250
A6	AltitudeControl	250
A7	ClimbControl	250
F1	TestPPM	25
F2	SendDataToAutopilot	25
F3	CheckFailsafe	50
F4	CheckMega128Values	50
S1	SimulatorFlightModel	25
S2	SimulatorGPS	250
S3	SimulatorIR	50

Table 6.3: Task partitioning

	Core	0	1	2	3	4	5	6	7
Priority	High				F1	F2			
		A3			F3	F4	A1	A2	
		SE	SE	SE			SE	SE	SE
		A4	S1	S2			A7	A6	A5
	Low	GC		S3					

Table 6.2 shows the task set of jPapabench, including the garbage collection task. The periods for the tasks are the same as in the original benchmark. Table 6.3 shows how the assignment of tasks to individual cores, including the garbage collection task and the stack scanning events. The tasks are sorted vertically to respect their priorities. For example, A3 has a higher priority than the stack scanning event SE on that core. Tasks A1 to A7 and F1 to F4 are considered real-time tasks and have to meet their deadlines. With the exception of the simulation tasks S1, S2 and S3, tasks with a period of 50 ms or less scan their own stacks and therefore have a higher priority than the stack scanning events. The cut-off period of 50 ms was chosen in order to have the stack scanning events at different relative priorities (top or medium priority) on the individual cores. On cores 3 and 4, there are no low-priority tasks, so the stack scanning events could be omitted. Relative priorities within a core are the same as in the original benchmark, except for A2 and A6, where the relative priorities are inverted so A2 has a higher priority than the stack scanning event. The priorities are mostly rate-monotonic, with the exception being the simulation tasks S2 and S3.

The simulation tasks S1, S2, and S3 are not hard real-time tasks, but are needed to simulate the environment and provide a self-contained benchmark. They are grouped on cores 1 and 2 to isolate their execution from the scheduling of the hard real-time tasks. Furthermore, tasks from the fly-by-wire module F1 to F4 are grouped on core 3 and 4, while the autopilot tasks A1 to A7 are grouped on cores 5, 6 and 7. As jPapabench is not a complete port of the original Papabench, the tasks A3 and A4 are empty. To evaluate the impact of the garbage collector on tasks that run on the same core, we use these tasks without compromising the correctness of the overall system and assigned them to core `0`.

6.5.2.1 Analysis

In order to get a better understanding of the behavior we can expect for the benchmark, we analyzed the WCETs of the individual tasks. The reported times do not include blocking, but otherwise reflect the worst case with regard to timing, even in the implementation of complex bytecodes such as `checkcast`. The analysis results are shown in Table 6.4.

For most tasks, the execution times seem to be reasonable. For A5 and the simulation tasks however, the WCETs seem unreasonably high. Further investigation revealed that these tasks use floating-point operations. These are implemented in software on JOP, which is costly even when considering average-case performance. As the analysis must assume the worst-case behavior for every single operation, the WCET contains considerable overestimation.

When comparing the WCETs and the task periods, it becomes clear that the tasks that use floating-point operations potentially violate their deadlines. For the scope of the evaluation, we left the tasks and the periods as they were. In a real system of course, appropriate measures would have to be taken to avoid the risk of deadline violations.

Apart from the execution time, we also analyzed the worst-case heap allocations (WCHAs). We analyzed the number of objects that are allocated per release and their raw size separately. In total, the tasks allocate 504 objects consuming 9.75 KB per second. When taking into account not only the actual object data, but also the eight words of the meta data in the handles, the byte per second figure rises to 25.50 KB. While these figures are probably not impressive, please remember that we aim to evaluate predictability rather than performance. Of course, we would have liked to increase the task frequencies to put higher stress onto the garbage collector. However, the tasks that use floating-point operations already require a higher budget than we can provide. Therefore, we have refrained from doing so.

We also performed a preliminary WCET analysis on the garbage collector. The analysis reported a WCET of around 25 seconds. However, we are aware of several sources of overestimations in the analysis. For example, we can formulate constraints that the number of handles limits the number of objects and the number of arrays to consider for tracing, but not that the sum of objects and arrays cannot be larger than the number of handles. Also, it is assumed that all handles must be sweeped, although an object that has been visited during tracing must not have its handle sweeped. The tightness of the WCET bound could also be improved by taking into account the number of reference fields in objects instead of assuming in the analysis that every field might be a reference. Eliminating these sources of overestimation and potentially other, less obvious ones, requires considerable changes in our WCET analysis tool and remains future work.

Table 6.4: Analysis results

#	WCET (μs)	WCHA	
		objects	bytes
A1	3502	3	68
A2	4697	3	68
A3	1	0	0
A4	1	0	0
A5	155511	0	0
A6	636	0	0
A7	2636	0	0
F1	299	0	0
F2	2580	5	112
F3	112	0	0
F4	2161	3	68
S1	579361	1	20
S2	397641	1	20
S3	397641	1	20

Interestingly, copying is cheap compared to other parts of the garbage collector, being responsible for 117 ms of the overall WCET. Although this is not a definite proof, it indicates that avoiding copying is unlikely to provide a significant performance gain.

6.5.2.2 Measurements

For our measurements, we used the `BraunshweigFlightplan`, which is part of jPapabench and takes about 200 seconds to complete. In order to maximize the interference from the garbage collection task, we chose a period of 199 ms, which is roughly its average observed response time and is not evenly divided by the periods of other tasks. We measured both with all release offsets set to 0 ms and with release offsets such that higher-priority tasks are released 1 ms after lower-priority tasks.

The maximum observed response time (MORT) and the maximum observed release jitter (MOJ) over ten benchmark runs are shown in Table 6.5, both for a version with atomic copying and a version with the copy unit. The MORT is the time between the (theoretical) release of a job and its observed completion. It includes release jitter, blocking times, and scheduling overhead. Therefore, it cannot be related directly to the worst-case execution times of the tasks that are executed on the respective core. The MOJ is the difference of the earliest and the latest start of the execution of a job relative to its release time. The results for S1 are not shown, because that task almost always violated its deadline, which rendered the measurement results useless.

In order to assess the MOJ we can expect to see from the benchmark, we measured the MOJ for a set of dummy tasks that continuously increments a shared counter within a synchronized

Table 6.5: Measurement results

#	atomic copy		copy unit	
	MORT (μs)	MOJ (μs)	MORT (μs)	MOJ (μs)
A1	1826	870	533	65
A2	3904	921	2622	9
A3	989	982	145	139
A4	3536	3529	2174	2168
A5	22835	381	24793	12
A6	3935	3639	3502	3123
A7	3449	1832	2461	964
F1	1188	1171	103	86
F2	1261	1239	246	224
F3	1605	1588	760	743
F4	4225	1863	2407	764
S1	–	–	–	–
S2	39900	414	38524	76
S3	44511	39616	43012	37979

block and an empty high-priority task. For the high-priority task, we observed a MOJ of 199 μs. While this may seem to be a rather large amount of jitter, it must be noted that a high-priority task cannot execute while a low-priority task on the same core waits for a lock or is within a synchronized block. In the worst case, the low-priority task has to wait until tasks on the $N-1$ other cores have completed their critical section. As critical sections in the benchmark do more useful work than just incrementing a counter, we can expect a somewhat higher jitter than 200 μs.

With atomic copying, we observe a jitter of several hundred μs even for the highest priority tasks. This indicates that copying has a considerable effect on the scheduling quality. With the copy unit in contrast, we see release jitter of well under 100 μs for tasks A1, A2, A5, F1, and S2. Scheduling on JOP includes the replacement of the stack cache contents upon tasks switches; considering that 1 μs is sufficient only to load 4 words of memory, we think that the observed jitter is highly satisfactory. For task F2, we observe jitter of 224 μs even with the copy unit. This can be explained by the fact F4 cannot be preempted while holding the garbage collector lock during memory allocations and therefore delays the release of F2. Compared with the 200 μs observed for dummy tasks, the MOJ for F2 seems to be reasonable. The MOJ for A3 is also satisfying, which indicates that it is feasible to execute high-priority tasks on the same core as the garbage collector.

The maximum observed time for stack scanning was 44.4 ms, which is in line with the theoretical bound presented in Section 6.4.2. According to the task periods and the measured response times, up to around 60 ms would have been acceptable.

6.6 Conclusion

In this paper, we presented a garbage collector that is suitable for hard real-time systems on chip multi-processors. The proposed stack scanning protocol requires little overhead, provides reasonable timing bounds, and minimizes interference between garbage collection and high priority tasks. We also presented a hardware copy unit that enables transparent and preemptible copying on a chip multi-processor. In measurements, the release jitter of high-priority tasks was reduced significantly. The results indicate that high scheduling quality on chip multi-processors is achievable even in the presence of garbage collection.

Future work will focus on the analysis of the garbage collector; WCET analysis currently suffers from severe overestimations. These overestimations need to be reduced in order to compute bounds that are not only safe but also tight.

Availability

The platform described in this paper is open source and available via `git clone git://www.soc.tuwien.ac.at/jop.git`. The version used for the evaluation is located in the branch `rtgc` and tagged `jtres11`. For detailed instructions how to reproduce the experiments, please refer to file `README.JTRES2011`.

Bibliography

[1] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *7th ACM & IEEE International Conference on Embedded Software*, pages 249–258, Salzburg, Austria, September 2007. ACM Press.

[2] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *8th ACM International Conference on Embedded Software*, pages 245–254, Atlanta, GA, 2008. ACM Press.

[3] David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM SIGPLAN Notices 38(7), pages 81–92, San Diego, CA, June 2003. ACM Press.

[4] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978. Also AI Laboratory Working Paper 139, 1977.

[5] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[6] Eric J. Bruno and Greg Bollella. *Real-Time Java Programming: With Java RTS*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[7] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 33(5), pages 162–173, Montreal, Canada, June 1998. ACM Press.

[8] R.I. Davis and A. Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. techreport YCS-2009-443, University of York, Department of Computer Science, 2009.

[9] Edsgar W. Dijkstra, Leslie 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(11):965–975, November 1978.

[10] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 70–83, Portland, OR, January 1994. ACM Press.

[11] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *20th Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Charleston, SC, January 1993. ACM Press.

[12] Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '03, pages 189–, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[14] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. Worst-case execution time analysis-driven object cache design. *Concurrency and Computation: Practice and Experience*, 2011. published online, to appear in print.

[15] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3–5):223–261, 2003.

[16] Tomas Kalibera. Replicating real-time garbage collector for Java. In *7th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '09, pages 100–109, Madrid, Spain, September 2009. ACM Press.

[17] Tomas Kalibera, Pavel Parizek, Michal Malohlava, and Martin Schoeberl. Exhaustive testing of safety critical java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 164–174, New York, NY, USA, 2010. ACM.

[18] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In Frank Mueller, editor, *WCET*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[19] Kelvin D. Nilsen and William J. Schmidt. Cost-effective object-space management for hardware-assisted real-time garbage collection. *Letters on Programming Language and Systems*, 1(4):338–354, December 1992.

[20] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. Stopless: A real-time garbage collector for multiprocessors. In Greg Morrisett and Mooly Sagiv, editors, *6th International Symposium on Memory Management*, pages 159–172, Montréal, Canada, October 2007. ACM Press.

[21] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 45(6), pages 146–159, Toronto, Canada, June 2010. ACM Press.

[22] Wolfgang Puffitsch. Data caching, garbage collection, and the Java memory model. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 90–99, New York, NY, USA, 2009. ACM.

[23] Wolfgang Puffitsch, Benedikt Huber, and Martin Schoeberl. Worst-case analysis of heap allocations. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *Lecture Notes in Computer Science*, pages 464–478. Springer Berlin / Heidelberg, 2010.

[24] Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *6th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '08, pages 68–76, Santa Clara, CA, September 2008. ACM Press.

[25] Martin Schoeberl. *JOP Reference Handbook*. Number ISBN 978-1438239699. 2009.

[26] Martin Schoeberl. Scheduling of hard real-time garbage collection. *Real-Time Systems*, 45(3):176–213, 2010.

[27] Martin Schoeberl and Wolfgang Puffitsch. Nonblocking real-time garbage collection. *ACM Trans. Embed. Comput. Syst.*, 10:6:1–6:28, August 2010.

[28] Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In Sunggu Lee and Priya Narasimhan, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 5860 of *Lecture Notes in Computer Science*, pages 180–191. Springer Berlin / Heidelberg, 2009.

[29] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas Books, 2002.

[30] Fridtjof Siebert. Limits of parallel marking collection. In Richard Jones and Steve Blackburn, editors, *7th International Symposium on Memory Management*, pages 21–29, Tucson, AZ, June 2008. ACM Press.

[31] Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In Jan Vitek and Doug Lea, editors, *9th International Symposium on Memory Management*, pages 11–20, Toronto, Canada, June 2010. ACM Press.

[32] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990.

[33] Taiichi Yuasa. Return barrier. In *International Lisp Conference*, 2002.

[34] Martin Zabel, Thomas B. Preußer, Peter Reichel, and Rainer G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 59–62. IEEE Computer Society Press, 2007.

7 Conclusions and Outlook

The previous chapters brought forward several ideas to make garbage collection feasible for hard real-time systems on chip multi-processors. The results presented in Chapters 5 and 4 already considered systems other than the Java Optimized Processor (JOP) [14]. The solutions presented in Chapters 2, 3 and 6, however, are built around JOP and are specific to this execution platform. The following section outlines how these results may be generalized for other execution environments. Section 7.2 presents the conclusions of this thesis, while Section 7.3 provides an outlook on future work.

7.1 Generalization

The implementations presented in this thesis are based on JOP. This processor is very specialized, and is considerably different from more widespread general-purpose processors. Most notably, it directly executes Java bytecode and is therefore a stack machine, whereas most other processors are register machines. JOP supports only Java and can therefore omit features that would be necessary for the execution of less restrictive languages such as C. On the other hand, it implements instructions that make the execution of Java programs more efficient, but are not found in common RISC instruction sets. For example, array accesses on JOP resolve the handle indirection and perform array bound checks in hardware. As JOP is targeted at hard real-time systems, its implementation favors predictability over average-case performance. As the specialization of JOP might bias the results presented in this thesis so far, this section investigates if and how the techniques developed in the scope of this thesis can be applied to more general execution environments.

7.1.1 Stack Scanning

The stack scanning techniques presented in Chapters 2 and 6 are independent from the underlying processor. Therefore, they could also be implemented in processors other than JOP. However, the descriptions in those chapters assume fixed-priority scheduling and that tasks do not migrate between cores. Letting tasks scan their own stacks would also be possible when loosening these requirements. The fundamental requirement that tasks do not modify their stack while it is being scanned is still fulfilled.

Stack scanning events for low-priority tasks as presented in Chapter 6 require fixed priorities and that tasks cannot migrate between processor cores. Otherwise, it would not be possible to guarantee the consistency of the root set. It would be possible to implement a scheduler that enforces these requirements while stacks are being scanned and uses a different scheduling algorithm otherwise. However, this would complicate the formal treatment of the emerging scheduler considerably, which would be a serious disadvantage.

The default scheduling policy for the Real-Time Specification for Java (RTSJ) [3] and Safety Critical Java (SCJ) [9] is priority-based preemptive scheduling. The solutions presented in this thesis are compatible with this scheduling policy. An implementation of the Java Virtual Machine (JVM) that conforms to the the RTSJ or the SCJ specification could make use of these solutions. The restrictions outlined above are only an issue for implementations that choose to deviate from the default behavior. Therefore, it can be expected that only few implementations would have to work around these restrictions.

Using dedicated hardware for root scanning as proposed in Chapter 3 would require considerable changes to general-purpose processors. Moreover, applications where its advantages (temporal decoupling of root scanning, lower memory bandwidth for root scanning) outweigh its overheads are probably rather rare. Therefore, this solution may find its way into custom-built processors, but is unlikely to be included in processors that are targeted at broader markets. However, the solution shows how delays related to stack scanning can be eliminated for systems where this is necessary.

7.1.2 Copying

The copy unit presented in Chapter 6 is relatively generic, and could be integrated with arbiters that use other arbitration policies than presented in this thesis. Furthermore, processors that support virtual memory already include hardware to redirect memory accesses in order to map virtual addresses to physical memory. The overhead for integrating the copy unit can therefore expected to be low. However, integrating the copy unit with caching requires careful design.

On processors with relaxed memory models, memory accesses may be reordered, or at least appear to be reordered to a certain processor core. Processors with relaxed memory models usually provide *memory barrier* instructions to enforce a particular ordering. Depending on the precise memory model, changes in main memory may become visible only after executing such a barrier instruction. As outlined in Chapter 4, a copy unit is not compatible with such memory models, as they could render updates to the indirection pointer invisible to the application. An example would be the Java Memory Model [10], where the visibility of writes by other threads is only ensured if `synchronized` blocks or `volatile` variables are used.

However, mainstream processors usually include support for cache coherence [7, 1], which eliminate some of the issues related to memory reordering. As the logic to implement cache coherence is already present, and the copying unit itself is rather small, the integration of such a unit in a mainstream processor would cause only little overhead. On JOP, the fact that caches use a write-through policy to simplify worst-case execution time (WCET) analysis and that the arbiter serializes all memory accesses would make the implementation of cache coherence relatively simple. Future work could therefore combine the copying unit with caching on JOP.

7.1.3 Locking

Mainstream chip multi-processors usually include compare-and-swap (CAS) or similar operations rather than a hardware lock as JOP does (e.g., `cmpxchg` on Intel 64 and IA-32 architectures [7][1], or `ldrex/strex` on ARM architectures from version ARMv7 onwards [1]). CAS operations are optimistic and indicate failure only if actual contention occurred. In contrast, locks are pessimistic and therefore preclude concurrent execution. When optimizing for average-case performance, the optimistic approach is well suited, as the common case is that a CAS operation is successful. For hard real-time systems however, worst case analysis has to assume that contention occurs. In such systems, the analyzability of locks is more important than the average-case performance provided by CAS.

In the past few years, computer architecture research has started to focus on time-predictable computer architectures [4, 15]. For time-predictable chip multi-processors, it is necessary to provide synchronization primitives with analyzable timing. Given their simplicity, hardware locks would be a reasonable choice for a such a synchronization primitive. Whether hardware locks or other predictable synchronization mechanisms will be found in future processors however depends on whether time-predictability as design goal is able to establish itself in a considerable share of the processor market.

7.1.4 Handles

The handle-based object layout of the garbage collector presented in this thesis introduces overheads compared to an object layout where references directly point to the object fields. In JOP, some of these overheads are reduced by resolving the indirection in hardware. Bytecodes such as `putfield` or `getfield` are implemented in hardware and therefore enable more efficient memory accesses. Furthermore, a specialized *object cache* [6] can effectively eliminate the cost of the indirection, while still being predictable.

Another example of a processor that provided hardware support for handles is the picoJava-II processor [17]. However, despite the high performance it provided compared to other Java processors [13], it fell into disuse. One reason for picoJava-II's lack of success may have been its complexity, as it was several times larger than competing Java processors.

RISC processors do not provide memory-indirect addressing, which could be used to resolve the handle indirection in hardware. Such an addressing mode would contradict the RISC design philosophy, which favors simple addressing modes [11]. Also, such memory-indirect addressing is relatively rare in conventional programs [5]. Therefore, benefits from supporting this addressing mode, such as code size reduction and potential performance gains, are negligible. Growing popularity of modern, garbage-collected languages might however change the picture. It remains to be seen whether future mainstream architectures will include dedicated support for resolving indirections as they occur with handle-based object layouts.

It has to be noted that compiler optimizations can eliminate some of the costs for handle-based object layout as well. Keeping the value of the indirection pointer in a register would reduce the number of memory accesses for repeated accesses to the same object or array. However, such optimizations would cause coherence problems similar to core-local caches, because updates to the indirection pointer during object relocation could remain unnoticed. Such optimizations therefore have to be applied cautiously in the presence of a copy unit.

[1]The Intel 64 and IA-32 architectures also provide a `lock` prefix, which however only forces an individual instruction to be executed atomically.

7.2 Conclusions

This thesis investigated garbage collection for hard real-time systems on chip multi-processors. Two areas of particular interest were identified: root scanning and object relocation. Solutions for both of these areas were developed and their theoretical soundness was proven. The solution for root scanning has low overhead and does not require the blocking of high-priority tasks. The hardware unit for object relocation enables preemptible copying that is fully transparent to the application tasks. As the presented techniques eliminate blocking, they reduce the release jitter for high-priority tasks compared to traditional garbage collection algorithms.

Furthermore, measurements were performed with prototype implementations. The implementation presented in Chapter 6 is fully based on analyzable mechanisms, from the arbitration of individual memory accesses to the scheduling of tasks. The measurements confirmed the theoretic results that the proposed solutions enable low release jitter in the presence of garbage collection.

For systems with exceptionally stringent timing requirements, a hardware unit to support root scanning was developed. Apart from decoupling the root scanning of the application threads and the garbage collector, it also reduces the memory bandwidth for root scanning.

The thesis also investigated the effects of relaxed memory models on garbage collection, and identified where such memory models might affect the correctness of garbage collection. While reasonable solutions could be found for several critical operations of the garbage collector, efficient object relocation requires a relatively strong memory model. Furthermore, a simple cache implementation that is consistent with the Java memory model was presented.

As a step towards a comprehensive analysis of applications that use garbage collection, an analysis to compute the worst-case allocation rates of tasks was developed. In order to enable the comparison of different garbage collectors, cost models for various object layouts were presented. The analysis reuses existing infrastructure for WCET analysis, which simplified the implementation considerably.

The previous section of this chapter investigated how the findings of this thesis can be transferred to more general execution platforms. Software-based techniques are likely to be applicable to other platforms than JOP. The adoption of solutions that are based on specialized hardware may however require changes in the processor market towards time-predictable architectures.

The results of this thesis provide evidence that hard real-time garbage collection on chip multi-processors is feasible and does not interfere with high scheduling quality. As garbage collection is a means to deal with the growing complexity of hard real-time systems, this might help in building future safety-critical systems.

7.3 Future Work

The soft- and hardware developed as part of this thesis provide a prototype implementation of the presented concepts. On the one hand, this implementation requires more work before being usable in actual safety-critical systems (most importantly, the implementation of exact root scanning). On the other hand, it is necessary to further develop analysis tools to enable comprehensive system analysis with regard to timing and memory consumption.

7.3.1 Exact Root Scanning

The prototype garbage collector presented in this thesis distinguishes reference and integer values in objects and arrays. However, it does not distinguish references and integers values on the stack, and conservatively assumes that an integer value that resembles a reference is actually a reference. If references would directly point to object data, this could result in data corruption, because the garbage collector could erroneously update integer values on the stack when relocating data. As the prototype garbage collector however only updates the indirection pointer in the handle, data corruption is avoided. Still, such behavior is problematic for a real-time garbage collector. An integer value on the stack could keep objects alive that are actually not reachable any more, and the application could leak memory.

A prior version of JOP's garbage collector implemented *exact root scanning* [12]. For each position in the code, the types of values in the current stack frame are computed when building the binary image of the application. This information can be accessed at run-time when scanning the stack to distinguish reference values from integer values. Unfortunately, this implementation became incompatible with the rest of JOP as the processor evolved over time. Future work will have to update the exact root scanning implementation to the current state of JOP in order to avoid the possibility of memory leaks due to conservative root scanning. For tasks that scan their own stacks, a possible optimization would be to record only information about stack frames that may be active when stack scanning is performed.

7.3.2 Analysis

The garbage collector developed in the scope of this thesis was designed with analyzability in mind. However, future work still has to prove that the theoretical properties can be exploited by actual analysis tools. Further research is necessary to derive safe, yet tight, bounds on the memory consumption and timing of the overall run-time system and the garbage collector. In the following, some areas for potential future work are outlined.

7.3.2.1 Flow Facts

In order to compute reasonably tight bounds on the execution time of the garbage collector, the WCET analysis tool has to model the complex flow facts of the garbage collector. For example, if the analysis assumes that the garbage collector traverses the whole heap during tracing, it should not assume that it reclaims the whole heap in the same garbage collection cycle as well. While it is possible to model the respective constraints, analysis tools rarely support sufficiently expressive annotations. Extending WCET analysis tools appropriately would not only enable more precise analysis of the garbage collector, but also of the whole application.

7.3.2.2 Synchronization

While all synchronization constructs used in the garbage collector prototype have bounded timing, an analysis to compute these timing bounds is still missing. Most importantly, it would be necessary to automatically extract `synchronized` blocks and compute their WCET in order to compute the worst-case blocking time of locks. Such an analysis is of course also required for the hardware lock. Computing blocking times is not specific to garbage collection, but a necessity for rigorous analysis of applications.

7.3.2.3 Scheduling

The partitioned fixed-priority scheduling of JOP enables simple schedulability tests for cases where blocking times and scheduling overheads can be ignored. However, for realistic scenarios such simplifications are not acceptable. Unfortunately, manual analysis quickly becomes infeasably complex without such simplifications. Therefore, it is necessary to develop analyses to compute the blocking times and scheduling overheads and take them into account for schedulability analysis. While the theory behind scheduling analysis is mature, JOP's tool chain currently lacks a tool for automated schedulability analysis.

7.3.2.4 Live Memory

In order to compute the maximum period for time-based garbage collectors, it is necessary to know the allocation rates of the individual tasks and the maximum amount of live memory [16]. Chapter 5 presented an analysis to determine maximum allocation rates. An analysis to compute an upper bound for live memory is presented in [8]. Unlike other analyses, it is able to model the interactions of multiple tasks. However, it requires a relatively restricted task behavior. Future work will have to show whether this analysis can be extended to more general tasks, and whether it scales to realistically sized task sets.

7.3.3 Verification

The theory behind the garbage collector presented in this thesis is sound. The correctness for features that have been adopted from prior work can be found in the literature, e.g., the correctness of a snapshot-at-beginning write barrier is proven in [18]. Proofs for the correctness of new concepts can be found in this thesis. However, a formal verification of the actual implementation is missing. For verification, it is necessary to model the memory model, the garbage collector and its interaction with the application. Furthermore, it has to be shown that the model matches the implementation. Given the complexity of the garbage collector, verification would require considerable efforts. Thorough treatment of this topic would perhaps justify a thesis of its own, and as such must remain future work.

A reasonable starting point for verification of the garbage collector could be the work by Bøgholm et al. [2]. That paper uses UPPAAL[2] to perform schedulability analysis and targets JOP. Some of the models could probably be reused, which would reduce the effort to create a comprehensive model of the garbage collector and JOP's run-time system.

[2]`http://www.uppaal.com/`

Bibliography

[1] ARM. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, December 2011.

[2] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, JTRES '08, pages 106–114, Santa Clara, California, 2008. ACM.

[3] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[4] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 264–265, San Diego, California, 2007. ACM.

[5] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[6] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. Worst-case execution time analysis-driven object cache design. *Concurrency and Computation: Practice and Experience*, 2011. published online, to appear in print.

[7] Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals*, December 2011.

[8] Martin Kero, Paweł Pietrzak, and Johan Nordlander. Live heap space bounds for real-time systems. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 287–303, Shanghai, China, 2010. Springer-Verlag.

[9] Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James J. Hunt, Johan Olmütz Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Jan Vitek, and Andy Wellings. *Safety-Critical Java Technology Specification, Public draft*. 2011.

[10] Jeremy Manson. *The Java memory model*. PhD thesis, University of Maryland, College Park, MD, USA, 2004.

[11] David A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28:8–21, January 1985.

[12] Rasmus Pedersen and Martin Schoeberl. Exact roots for a real-time garbage collector. In *4th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '06, pages 77–84, Paris, France, 2006.

[13] Wolfgang Puffitsch. picoJava-II in an FPGA. Master's thesis, Vienna University of Technology, 2007.

[14] Martin Schoeberl. *JOP Reference Handbook*. Number ISBN 978-1438239699. 2009.

[15] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[16] Martin Schoeberl. Scheduling of hard real-time garbage collection. *Real-Time Systems*, 45(3):176–213, 2010.

[17] Sun. *picoJava-II Programmer's Reference Manual*, March 1999.

[18] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, March 1990.

Curriculum Vitae

Personal Data

Name	Wolfgang Puffitsch
Date of birth	16.5.1983
Place of birth	Wiener Neustadt
Nationality	Österreich
Address	Hegergasse 17/2/19, 1030 Wien, Österreich
email	`wpuffits@mail.tuwien.ac.at`

Employment Record

since 3/2009	Research and teaching assistant at the Institute of Computer Engineering at the Vienna University of Technology
1/2008 – 9/2010	Research assistant at the Institute of Computer Engineering for the EU-Project JEOPARD
1/2006 – 9/2006	Civilian service at the Lower Austrian Red Cross
10/2005 – 7/2007	Tutor for Systems Programming course
3/2005 – 7/2005	Tutor for Microcontroller course
10/2004 – 2/2005	Tutor for Embedded Systems Programming course
10/2002 – 7/2004	Tutor for Introduction to Programming course

Education Record

since 3/2008	Doctoral studies in Computer Science at the Vienna University of Technology (TU Vienna)
1/2008	Received academic degree *Diplom-Ingenieur* (comparable to Master of Science degree)
4/2005 – 12/2007	Master studies in Computer Engineering at the TU Vienna, graduated with distinction
5/2005	Received academic degree *Bakkalaureus der technischen Wissenschaften* (comparable to Bachelor of Science degree)
9/2001 – 3/2005	Bachelor studies in Computer Engineering at the TU Vienna, graduated with distinction

Publications

Master's Thesis

Wolfgang Puffitsch. picoJava-II in an FPGA. Master's thesis, Vienna University of Technology, 2007.

Journal Papers

1) Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 2012. to appear.

2) Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. Worst-case execution time analysis-driven object cache design. *Concurrency and Computation: Practice and Experience*, 2011. published online, to appear in print.

3) Martin Schoeberl and Wolfgang Puffitsch. Nonblocking real-time garbage collection. *ACM Transactions on Embedded Computing Systems*, 10(1):1–28, 2010.

4) Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40(6):507–542, 2010.

Workshop and Conference Papers

1) Wolfgang Puffitsch. Hard real-time garbage collection for a Java chip multi-processor. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 64–73, York, United Kingdom, 2011. ACM.

2) Benedikt Huber, Wolfgang Puffitsch, and Peter Puschner. Towards an open timing analysis platform. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Porto, Portugal, July 2011.

3) Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Reinhard Wilhelm, Armelle Bonenfant, Hugues Casse, Sven Bünte, Wolfgang Fellger, Sebastian Gepperth, Jan Gustafsson, Benedikt Huber, Nazrul Mohammad Islam, Daniel Kästner, Raimund Kirner, Laura Kovacs, Felix Krause, Marianne de Michiel, Mads Christian Olesen, Adrian Prantl, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Simon Wegener, Michael Zolda, and Jakob Zwirchmayr. WCET tool challenge 2011: Report. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Porto, Portugal, July 2011.

4) Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OASICS 18 Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik*, pages 11–21, Grenoble, France, March 2011.

5) Wolfgang Puffitsch, Benedikt Huber, and Martin Schoeberl. Worst-case analysis of heap allocations. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation (ISoLA'10)*, LNCS 6416, pages 464–478, Heraklion, Greece, 2010. Springer-Verlag.

6) Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. Wcet driven design space exploration of an object cache. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 26–35, Prague, Czech Republic, 2010. ACM.

7) Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS '09)*, LNCS 5860, pages 180–191, Newport Beach, CA, 2009. Springer-Verlag.

8) Wolfgang Puffitsch. Data caching, garbage collection, and the Java memory model. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 90–99, Madrid, Spain, 2009. ACM.

9) Wolfgang Puffitsch. Decoupled root scanning in multi-processor systems. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 91–98, Atlanta, GA, USA, 2008. ACM.

10) Martin Schoeberl and Wolfgang 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*, JTRES '08, pages 77–84, Santa Clara, California, 2008. ACM.

11) Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, JTRES '08, pages 68–76, Santa Clara, California, 2008. ACM.

12) Wolfgang Puffitsch and Martin Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, JTRES '07, pages 213–221, Vienna, Austria, 2007. ACM.

13) Martin Jankela, Wolfgang Puffitsch, and Wolfgang Huber. Towards a rapid prototyping framework for architecture exploration in embedded systems. In *Proceedings of the Second Workshop on Intelligent Solutions in Embedded Systems*, WISES '04, pages 117–127, Graz, Austria, 2004. Graz University of Technology.