A SYSTEMATIC APPROACH TO GARBAGE COLLECTION FOR REAL-TIME SYSTEMS

By

WEI FU

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2007

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of WEI FU find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGEMENT

First of all, I would like to thank my advisor Dr. Carl Hauser for his guidance throughout my PhD program. His constant support have helped me immensely in my studies here at WSU. I would also like to thank my other committee members Dr. David Bakken and Dr. Sirisha Medidi for their valuable advice and support.

I am grateful to the School of Electrical Engineering and Computer Science for its great faculty and for the financial support it has offered me. Especially, my thanks go to chair of graduate studies Dr. Ali Saberi and graduate advisor Ruby Young. I have been a teaching assistant since I came to Washington State University. I thank all the instructors and students that have supported my work. I would also like to thank the College of Electrical Engineering and Computer Science and the Graduate School for awarding me “Outstanding Teaching Assistant” for my teaching assistant work at WSU.

Special thanks also go to my husband Chad Zanonie for his support and help during my studies. Also, my accomplishment would not have been possible without the support from my parents XiCai Fu and QingXian Zeng, my husband’s parents Chris Zanonie and Cathy Zanonie for their constant encouragement and for always being there for us. Also thanks go out to all the acquaintances and friends that I have had during my time in Pullman. They all made the stay so much more enjoyable.

A SYSTEMATIC APPROACH TO GARBAGE COLLECTION FOR REAL-TIME SYSTEMS

Abstract

by Wei Fu, Ph.D.
Washington State University
August 2007

Chair: Carl Hauser

Garbage collection (GC) is becoming increasingly prevalent within the programming landscape. The use of garbage collection in real-time systems has also gained increasing attention given the major advantages in productivity and safety. Unfortunately, the progress in GC for mainstream applications and languages does not solve the problems of memory management for embedded real-time systems. The fundamental problem is that conventional GC techniques do not provide the timing and memory use predictability that are basic real-time program requirements.

This dissertation presents a systematic approach to real-time garbage collection (RTGC) for real-time systems. Given a real-time system with a bounded workload, the goal is to calibrate the control of RTGC in a systematic and automatic way. We use an approach that separately models (1) the performance of collector operations and (2) the garbage collection load offered by real-time tasks. It allows prediction of system behavior from knowledge of component behavior and environment specifications. This knowledge provides designers tools to use GC in a predictable way.

Three major topics are presented: a system model that outlines the systematic RTGC approach, a GC cost model that guarantees garbage collection activities' worst-case execution time (WCET), and GC-integrated scheduling and schedulability analysis mechanisms that compute feasible GC execution parameters satisfying real-time timing requirements.

A modified Boehm-Demers-Weiser (BDW) GC implemented in Mono 1.1.16, [96], demonstrates the use of the RTGC approach with a status router real-time application.

ATTRIBUTION

- Wei Fu and Carl Hauser, “Modeling real-time garbage collection cost”, In the 13th IEEE International Conference on Embedded and Real-time Computing Systems and Applications, August, 21-24, 2007
- Wei Fu and Carl Hauser, “A real-time garbage collection framework for embedded systems”, In Proceedings of the 9th International workshop on Software and Compilers for Embedded Systems, Pages 20-26, October, 2005
- Wei Fu and Carl Hauser, “A detailed cost model for real-time garbage collection”, IEEE Transactions on Software Engineering (submitted, July, 2007)

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Garbage collection (GC) for memory management (dynamic memory management) is an increasingly prevalent feature of modern programming languages. As programs grow in size and complexity it becomes more and more difficult to correctly manage memory using manual techniques. Automatic garbage collection significantly reduces memory leaks and dangling pointers. The incorporation of GC in recent mainstream programming languages such as Java and C# heralds increasing acceptance of GC as a part of the programming landscape. Unfortunately, the progress in GC for mainstream applications and languages does not solve the problems of memory management for embedded real-time systems. The fundamental problem is that conventional GC techniques do not provide the timing and memory use predictability that are standard real-time program requirements, [79].

In many of today's real-time systems GC is not needed; they use micro-controllers with little memory and few cycles to spare. Manual, even static, memory management is a good solution. However, the trend in embedded systems is toward use of much faster, larger micro-controllers. Many have 32-bit addressing and hardware memory management capabilities. Prices for these devices are dropping rapidly allowing their use even in low-cost products. Micro-controllers like these invite implementation of much greater functionality in embedded system. That functionality is accompanied by additional complexity in the software, outstripping the ability of implementors to implement memory management using manual methods.

GC is particularly valuable when programs involve concurrency or exception handling: the

difficulty of maintaining ownership invariants when using manual storage management in these programs often leads programmers to copy data structures which in turn causes poor performance, [17, 18, 60]. Real-time embedded systems often involve complex concurrent interactions with physical processes. Thus, embedded systems could benefit from GC if mechanisms and analysis can be developed to support their real-time requirements.

This dissertation presents a systematic approach toward the automatic GC integration in the real-time embedded systems. The approach to real-time garbage collection (RTGC) is achieved by (1) proposing a *RTGC framework*, which models the RTGC methods, and a *system model*, which assembles the GC-enabled real-time systems; and (2) studying and characterizing interactions between GC and the rest of the real-time system in the system model. This chapter will briefly overview relevant computer science areas to provide a context to enable the reader to better understand the research contributions of this dissertation. First, garbage collection concepts are presented, followed by the real-time systems. Then, there is the discussion of the problem statement of applying garbage collection in real-time systems and the thesis statement which the research validates. This chapter ends with an outline of the organization of the remainder of this dissertation.

1.1 Garbage Collection

The oldest form of memory management is *static memory management*, [59]: a method requiring space for variables and data structures to be allocated statically by the programmer or compiler. The limitations are severe when there is the need to build dynamic input-driven data structures. *Dynamic memory management*, [62], overcomes the limitations by making it possible to allocate memory at any time in the program. However, this requires dynamically-allocated memory to be managed at run-time. It can be done *manually*, by explicitly inserting instructions to allocate and

deallocate a certain memory area (as `free` and `delete` in C/C++) or *automatically* by the run-time system. The latter option is *automatic memory management* which is also called *garbage collection* (GC). Examples of early programming languages that use garbage collection are Lisp, [75] and Simula [31]. Today, garbage collection is an important part of the memory management system of many modern programming languages, *imperative*, such as Java, [32], and C#, [76], as well as *declarative*, such as Haskell, [57], ML, [77]. Although GC significantly reduces memory leaks and dangling pointers, it comes with its own costs in terms of both time and space. These issues will be extensively addressed in chapter 2.

1.2 Real-time embedded systems

For systems, typically those that interact with an external environment, the program's correctness is not only producing the right output, but also its ability to produce results before a given time, the *deadline*. A system is said to be *real-time* if the correctness of an operation depends not only upon the logical correctness of the operation but also upon the time at which it is performed, [88].

Furthermore, real-time systems can be divided into *soft* real-time and *hard* real-time, [88]. A failure in a soft real-time system causes a temporary decrease in the qualify of service and does not cause the whole system to fail, examples are audio/video systems, where a missed deadlines causes minor pause in the stream but the playback still continues. In a hard real-time system, missing a deadline may cause the whole system to fail. Examples are the engine control systems where a delayed signal may cause engine failure or even engine damage. Medical systems such as heart pacemarkers and industrial process controllers are other examples of hard real-time embedded systems.

In order to guarantee real-time behavior, more attention must be given to the *worst-case* performance than the *average-case* performance. A key attribute of proper real-time systems is predictability. It is necessary to perform the worst-case analysis on execution time and memory usage,

thus an *a priori schedulability* analysis can be performed. Schedulability analysis is a theoretical analysis aimed at determining whether a given set of tasks can be scheduled to meet all deadlines under a given scheduling model.

1.3 Problem Statement

The fundamental problem is that classical garbage collection approaches cause unpredictable overhead on the execution of programs because GC scheduling interacts with real-time tasks (*mutators*) timing-properties, [60]. A major goal of garbage collection in a real-time system is to ensure that execution of the garbage collector does not prevent the mutators from meeting their deadlines.

To be more specific, real-time tasks require guarantee that memory requests from them can be served at any time and that the upper bounds on the time spent serving such requests be small enough to satisfy all their deadlines. Therefore, GC is an attractive option for complex embedded systems *if* real-time constraints can be satisfied.

In 1978, Baker, [8], developed a GC that caused only bounded delays in the execution of a list-processing mutator. To deal with the unbounded-pauses problem Baker introduced *incremental garbage collection*: the execution of GC work is interleaved at a fine grain throughout the execution of the mutator. Baker's algorithm has been the basis of much later works on real-time garbage collection. The algorithm performs a small increment of GC work each time a mutator performs an allocation. The algorithm is real-time in the sense that it guarantees upper bounds on the cost of every pointer operation and it is incremental. In general, bounding the delay of each GC increment is not sufficient to ensure that the program as a whole can meet its timing requirements. If GC increments are executed too frequently deadlines may still be missed. Conversely, if GC increments are too infrequent memory may not be available when needed by a mutator. Recent real-time garbage collection (RTGC) research [7, 30, 54, 61, 83] has looked at this problem by considering the scheduling of GC increments in relation to the work of mutators.

The existing RTGC research has produced techniques for breaking GC activity into small increments that are scheduled to meet the real-time deadlines and memory requirements of the mutators that perform the real work of a system. These techniques rely on predictions of mutator behavior obtained either from an oracular programmer or from observed performance of the mutator tasks. The resulting GC algorithms can be empirically adjusted (either manually or automatically) so that deadlines are observed to be met.

Real-time garbage collectors can be built on any of the classical garbage collection techniques and variants: reference counting, mark-and-sweep collection, compacting collection, copying collection, generational, concurrent and distributed collection, with additional mechanisms for *incrementality, measurability,* and *schedulability*, which will be extensively discussed in chapter 2. Details of the adaptation to real-time of course varies amongst the approaches.

As yet, research in RTGC has not produced a systematic GC approach and corresponding methodology for analyzing system behavior that allows requirements. From a practical engineering perspective, developing a systematic scheme to facilitate the automation of integrating GC in real-time embedded systems is promising. Such an approach need have the following attributes:

- First, the RTGC *incrementality, measurability* and *schedulability* attributes need be preserved;

- Second, the approach should provide a uniform way for users to express and model real-time applications and GC implementations;

- Third, the approach needs capture interactions among real-time tasks and garbage collector. It will compute the feasible GC execution parameters that is directly applicable to the GC-enabled system at run-time. The computation depends on the knowledge of the real-time application with its workload and the GC implementation.

Ideally, given a real-time application and workload, the RTGC approach automates the computation of the valid GC execution parameters to preserve the real-time constraints of the GC enabled real-time system.

1.4 Thesis Statement

This dissertation proposes the systematic approach to the application of RTGC into the real-time development realm. An original inspiration of the thesis's approach is the Giotto system, [55]. The fundamental requirement of a real-time system is, of course, that its deadlines are met. Until recently real-time system implementation involved detailed study of code paths to ensure that every deadline was met for all inputs to the system,[87]. Details of a system's timing requirements were often reflected throughout the implementation and software designs were platform-specific. The Giotto system introduced a new approach to constructing real-time systems. Giotto's key contribution is that it supports "the automation of real-time control system design by strictly separating platform-independent functionality and timing concerns from platform-dependent scheduling and communication issues", [55]. Giotto's tools analyze the worst-case execution time of components when used on particular hardware platforms then schedule their execution so that all timing requirements are met. One key to Giotto's success in automated scheduling of components was its development of a synchronous timing model that simplifies calculation of the timing characteristics of composed components.

Given a system with the bounded workload, the overarching goal of the dissertation work is to similarly calibrate the control of RTGC in a systematic and automatic way. The validation of the following statement will be investigated throughout the thesis:

> ***Thesis:*** Decomposing the model of (1) the performance of collector operations and (2) the garbage collection load offered by a real-time task set allows prediction of system behavior from knowledge of component behavior and environment specifications. This approach provides the models needed to engineer GC into real-time applications.

The dissertation studies three major topics to achieve having RTGC as a feasible option in the real-time embedded system realm.

- **RTGC framework and system model** outlines the systematic RTGC approach, where the system model provides a uniform platform for characterizing and examining the GC-enabled real-time systems and the RTGC framework attempts to encompass a variety of RTGC techniques and to provide uniform language for describing the parameters that are used to configure them.

  Describing the interactions between mutator's workloads and the control of real-time garbage collectors is a step along the road to an engineering approach to garbage collection in real-time systems. The proposals of the system model together with the RTGC framework perceive the interactions as well.

- **GC cost model (GCCM)** estimates the GC-related cost, for instance, the worst-case GC cycle time, in terms of (1) the performance of collector operations and (2) the garbage collection load offered by a real-time task. Separation of these aspects allows prediction of system behavior from knowledge of component behavior and environmental specifications. The worst-case cost analysis is necessary by real-time scheduling and schedulability analysis to achieve and validate system predictability.

- **GC integrated scheduling mechanisms (SCHED)** computes the GC execution parameters in order to preserve the real-time systems predictability. The GC scheduling parameters are specified using either the explicitly controllable *GC incrementality* parameters or a real-time task's scheduling parameters, such as task period. The GC incrementality determines the size of GC increments, which characterizes the interleaving pattern among the mutators and GC.

  Meeting real-time timing requirements has traditionally relied on proper scheduling and

schedulability analysis techniques, [70, 4, 6, 15, 33], upon the underlying task and scheduling models. With the existing scheduling techniques, the `SCHED` takes the *GCCM* estimates and outputs the GC scheduling parameters.

As an illustration of the systematic RTGC approach, a typical usage consists of the steps:

1. Given a real-time system with workload, define it using the *system model*;
2. Model the underlying GC implementation with the *RTGC framework*;
3. Obtain the required space and timing metrics of the real-time application and its workload that should be known *a priori* in the system model, for instance, the worst-case task execution time C_i *etc.*;
4. Calculate the GC-related cost with the *GCCM*, for instance, one GC cost is the worst-case GC cycle execution time, C_{GC};
5. Calculate GC scheduling parameters upon a chosen scheduling policy in the scope of *SCHED*. The set of GC scheduling parameters passes the schedulability analysis are feasible to use;
6. Deploy GC with the calculated GC scheduling parameters in the real-time application at runtime.

1.5 Dissertation Organization

The rest of the thesis is organized:

Chapter 2: Literature Survey describes the fundamental concepts of the areas of real-time computing and memory management techniques. It also presents and compares related RTGC work.

Chapter 3: RTGC framework and system model present the common language for describing the GC included real-time systems using *a system model* and a broad class of real-time collectors using *a RTGC framework*. An RTGC configuration with GC incrementality in the framework provides a target for automating the task of configuring automatic memory management for a real-time system and its workload. A real-time system with a copying GC is modeled with the system model as an example.

Chapter 4: Modeling RTGC cost demonstrates a detailed GC cost model for incremental mark-and-sweep GC exemplified by a modified Boehm-Demers-Weiser (BDW) collector [21]. The GC implementation is modeled using the RTGC framework. The GC cost model computes the WCET for garbage collection in terms of (1) the performance of collector operations and (2) the garbage collection load offered by a real-time task. The model incorporates the cost of write barriers that are needed to support incremental GC. A *pessimism* metric assesses the degree to which the model's predictions exceed the measured worst-case cost in particular instances. The coefficients defined in the GC cost model have been found to be stable when measured across a variety of applications in experiments. The stability of the measurements verifies the fine granularity used in the GC cost model.

Chapter 5: Schedulability mechanisms with RTGC incrementality calculates the RTGC configuration parameters defined in the RTGC framework with the specific scheduling policy. The controllable RTGC incrementality introduced with the RTGC framework is implemented in the modified RTGC in Mono, [96]. Two scheduling mechanisms are studied: (1) time-based GC scheduling ξ_T with *C-space* scheduling policy, and (2) work-based GC scheduling ξ_W with *MMU* scheduling requirement.

Chapter 6: Experimental results with Status Router demonstrates a comprehensive illustration of the systematic RTGC approach. The GridStat status router, a multi-casting packet

forwarder with real-time processing requirement, [50], is re-implemented. The status router application is modeled using real-time task model introduced in the system model in chapter 2. The status router's GC costs are estimated in the GC cost model presented in chapter 4. Both of the scheduling mechanisms demonstrated in chapter 5 are applied to the status router at runtime. Finally, experiments showed the validity of the calculated RTGC configurations. Performances are evaluated by comparing a set of application metrics for various GC scheduling settings.

Chapter 7: Conclusion summarizes the contributions of the thesis and discusses the future work.

CHAPTER 2

LITERATURE SURVEY

This chapter presents the fundamental concepts of real-time systems, memory management mechanisms and classical garbage collection algorithms. It also summarizes the RTGC properties and algorithms. A discussion of the related RTGC work is followed. It concludes with a comparison among various existing RTGC approaches.

2.1 Time and space concerns for real-time programs

The integration of GC complicates the picture of real-time programs, since although GC significantly reduces memory leaks and dangling pointers, it comes with both time and space cost, which elicits two major concerns.

- Hard real-time systems are subject to timing-constraints and must perform real-time tasks that satisfy both timing properties and logical criteria for correctness. Reliability and predictability are key attributes for hard real-time systems. Failure in a safety critical real-time system can lead to considerable damage. Examples of timing properties and constraints include deadlines, the execution frequency of tasks, and external event recognition based on time of occurrence, [87]. Therefore, over the years researchers have developed a range of schedulability analysis and time reasoning techniques, which are generally based on worst case execution time (WCET), [66, 93], of each real-time task in the system.

  As a principled, tool-supported design methodology, *Giotto*, [55], supports the automation

of real-time control system design by strictly separating platform-independent functionality and timing concerns from platform-dependent scheduling and communication issues by performing implicit schedulability analysis with known WCETs.

- Memory management is another focus for real-time systems. Despite rapid growth, little attention has been given to real-time memory management compared to real-time processes scheduling analysis. While memory allocation falls into the category of resource allocation, it differs from CPU allocation in a major way since preemption is not possible due to embedded system lacking secondary storage, [83]. Therefore, memory safety in terms of both *access* safety and *availability* safety must be addressed as complementary aspects for real-time systems.

  Access safety is the concern in the possibility of dangling references that allow program to access unsafe memory that has been deallocated; and availability safety is the concern in the possibility that program might attempt to allocate more memory than current run-time platform can afford. In fact, these two memory safety concerns are addressed throughout the development of memory management and elicit many research attempts with the fact that the former concern gains more attention than the latter one. To illustrate the availability memory safety property, given t_{any}, an arbitrary time point during real-time tasks execution, there are three metrics (H_f, H_l, H_r). They are the amount of free memory, live memory and memory request at time t_{any}. The truth of the *invariant*: $(H_f \geq H_r) \wedge (H_f + H_l) \leq H$ guards the memory availability safety, where H is total amount of memory.

A range of memory management approaches have been proposed including predictable *static memory allocation*, *manual memory management* with explicit allocation and deallocation, *garbage collection* and *region-based memory allocation*, [56]. These approaches are listed in table 2.1 and

Table 2.1: Memory management approaches comparison

	Heap-Allocated	Automatic	Predictable	Flexible
Static Memory management	×	×	✓	×
Manual Memory management	✓	×	✓	✓
Classical Garbage Collector	✓	✓	×	✓
Region-based Memory management	–	✓	✓	–
Real-time Garbage Collector	✓	✓	✓	✓

the attributes of *heap-allocated*, *automatic*, *predictable* and *flexible* are compared.[1] Refactoring codes with flexible memory management mechanisms does not need the users to explicitly re-implement the memory allocation-deallocations in programs. The *flexible* attribute captures such memory management feature.

The region-based memory management have been implemented to support the real-time Java specification (RTSJ), [12]. Real-time Java builds *scoped memory* that each scoped memory conceptually contains a preallocated region of memory that threads can enter and exit. Once a thread enters a scoped memory, it can allocate objects from that memory. Each allocation takes a predictable amount of time. When the thread exits the scoped memory, the implementation deallocates all objects allocated in the scoped memory without garbage collection. The whole scoped memory can only be garbage collected when all of its associated objects are no longer in use. Region-based memory management is predictable, however, the unused memory cannot be recycled until the whole region becomes garbage. The region-based memory management has the flexibility in between static and dynamic memory management mechanisms.

2.2 Classical garbage collection algorithms

Classical garbage collection algorithms include reference counting, mark-and-sweep, mark-and-compact collection, copying collection, generational collection, incremental concurrent collection, distributed collection and variants of all the above, [60]. Garbage collection (GC) has its overhead in terms of time and space. A 10% overhead of the overall execution time is reasonable for a

[1](✓ means 'true', × represents 'false' and – means 'partially true').

well-implemented system with GC, [94].

Garbage collection automatically reclaims the space occupied by data objects that the running program can never access again, which ensures memory access safety property. The basic functional abstraction of a garbage collector consists of two phases:

1. **Garbage detection**: distinguish live objects from garbage caused by deallocation;

2. **Garbage reclamation**: reclaim garbage while scanning the heap memory, compacting might take place here to avoid fragmentation; finally the reclaimed memory are free to use again.

Various garbage collections differ in the way how garbage is detected, which can be *reference counting, marking or copying*. An outline of basic garbage collection techniques are given. Detail illustration of classical garbage collection techniques can be found in [60, 94].

2.2.1 Reference counting (RC)

Reference counting associates each object with a reference count. The idea is simple; count the number of references to every object and reclaim the object if its reference count reaches zero. A reference counting memory handler has two main operations: *Obj.ref.increment()* and *Obj.ref.decrement()*. Reclamation might be triggered by *Obj.ref.decrement()* when the reference counter is zero. *Obj.ref.increment()* operation takes constant time, while *Obj.ref.decrement()* is potentially recursive, which imposes difficulties in time relevant analysis, such as WCET mentioned in chapter 1.

By using deferred reference counting, [92], *Obj.ref.decrement()* operation is made constant in time. Deferred reference counting takes advantage of the observation that “the majority of pointer stores are made into local variables”, [60] and treats operations on local variables and stack-allocated compiler temporary variables specially: there is no reference counting when these references are modified. Objects are no longer be reclaimed as soon as their reference count drops to zero since they might still be reachable from local or temporary variables. Instead, objects with

zero reference count are added to *zero count* table (ZCT) by `delete`. Periodically the ZCT is reconciled to remove and collect garbage. The reconciliation works in three phase: mark the stack, reclaim garbage that are the unmarked objects with entries in ZCT and finally turn all marked objects unmarked.

The most significant advantage of reference counting is its simplicity and fine-grainedness.[2] On the other hand, basic reference counting cannot deal with cyclic data objects, which may result floating garbage. Secondly, since garbage reclamation keeps data in fixed place without back-and-forth move, fragmentation will occur. One way to handle cyclic data is to use other garbage collection techniques when floating garbage accumulates beyond pre-defined threshold.

2.2.2 Mark-and-sweep and mark-and-compact collection (MS and MC)

Mark-and-sweep collection and mark-and-compact collection are named in accordance to the two-phase abstraction with the latter relocates objects by compaction to remedy fragmentation:

1. Mark phase(Garbage detection): traverse the graph of pointer relationships (usually either depth-first or breadth-first) starting at the root set and mark reachable objects in various ways;

2. Sweep phase(Garbage reclamation): exhaustively examine the whole heap to find all of the unmarked objects(garbage) and reclaim their space for reuse;

3. Compact phase(Garbage reclamation by compacting): most of the live objects marked during mark-phase are moved until all of the live objects are contiguous, which leaves the rest of memory as a single contiguous free space.

[2]Fine-grainedness here means that only a few instructions need locked in a multi-thread context.

Mark-and-sweep collection's major deficiency is that the cost of collection is proportional to the heap size, which leaves little space to improve efficiency. Mark-and-compact remedies fragmentation problem but might perform significantly slower than mark-and-sweep if a large percentage of data survives to be compacted, since several computation passes need to operate on the live data to get them compacted. There are many efforts to improve mark-and-compact efficiency by reducing compaction overhead, and one of them is a two-pointer algorithm that scans the heap in two directions during compaction.

2.2.3 Copying collection (CC)

Copying collection divides the heap into semi-spaces: old semi-space *Fromspace* and new semi-space *Tospace*, and copies reachable objects from *Fromspace* to *Tospace* during collection and reverses the role of two semi-spaces at the end of a complete copying cycle.

Copying collection is similar to mark-and-compact in the way that live objects are relocated in a contiguous area after a complete copy. Copying collection integrates data mark and compaction into a single copying process with the help of semi-spaces so that most objects need only be traversed once rather than several passes in mark-and-compact collection. Copying collection cost is proportional to the size of reachable data. In copying collection, heap is divided into two spaces, and during normal program execution, only one semi-space is in use. This scheme gains processing efficiency by sacrificing space. Examples of copying collection include Cheney's algorithm, [24], a stop-and-copy collector using semi-space, Bakers incremental copying collector, [8].

2.2.4 Generational collection (GeC)

Generational collection is based on the Generational Hypothesis: *Most objects die young*; and this gives the insight behind generational garbage collection that storage reclamation can be made more efficient and less obtrusive by concentrating effort on reclaiming those objects most likely to become garbage, such as young objects.

Generation garbage collection is a technique suggested by Lieberman and Hewitt, [65], in the early 1980's that divides a program's heap into regions containing objects of different ages. The younger an area is, the more frequent collection is performed. And objects that survive a small number of collections are moved to a less frequently collected area. Thus, unnecessary repeated copying is avoided by only copying the few survivors.

Two observations are worth mentioning before closing this section. First, all garbage collection approaches encounter similar time-space trade-offs. Performing less garbage collection reduces overhead but leaves more unreclaimed garbage. Second, various garbage collection techniques may also differ in the way how things are approximated, such as true liveness because of their undecidable property. In reference counting, an object not pointed to by any variable or other object is clearly garbage. However, the reciprocal is not always true. A conservative tracing garbage collector is unable to distinguish between pointers and non-pointers so it treats any word that contains a value that represents a valid address as a pointer. Baker's incremental non-copying algorithm, [9], assumes objects allocated during collection to be live as a liveness approximation, even if they die before the collection is completed. The approximation closeness effects garbage collection performance and complicates its implementation. Incremental tracing collectors are discussed in next section.

2.3 Real-time Garbage Collection: Properties and Algorithms

Real-time garbage collection is built on top of techniques of classical GC work with added characteristics of *incrementality, schedulability and measurability*[3].

- **Incremental** For hard real-time systems, garbage collection cannot be executed as atomically while the real-time program sleeps. Fine-grained incremental garbage collection is

[3]Since fragmentation conflicts with predictability of free memory and most classical garbage collections handle non-fragmentation, it is assumed to be implicitly implemented within classical garbage collection, thus it is not treated as a special property for RTGC. Although, due to high overhead for non-fragmentation, some RTGCs are based on fragment-tolerant collection and leave fragmentation issues for RTGC.

necessary since it interleaves small units of garbage collection with small units of real-time tasks execution. This elicits the need for work, time and hybrid based interleaving controls.

- **Schedulable** For hard real-time systems, RTGC needs to be predictable to fulfill schedulability requirements. This increases both RTGC algorithm complexity and overhead. Real-time task scheduling analysis must also be aware of RTGC existence. Conversely, the execution of RTGC must perform enough work to provide available memory for mutators. To ensure correct real-time behavior of hard real-time program, the program must be proven to be valid before it can be used in a real-time system. This elicits the need for schedulability analysis for both real-time workload and RTGC.

- **Measurable** *A GC cycle* is defined as the set of all activities required to identify and reclaim garbage. For instance, a GC cycle in mark-and-sweep consists of root scanning, pointer traversal and sweeping. *A GC increment* is partial work within a GC cycle that contributes to the whole cycle. Incrementality and schedulability attributes describe when/how GC work is triggered and how to schedule real-time tasks and GC work together. In order to do this, GC measurability must express how much work is performed each increment within a GC cycle. GC work is measured in terms of various metrics that will be explained in chapter 3. Thus, the GC cycle together with GC increments are measured by chosen metrics to represent their progress.

Proposals to make incremental garbage collection more suitable for real-time systems have been submitted for many years. The incremental copying algorithm, Baker's algorithm, [8], has been a basis for RTGC work since its inception. Baker's algorithm is a fine-grained work-based approach where each allocation request triggers the GC work by suspending real-time tasks for a short period of time using a read barrier. It is claimed to be a real-time algorithm since it is incremental and guarantees low upper bounds on the cost of every pointer operation. A potential

problem occurs when frequent GC invocations result in an unacceptable mutator utilization[4] due to bursty pauses when burst allocations are carried out. Brook's algorithm, [22], improves Baker's algorithm using a write barrier and makes the overhead smaller since the pointer writes are much less frequent than pointer reads. In Baker's more recent algorithm, [9], cyclic double-linked lists are used to implement a non-copying version of his original incremental garbage collection.

2.3.1 Barrier techniques

In Wilson's survey paper, [94], an incremental mark-sweep collector poses a multiple-readers, single-writer coherence problem since both the mutator and the collector read pointer fields but only the mutator can modify them. Incremental copying collectors provide an example of a multiple-readers, multiple-writers problem since the collector also writes pointer fields when it moves objects, [60].

Incremental GC Conservatism

It is necessary to ensure that the mutator's or GC's view of "data connectivity" is consistent, *i.e.*, it does not attempt to access objects through obsolete references. It is not necessary for the mutator and the collector to share an identical view of the computation graph. The consistency requirement can be relaxed to allow the collector to work with a conservative approximation of the graph of active objects. While the collector must treat any reachable object as active, the semantics of garbage collection are preserved even when the collector treats some objects that are unreachable as if they were still visible to the mutator (alive).

In general, the more consistency requirements are relaxed, the more conservative the collector's view of the reachability graph becomes, thus, the more floating garbage accumulates. Floating garbage fragments the heap, increases the effective residency of programs and puts more pressure on the garbage collector.

[4] A mutator utilization is defined as the percentage of CPU time contributing to the execution of mutators, thus $U_M = CPU_M/CPU_{Total}$.

General incremental GC attributes are: (1) the degree of GC conservatism; (2) bounds on the pauses of GC increments and (3) bounds on uninterruptible processing sections, for instance, processing the root set or checking for termination of a garbage collection cycle as atomic actions. The GC conservatism is determined by the barrier techniques that GC incorporates to support incrementality.

Tricolor marking

Tricolor marking, [36], was originally introduced by Dijkstra to describe incremental garbage collection, [60]. The abstraction is restated here, where nodes in the heap are painted one of the three colors:

1. **Black** indicates that a node and its immediate descendants have been visited, which means the garbage collector has processed black nodes and need not visit them again.

2. **Grey** indicates that the node must be visited by the collector. There are two cases resulting a grey node: (1) the node has been visited by the collector but its constituent pointers have not been scanned; (2) the node connectivity to the rest of the memory allocation graph has been altered by the mutator behind the collector's back.

3. **White** indicates nodes unvisited by the garbage collector at the end of the tracing phase, which means they are unreachable (garbage).

Barrier techniques aim at guaranteeing that the collector will not miss any reachable objects. Two tricolor invariants are the basis for various barrier algorithms:

Invariant 1 (Inv_1): No destroyed original reference to the white object;

Invariant 2 (Inv_2): No black-white pointers.

In [81], these two invariants are named *weak* and *strong* tricolor invariants respectively.[5]. Barriers are used to intercept the mutator's accesses to objects and they can be divided into two categories – *read-barrier* and *Write-barrier*:

- **Read-barrier (trapping reads)**: deals with object access and ensures that the mutator never sees a white object. Whenever it attempts to access a white object, the object is immediately visited by the collector. This is used in copying collectors so that mutator can only see objects in *Tospace*. Examples are:

  1. *Baker's read-barrier*: preserve Inv_2. Mutator cannot see white objects, until they are copied and marked grey;
  2. *Apple-Ellis-Li's read-barrier*: mutator cannot see white and gray objects, until they are scanned, copied and marked black.

- **Write-barrier (trapping write)**: deals with reference updates and records where the mutator writes black to white pointers so that the collector can visit or revisit the nodes in question. At least one of the following barrier invariants need be true.

  Write barrier methods are further distinguished as either *snapshot-at-the-beginning*, [99], or *incremental-update*, [36, 53]. Each of them preserves Inv_1 and Inv_2 respectively. Examples are:

  1. Write barriers in mark-and-sweep collectors:
     - Yuasa's snapshot-at-the-beginning write-barrier: Preserves Inv_1. No objects that become garbage in one garbage collection cycle can be reclaimed in that cycle that they must wait until the next.

[5]By calling invariant strong, it means it implies the weak one.

- Dijkstra's incremental-update write-barrier: Preserves Inv_2 and adopts the most conservative of the incremental-update coloring strategies: white nodes are shaded grey when a reference is created, regardless of the color of the parent node. The strategy is usually referred as "the barrier's wave-front is advanced".
- Steel's incremental-update write-barrier: Preserves Inv_2 and retreats the grey wave-front rather than advance it, making it less conservative than Dijkstra's. The strategy is usually refered as "the barrier's wave-front is pushed back".

2. Write barriers in copying collector:
   - Brook's forwarding pointer write-barrier: preserve Inv_2 where mutator can see objects in both *Fromspace* and *Tospace*.

Impact of barriers on incremental GC

The total overhead of an incremental or concurrent algorithm is determined by the conservatism of the barrier and how it affects the collector's view of the reachability of the graph. The barriers and increment GC are correlated in the following aspects:

- The time and space costs of the barrier depend on its frequency and how it is implemented (color bits or a mark stack);
- The barrier implementation, such as whether a single node is colored or an entire page of references is scanned, affects the barrier pause time resulting in the incremental GC;
- How a collection cycle is initiated and terminated are significant factors for the barrier mechanisms as well.

2.4 Discussions of existing RTGC approaches

As in chapter 2.3, classical garbage collection algorithms enhanced with mechanisms for *incrementality*, *measurability* and *schedulability* are necessary to achieve real-time garbage collection.

In this thesis, the work that eventually automates the integration of GC into real-time systems consists of the concept of RTGC framework, GC cost model and the GC scheduling mechanisms. The proposal of RTGC framework is original in this thesis, therefore, existing RTGC approaches are presented and compared from two perspectives which are the cost model and scheduling mechanisms.

2.4.1 Existing work on modeling GC cost

Deeply embedded real-time applications are pervasive in the automotive, industrial automation and telecom industries. Today's embedded systems have much faster, larger micro-controllers with larger memory space, much greater functionality and more complex software implementations. This trend outstrips the ability of implementors to implement memory management using static or manual methods. At the same time, the incorporation of GC in recent mainstream programming languages such as Java and C# suggests increasing acceptance of GC as a part of the programming landscape [45]. GC is an attractive option for embedded systems *if* real-time constraints can be satisfied.

Recent research in real-time garbage collection (RTGC) has produced techniques for breaking GC activity into small increments that are scheduled so that mutator tasks (mutators) can be shown to meet their real-time deadlines while never exhausting memory. [7, 51, 54, 61, 83]. These techniques rely on predictions of mutator behavior obtained either from an oracular programmer or from observed performance of the mutator tasks. The resulting GC algorithms can be empirically adjusted (either manually or automatically) so that deadlines are observed to be met [35]. As yet, research in RTGC has not produced a GC approach and corresponding methodology for analyzing system behavior that allows designers to *prove* that a system meets its real-time requirements. It is important to ultimately understand and model the workload imposed on the GC by the mutators–which in turn depends on the mutators' workloads.

In real-time system development, Worst-case Execution Time (WCET) analysis provides *a*

priori an execution cost model for a piece of code. Based on WCET estimates of the components, performance of the whole system can be predicted including scheduling and schedulability analysis to determine whether performance goals are met for periodic tasks and to check that interrupts have sufficiently short reaction times and for many other purposes.

Integrating GC with a real-time system development complicates the situation because enough time must be allotted for GC-related activities and time spent in GC must be scheduled so as not to interfere with mutators meeting their deadlines. The amount of time required depends not only on the algorithms used in the GC but also on the collection load which in turn depends on the implementation of the mutator tasks (which do the useful work in the system) and the load imposed on them by the environment. To perform scheduling and schedulability analysis in a system using GC, worst-case execution times (WCET) of both mutator and GC tasks are needed. There are many existing techniques to model and analyze real-time tasks' WCETs either statically or dynamically [66, 84, 93] and these may be used for the mutator tasks. For garbage collection activities, it is crucial to have a complete and accurate GC cost model to estimate any GC related time costs based on the GC implementation in use that accounts for the allocation behavior of the mutator tasks. The cost of GC should be predictable based on (1) the performance of collector operations and (2) the garbage collection load offered by a real-time task. Separation of these aspects is a step along the road to an engineering approach to garbage collection in real-time systems, allowing prediction of system behavior from knowledge of component behavior and environmental specifications.

Work on estimating GC costs for real-time settings (RTGC) has been occurring since the early 1990's. An early model by Bengtsson [14] uses fine-grained accounting for costs, often captures costs with great precision. Bengtsson's GC cost model is based on the cost of primitive operations such as moving one memory cell, performing a flip, *etc.*, for the compacting collector which he worked with. The worst-case GC cycle cost is calculated by summing up the costs of all the individual steps where each step's cost is estimated over its primitive operation costs. This need be

performed with knowledge of the worst-case state of an application's data structures. The difficulty with this model is that it is hard to relate application and workload characteristics to the number of primitive operations during system design.

Other approaches, for example Henriksson [54] and Kim [61], attempt to model GC costs at the granularity of significant GC execution stages such as rootscan, barrier processing, heap marking, *etc.*, without decomposing the work of each stage. These coarse-granularity GC cost estimates suffer in accuracy because they are insensitive to differences in application behavior that affect different aspects of the algorithms that make up each sub-stage.

For instance, if a single constant coefficient δ_M is associated with the GC heap marking phase and the GC heap marking cost is calculated by $\delta_M \cdot L$ (L is the maximum live heap memory) as in [61], the costs of detailed steps such as reference checking, heap scan, reference marking *etc.* will not be differentiated; since the coefficient depends not only on the GC implementation but also on characteristics of the application such as the reference density in live memory. To give accurate GC cost predication, the coefficient δ_M need be modeled as a function of both GC implementation and the application characteristics instead of set constant that only covers GC implementation. Thus, coarse-granularity might generate false GC cost predication that leads to false schedulability analysis. Moreover, in this approach, the cost coefficients depend on details of both the GC implementation and the mutator behavior. This works against the goal of separately characterizing the GC implementation and the GC load offered by the application and its workload.

More recently, Bacon, [7], introduced a GC cost model for their Metronome real-time GC implementation (the mostly non-copying collector) with a granularity in between the detailed granularity of Bengtsson's approach and coarse granularity of the Henriksson and Kim approaches. In Metronome, major contributors to the cost of each GC stage are characterized by *GC coefficients* that express the rate at which that kind of work is performed. The Metronome GC is designed to provide a specific minimum mutator utilization (MMU) over a given time interval Δt. The cost

model is used in the real-time scheduling analysis to ensure that the required MMU is achieved. The worst-case GC cycle C_{GC} cost is the only cost discussed in the GC cost model presented in Metronome.

Goh, [51, 52], presented a GC cost model for an RTGC implementation in Mono 0.25, [96]. The GC cost estimation granularity is of the same degree as Bacon's. The model expresses GC cost in terms of GC coefficients such as "marking an object in root-scan phase", "scan heap memory to find references in mark phase", *etc.* and mutator-dependent parameters such as "size of memory scanned in root-scan phase", "number of objects marked in mark phase", *etc.* The model does not fully separate the GC implementation, application, and workload effects: the definitions of the mutator-dependent parameters are expressed in terms that depend on the details of the GC implementation. For example, the parameter "number of objects marked in mark phase", in addition to depending on the mutator code and workload depends on both the GC implementation and details of the mutual scheduling of the mutator and GC.

2.4.2 Existing work on GC scheduling mechanisms

The automation of the GC integrating in the real-time embedded system leads to two topics of interests: the GC cost estimates and the GC scheduling mechanisms. The cost of GC should be predictable based on (1) the performance of collector operations and (2) the garbage collection load offered by a real-time task. Separation of these aspects is a step along the road to an engineering approach to garbage collection in real-time systems, allowing prediction of system behavior from knowledge of component behavior and environmental specifications. In this section, GC scheduling mechanisms is studied. The rules of scheduling GC increments along with mutators vary among these approaches that are discussed below:

- In 1978 Baker's [8] approach: each time an object is allocated, an increment of scanning and copying is done so that the rate of copying collection work keeps up with the rate of

allocation [94]. The GC work within an increment is proportional to the size of the allocation request, which makes it a work-based GC scheduling.

- In 1998 Henriksson's [54] approach: A GC increment must be scheduled when $GCR < GCR_{min} = \frac{W_{max}}{F_{min}}$, interrupting mutator activity if necessary. Scheduling is based on the three-level priority process scheme together with the following triggering and interruption policies. Allocation from high priority processes does not trigger GC. Instead, GC motivated by high-level priority processes forms the middle-priority process and is triggered when no high-priority process is in a runnable state. Scheduling analysis must be done in order to tune the collector to a specific real-time task set and real-time platform.

- In 1999 Kim's [61] approach: GC is scheduled as an aperiodic task with the highest priority using the *sporadic server* technique. GC worst-case response time, R_{GC}, is derived by means of memory consumption, M_c, and memory reclamation, M_r, functions obtained from the programmer for a real-time system with a fixed set of periodic tasks. A recent paper, [95], modifies this approach, using deferrable server scheduling and obtaining superior memory utilization. Similarly, in 2001, a more general schedulability analysis where the aperiodic GC task is not limited to have the highest priority is studied in [5].

- In 2003 Robertz's [83] approach: triggering of a GC cycle is implicit in the estimated adaptive GC cycle-time T_{GC} and a GC cycle deadline is calculated so that GC can be scheduled as any other real-time task using real-time scheduling such as Earliest-Deadline-First(EDF).

- In 2003-2004 Bacon's [7] Metronome approach: depending on the metric in use, GC is allowed to execute for C_T CPU time or process C_W MB memory before yielding to the mutator. Similarly, mutators may expend Q_T CPU time or allocate at most Q_W MB memory before yielding to the GC. One of the main results is that interrupt-driven scheduling

based on the space metric and scheduling based on the time metric are essentially equivalent. When GC scheduling is time-based, real-time guarantee aims at satisfying mutator mutator utilization (*MMU*) over a given time resolution Δt.

- In 2006 Goh's [51] approach: their scheduling approach can be viewed as a hybrid of Robertz's and Bacon's. GC task is scheduled as a real-time task with its minimum period P_{GC} calculated. GC task is scheduled with any other real-time task using real-time scheduling Rate-Monotonic Scheduling (RMS) as in [51], which is similar to Robertz's approach. Further, once GC cycle is triggered, a fixed execution time quantum for each GC increment E_{GCI} is assigned, which are invoked periodically with a fixed period P_{GCI} until the GC cycle completes. The execution pattern among GC and real-time tasks when GC cycle is triggered is similar to Bacon's time-based GC approach.

Among the above approaches, except in Baker's approach, GC work was carried out as an individual task in real-time systems in all other approaches for the purpose of better controllability.

2.5 Summary

The fundamental concepts of real-time systems, memory management mechanisms and classical garbage collection algorithms are presented in this chapter as background of the thesis work. The general RTGC approach carries the three characteristics of *Incremental*, *Schedulable* and *Measurable* with the implementation help of barrier techniques. The discussion of the related RTGC work is categorized accordingly to the organization of this thesis work, where the related work on the GC cost model and GC scheduling mechanisms are presented.

The next chapter presents the RTGC framework and the system model which provides an abstraction of the automated RTGC approach and also gives the whole pictures of the architecture of the systematic approach toward integrating RTGC in the real-time embedded system.

CHAPTER 3

RTGC FRAMEWORK AND SYSTEM MODEL

A key attribute of proper real-time systems is predictability. Using GC in real-time systems complicates the situation in order to preserve the predictability, because enough time must be alloted for GC-related activities and time spent in GC must be scheduled so as not to interfere with mutators meeting their deadlines. Recall in chapter 1, the thesis statement is that "Decomposing the model of (1) the performance of collector operations and (2) the garbage collection load offered by a real-time task allows prediction of system behavior from knowledge of component behavior and environment specifications. This approach provides the models needed to engineer GC into real-time applications." This statement is pursued throughout the dissertation in order to satisfy the requirements of an automatic RTGC approach. While decomposing the thesis statement there are two aspects that must be taken care of:

- First, given a real-time system with GC, there are two entities to model: the *real-time task set* with the real-time system and the *garbage collector*. Existing real-time task models can be directly applied to the real-time task set in the RTGC enabled systems. Additional techniques, [41, 88], such as estimations of the worst-case execution time of a real-time task are directly available for obtaining real-time tasks' parameters that must be known *a priori*. There are no existing RTGC models for garbage collection, or existing techniques for estimating the worst-case GC cost. As the common language for describing a broad class of real-time collectors, the **RTGC framework**, presented in section 3.1 in this chapter, offers

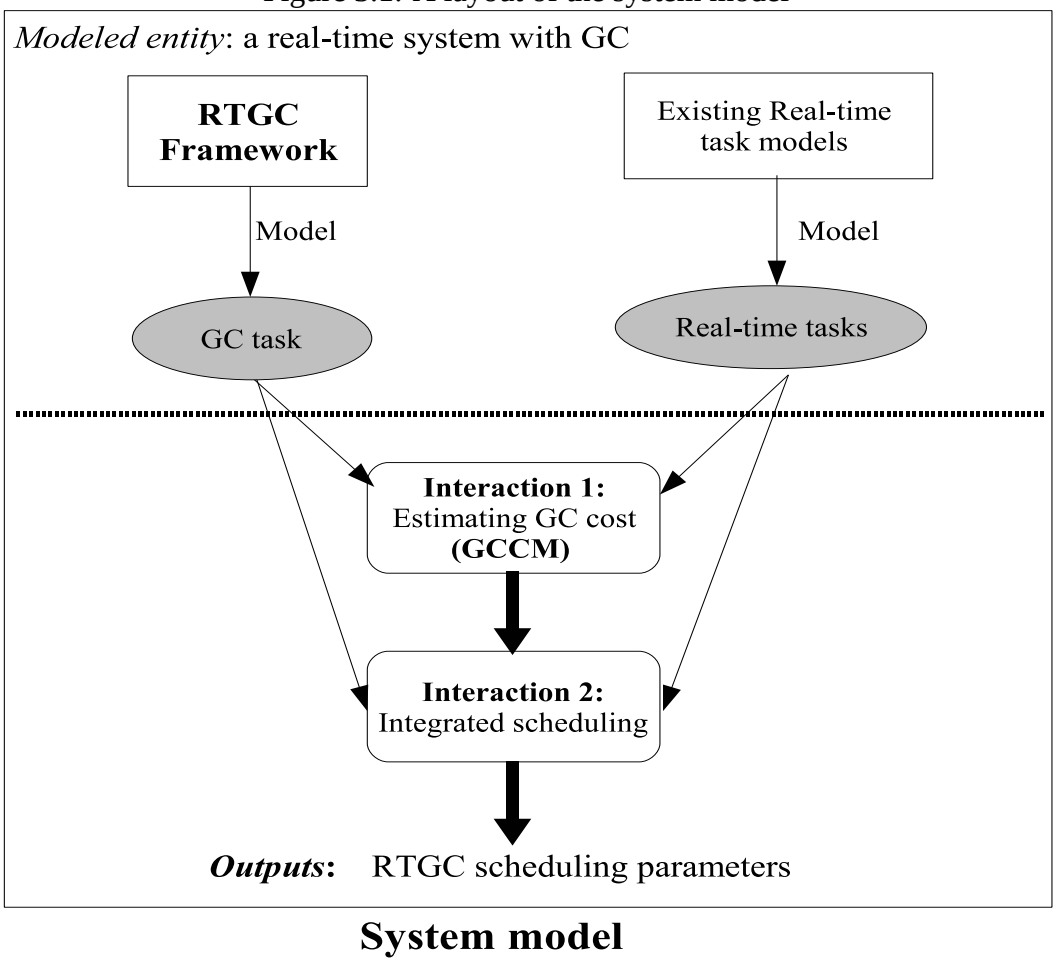
an uniform way to model the GC among various GC implementations. The GC cost model is built for the modeled RTGC to provide GC-related costs estimations.

- Second, between the real-time task set and the garbage collector there are potential interactions that are implicitly stated in the thesis statement. Such interactions come in two parts: (1) the garbage collection load depends on the workload from real-time tasks; and (2) GC-related activities need be scheduled together with the real-time task set. The first interaction relates to the GC cost estimation and the second one determines the GC scheduling parameters. These parameters pertain to the automatic RTGC approach's final outputs.

At the level of a complete real-time system with GC, this chapter will present a **system model** that provides a consistent way to describe GC-enabled real-time systems. The system model covers both modeling of entities and entity interactions within the real-time systems. Figure 3.1 shows the layout of the system model supporting the RTGC approach that the dissertation work defines. It models the given real-time system with knowledge of workload and GC implementation and computes the GC scheduling parameters to assist developers to use GC in a predictable way. More specifically, the **RTGC framework** takes care of modeling the GC task and the **system model** captures the complete RTGC approach.

This chapter is organized as follows: section 3.1 presents the real-time garbage collection framework. A RTGC configuration with GC incrementality in the framework provides a target for automating the task of configuring automatic memory management for a real-time system and its workload in section 3.2. Section 3.3 proposes the system model and section 3.4 discusses how to obtain the system model's inputs. Finally, a copying GC implementation is described in the system model as a case study in section 3.5.

Figure 3.1: A layout of the system model



3.1 RTGC framework

Recall in the thesis statement that the overarching goal is to separate platform-independent functionality and timing concerns from timing-related memory-management issues that depend on details of a real-time system's implementation. In this section, the aspects of a GC component are narrowly observed and the important characteristics and parameters that will eventually have to be modeled and controlled are identified. The *RTGC framework* described here attempts to encompass a variety of RTGC techniques that have been proposed previously and to provide uniform language for describing the parameters that are used to configure them.

A RTGC framework is proposed in this section followed by a description of RTGC configuration in the next section. The framework captures common features of many RTGC algorithms while the RTGC configuration concerns itself with ways that a RTGC algorithm would be controlled in a particular system. The RTGC framework for describing RTGCs involves the following components:

- the underlying GC algorithm: the **Impl** component of the framework;
- the rule for triggering execution of GC cycles: the **Trig** component;
- the method of measuring GC progress: the **Metric** component;
- the rules for scheduling GC increments along with execution of the mutators: the **Sched** component.

There are a few comments on the components then their interactions will be explored more fully in a number of examples below.

3.1.1 The Impl Components

The GC Impl determines the constitution of *a GC cycle* – all activities required to identify and reclaim garbage; such that a GC cycle in mark-and-sweep consists of root scanning, pointer traversal and sweeping *etc.* The GC impl also affects the "shape" of *a GC increment*, which is defined as an individual GC invocation within a GC cycle contributing partial work to the whole GC cycle. The underlying GC algorithm of an RTGC is a major factor in estimating the GC-related cost along with the real-time workload.

The appropriateness of a GC algorithm for real-time use in a given situation depends on what operations it uses and how long they take. Prior work on analyzing the GC overhead includes [8, 9, 24, 22], which are discussed in section 2.2 in chapter 2. It is reasonable for a well-implemented system with GC to have a 10% overhead within execution, [94].

3.1.2 The Metric Components

GC work-metrics include pure time and pure evacuated memory. "Pure" here means that either time or memory measurement is used without consideration of the other. Combinations are also possible such as a memory reclaiming rate *etc.* In general, a GC work-metric must be compatible with the Impl, Trig and Sched components. That is, it must provide the information Trig and Sched need to ensure that deadlines are met.

Other aspects of a good GC work-metric are its simplicity and low-computational cost. It is also important to be able to accurately derive work-metric from available data, but this can be taken too far. For example, in copying implementations pure evacuated memory in *Tospace*, E, is calculated easily by the position of the evacuation pointer B relative to the start S, of *Tospace*. As observed in [54] E is a poor approximation to the actual amount of work performed by the GC because it captures only the cost related to copying and not the cost of finding objects to be copied. Also, the E metric does not incorporate any notion of the total time for a GC cycle. For these reasons it does not provide good guidance for scheduling GC in a real-time system.

In real-time systems, a better GC work-metric is the amount of time actually used for performing GC, [54]. It is *time* that must be controlled and distributed throughout a whole GC cycle in order to meet deadlines. A comprehensive GC cost model that will be presented in chapter 4 shows how to convert all the GC-related work into time, which is directly useful in real-time systems.

3.1.3 The Trig Components

The Trig component of the framework is the rule for determining when a GC cycle is needed. It may use work-based metrics such as the amount of available memory or the predicted amount of garbage. Alternatively, it may use time-based metrics such as wall-clock time, time consumed by the mutators, or predicted time needed to complete a GC cycle. Within a GC cycle the Sched component describes the way that GC increments are interleaved with mutator execution. Hybrid

rules involving both time and work are also possible.

GC triggering and scheduling have received much attention recently, such as Henriksson's and Robertz's PhD theses, [54, 83], and Bacon's paper on time-based scheduling for real-time Java GC, [7]. Henriksson also addresses techniques of reserving memory for high-priority processes in case normal memory is exhausted (because the GC did not keep up).

3.1.4 The Sched Components

Scheduling involves jointly scheduling GC work together with mutator work. Two questions must be answered: when to start an increment and how much work to do before switching back to a mutator. For instance, if time is chosen as the GC work metric as in [83], fixed priority scheduling and earliest deadline first (EDF) scheduling for real-time tasks can be directly applied to schedule GC work. In [61], time is also chosen as a GC work metric, but GC work is scheduled as a sporadic real-time task with an estimation of its worst-case response time used as the GC cycle deadline. Evacuated memory is used as the GC work metric in [54]. It schedules GC work based on a comparison between the GC current work ratio and a minimum GC ratio computed *a priori*.[1]

If the *GC incrementality*, defined to be the GC increment size in either GC work or CPU time, is explicitly controllable, then GC can be scheduled with a set of GC incrementality parameters. Such GC Sched instances will be extensively discussed in chapter 5 together with schedulability analysis.

3.1.5 Examples of use of the RTGC framework

The RTGC framework can be used to describe, analyze and identify key components of an existing RTGC approach. In order to illustrate the use of the proposed RTGC framework now several existing RTGC approaches are described beginning with Baker's original system, [8].

- **RTGC proposal 0: Baker78 [8]**

[1]The detail scheduling strategy in [54] is given in subsection 3.1.5.

- **Impl** Minsky garbage collector, [78], with read barrier;
- **Trig** TRUE; as soon as one GC cycle ends the next one begins;
- **Metric** Pure evacuated memory;
- **Sched** Each time an object is allocated, an increment of scanning and copying is done so that the rate of copying collection work keeps up with the rate of allocation [94]. The GC work within an increment is proportional to the size of the allocation request.

- **RTGC proposal 1: Henriksson98 [54]** The work focuses on embedded systems which are assumed to have a number of high-priority (typically periodic) threads that must meet hard deadlines.
  - **Impl** Copying GC;
  - **Trig** Based on the three-level priority scheme, GC motivated by low-priority processes is triggered when allocation occurs; however, allocation from high priority processes does not trigger GC. Instead, GC motivated by high-level priority processes forms the middle-priority process and is triggered when no high-priority process is in a runnable state;
  - **Metric** GC work, W, is measured by the amount of evacuated memory. The measurement improves on pure evacuated memory by taking into account other GC time-consuming activities yielding a better approximation to the actual time used by the GC. Other important metrics of the scheme are: A, the amount of memory newly allocated in the current GC cycle W_{max}, the estimated total work of a GC cycle, and F_{min}, the minimum amount of memory available in Tospace immediately following a flip. The GC ratio, $GCR = W/A$, indicates how well the GC is keeping up with allocation;

- **Sched** A GC increment must be scheduled when $GCR < GCR_{min} = \frac{W_{max}}{F_{min}}$, interrupting mutator activity if necessary. Scheduling is based on the three-level priority process scheme together with triggering and interruption policies described above. GC Scheduling was proposed in order to tune the collector to a specific real-time task set and real-time platform. Schedulability of the garbage collector verified that such GC work with the high-priority processes of the system is schedulable.

- **RTGC proposal 2: Kim99 [61]**
  - **Impl** A modified copying GC with hardware support for fast initialization of to-space;
  - **Trig** GC is triggered when the current amount of available memory becomes less than a certain threshold: ($M_{current} <= M_{threshold}$);
  - **Metric** pure time;
  - **Sched** GC is scheduled as an aperiodic task using the *sporadic server* technique. GC worst-case response time, R_{GC}, is derived by means of memory consumption, M_c, and memory reclamation, M_r; functions obtained from the programmer for a real-time system with a fixed set of periodic tasks. A recent paper, [95], modifies this approach, using deferrable server scheduling and obtaining superior memory utilization.

- **RTGC proposal 3: Robertz03 [83]**
  - **Impl** No fixed GC implementation is assumed: any GC meeting a defined GC interface, [58], can be used;
  - **Trig** A deadline is calculated for completion of each GC cycle. It is obtained from *a priori* knowledge of mutators' memory usage metrics and from a run-time feedback facility. Triggering of a GC cycle is implicit in the estimated adaptive GC cycle time T_{GC};

- **Metric** The actual CPU time spent in GC's work. Robertz describes a method for predicting the GC time in a GC cycle;
- **Sched** Since each GC cycle has a deadline it is scheduled as any other real-time task using real-time scheduling such as EDF.

- **RTGC proposal 4: Bacon03 [7]**
  - **Impl** A mostly non-copying (mark-and-sweep) GC. When large enough blocks are available the collector is non-copying but when fragmentation prevents allocation limited copying will be done;
  - **Trig** A GC cycle is triggered when the amount of memory in use reaches a pre-defined threshold similar to the Kim99 RTGC. The chosen threshold is $m + e$ where m is the maximum live heap memory and e is the space required to allow allocation to proceed during a GC cycle;
  - **Metric** The collector can be controlled using either time or space metrics.
  - **Sched** Depending on the metric in use, GC is allowed to execute for C_T CPU time or process C_W MB memory before yielding to the mutator. Similarly, mutators may expend Q_T CPU time or allocate at most Q_W MB memory before yielding to the GC. One of the main results is that interrupt-driven scheduling based on the space metric and scheduling based on the time metric are essentially equivalent.

JOSES, [30], includes a RTGC approach based on reference counting. The Real-time Reference Counting (RTRC) scheme does not fit well in our RTGC framework because there is no way to identify a GC cycle. In RTRC all GC work takes place incrementally during allocations and pointer manipulations. The RTRC proposal describes how to bound the execution time of these incremental operations using deferred reference counting, [92], but as in [8] there is no attempt

to estimate the overall work required for collection. Further, reference counting collectors require backup from a tracing collector to solve the problem of cyclic garbage. Region-based approaches such as that of [56] also fall outside the current framework.

In this dissertation, the modified mark-and-sweep Boehm-Demers-Weiser (BDW) collector was implemented in the Mono runtime, [96], to prototype the system model and RTGC framework described here. The detail implementation will be given in chapter 4. As the last example of the use of the RTGC framework in this section, our modified BDW collector is described as:

- **Impl** A mark-and-sweep BDW GC. A 2-level hierarchical allocation mechanism separately allocates large objects (bigger than one half the allocator page size) and small memory objects is used as the allocator;

- **Trig** A GC cycle is triggered when either the amount of free memory drops below a predefined threshold or according to the pre-calculated GC cycle period;

- **Metric** Both the CPU time and GC *work unit* can be used as GC metrics. The GC work unit is defined as the atomic GC execution performed within a GC phase during one GC cycle. For instance, thread scan is a GC phase in one mark-and-sweep collection cycle and the GC work unit is *scanning one thread stack for references*.

- **Sched** When GC is triggered, the interleaving pattern of GC and mutator work is determined by the calculated GC and mutator increment sizes in terms of CPU time. Such interleaving continues until the completion of the current GC cycle and resumes when the next GC cycle is issued. The Sched component calculates the increment sizes which are further verified by the schedulability analysis to guarantee that real-time deadlines are met.

Table 3.1: RTGC configuration metrics

Time-based		Work-based	
Symbol	Description	Symbol	Description
t_M	Time spent by mutator	A_M	Memory allocated by mutators
t_{GC}	Time spent by GC	G_{GC}	Memory reclaimed by GC
U_M	Mutator Utilization	A_R	Allocation Rate
U_{GC}	GC Utilization	G_R	Garbage Collection Rate
U_{ratio}	$U_{ratio} = U_M/U_{GC}$	W_{ratio}	$W_{ratio} = A_R/G_R$

3.2 RTGC configuration

The RTGC framework described above helps identify key parameters that a designer can use to control the GC behavior of a given GC in a real-time system. A particular set of parameterized rules and parameter values for a collector is termed an *RTGC configuration*. The RTGC configuration of a GC-enabled system can be fixed at design time or can be changeable, which allows the system to adapt to different workloads as the system runs.

What does an RTGC configuration look like? An RTGC defined in the RTGC framework contains

- entities: Metric and Impl that provide ways to measure and characterize GC;
- rules: Trig and Sched that describe how GC work is invoked and distributed.

The rules and parameter values that control Trig and Sched, *i.e.*, the RTGC configuration, are chosen using knowledge of the real-time system's workload and details of the mutator's allocation behavior. Table 3.1 presents possible configuration metrics and table 3.2 present possible GC triggering rules for generic time-based and work-based RTGCs. Not all of these configuration metrics and rules would be used in any given configuration.[2]

Consider a time-based RTGC with utilization U_M and U_{GC} as metrics. A configuration might

[2]In table 3.2 of the GC triggering formulas, the metrics with a superscript *thre* are the preset values to be compared to the metric's observed values.

Table 3.2: RTGC configuration rules - **Trig**

Trig: A set of rules using thresholds		
	Formula	**Explanation: GC is triggered when...**
Time-based	$t_M > t_M^{thre}$	mutator continuously consumes CPU time for t_M^{thre}
	$t_{GC} < t_{GC}^{thre}$	GC is continuously suspended from CPU time for t_{GC}^{thre}
	$U_M > U_M^{thre}$	when mutator utilization goes beyond U_M^{thre}
	$U_{GC} < U_{GC}^{thre}$	when GC utilization goes below U_{GC}^{thre}
	$U_{ratio} > U_{ratio}^{thre}$	when mutator/GC utilization ratio goes beyond U_M^{thre}
Work-based	$A_M > A_M^{thre}$	when mutator allocates A_M^{thre} amount of memory since last GC cycle
	$A_R > A_R^{thre}$	mutator's allocation rate over the preset time interval goes beyond A_R^{thre}
	$G_R < G_R^{thre}$	when GC collection rate over the preset time interval goes below G_R^{thre}
	$W_{ratio} > W_{ratio}^{thre}$	when mutator/GC rate ratio over the preset time interval goes beyond W_R^{thre}

contain the rule *trigger GC when*

$$(U_M > U_M^{thre1}) \vee (U_{GC} < U_{GC}^{thre}) \vee (U_{ratio} > U_{ratio}^{thre})$$

along with values for thresholds U_M^{thre1}, U_{GC}^{thre},and U_{ratio}^{thre}. This particular rule captures the concern that GC work should not be allowed to get too far behind mutators' allocation (so as to avoid memory exhaustion). Similarly for scheduling, the rule *suspend GC work when* $U_M < U_M^{thre2}$, with a different threshold for mutator utilization, may be needed to ensure sufficient time is available for the mutators.

Some possible RTGC scheduling rules, when GC cycle is triggered, are:

- *Rule #1*: Schedule GC using estimated GC granularity which is the number of increments per GC cycle
- *Rule #2*: Schedule GC using its worst-case response time R_{GC} (GC cycle deadline) with a sporadic server (SS)
- *Rule #3*: Assign GC increment with a calculated time quanta t_{GC} to interleave with mutators until a GC cycle is complete
- *Rule #4*: Assign GC increment with a set of computed work-based quanta, such as scanned

thread stack quanta t_{GC}, evacuated memory quanta m_{GC} *etc.* to interleave with mutators until a GC cycle is complete

- *Rule #5*: Issue GC increments when certain condition rule becomes true, for instance, $U_{GC} < U_{GC}^{thre}$. Similarly, suspend GC execution when such condition rule becomes false.

3.3 System model for GC-enabled Real-time systems

In the RTGC framework in section 3.1, any particular RTGC proposal consists of four elements: *Impl*, *Trig*, *Metric* and *Sched*. The system model introduced here provides a uniform way to describe a GC-enabled real-time system where GC is modeled using the RTGC framework. It also generalizes the systematic RTGC approach that derives GC scheduling parameters by giving a real-time system with its workload. The system model lays out the necessary components to assist real-time developers to use GC in a predictable way.

We now describe a system model $\Psi = (\Gamma_{N+1}, \texttt{GCCM}, \texttt{SCHED})$ for GC-enabled real-time systems on uniprocessors where GC is modeled using the RTGC framework. Here, Γ_{N+1} is a real-time task set with the GC task, `GCCM` is the GC cost estimation component and `SCHED` is the integrated GC scheduling and schedulability analysis component. These three components are explained separately in this section.

3.3.1 Task model Γ_{N+1} component

A real-time task set can be modeled in several ways. These models include (1) *periodic or sporadic model*, [11, 15, 33], which is used to model a general real-time system comprised of a finite collection of tasks; (2) *multiframe model*, [3], which was proposed to accurately model multimedia traffic; (3) *generic recurring real-time task model*, [85], which models conditional real-time code by directed acyclic graph *etc.* Here we consider periodic or sporadic task models and tailor them to fit the GC-integrated real-time systems.

The task model follows closely as the one established in [15] for the convenience to integrate a general schedulability analysis for fixed-priority real-time systems. Such schedulability analysis will be integrated in our RTGC scheduling mechanisms in chapter 5.

A real-time task set with RTGC, denoted as $\Gamma_{N+1} = \Gamma_N \cap \{\tau_{GC}\}$, where $\Gamma_N = \{\tau_1, \cdots, \tau_N\}$ of N periodic (or sporadic) tasks, where each task $\tau_i = (\Phi_i, C_i, \sigma_i^{wb}, T_i, D_i, A_i, B_i)$ is characterized by: an initial activation time Φ_i; a worst-case execution time C_i; a (write or read) barrier cost σ_i^{wb} derived using the GC cost model; a period (or a minimum inter-arrival time for sporadic tasks) T_i; a relative deadline D_i not greater than T_i; A_i, the worst-case memory allocation during one request of task τ_i; and a blocking time B_i, where $B_i = 0$ for task sets with independent tasks and $B_i \geq 0$ for task sets with dependent tasks using shared resources. The relative deadline D_i is no greater than T_i that $D_i \leq T_i$. In a GC-enabled real-time system, the worst-case execution time of a real-time task τ_i is recalculated by $C_i * \sigma_i^{wb}$ to include the GC barrier overhead. The task model Γ_{N+1} in Ψ here does not include spontaneous tasks, since they have unbounded arrival rates and related analysis can only have statistical guarantee.

Without loss of generality when fixed priority assignment (FP) scheduling is used, the tasks in Γ_N are ordered by decreasing priority, where τ_1 has the highest priority. Tasks ordered in decreasing priorities are denoted as:

$$\Gamma_{N+1} = \{\tau_1, \cdots, \tau_i, \tau_{GC}, \tau_{i+1}, \cdots, \tau_N\}$$

τ_{GC} is an individual task that provides garbage collection work. A *GC cycle* is one invocation instance of τ_{GC}, which is scheduled in *GC increments*. Similarly, GC task τ_{GC} is characterized as Φ_{GC} (GC activation time), C_{GC} (GC cycle worst-case execution time), T_{GC} (a period or a minimum GC cycle inter-arrival time), D_{GC} (relative deadline which is not greater than T_{GC}). The C_{GC} is the major estimate computed by the GC cost model in chapter 4.

Each task τ_i in Γ_{N+1} consists of an infinite sequence of jobs, where τ_{ik} denotes the kth job of task τ_i. The release time of task instance τ_{ik} is denoted as $r_{ik} = \Phi_i + (k-1) \cdot T_i$. For each task τ_i, the ratio $U_i = \dfrac{C_i}{T_i}$ is called the *utilization factor* of task τ_i. And the value $U_p = \sum_{i=1}^{N} U_i$ is called the *total CPU utilization factor*.

In order to specify the real-time tasks' dependency, which is usually characterized as precedence and synchronization constraints, [43], $Dep = (D, \overline{D})$ denotes such dependency features, where D groups all real-time tasks with dependencies while $\overline{D}$ gathers the rest of individual tasks. The structure of the set D further describes the exact dependency among the tasks.

The most simple scenario is that the complete real-time tasks are independent that $Dep = \{NULL, \{\tau_1, \cdots, \tau_n\}\}$. The other common scenario is that some tasks execute in Producer-Consumer pattern, $Dep = \{\{(\tau_{p_l}, \tau_{c_l}), \cdots, (\tau_{p_k}, \tau_{c_k})\}, \{\tau_i, \cdots, \tau_j\}\}$, which groups tasks in Producer-Consumer pattern in pairs in D and those independent tasks in $\overline{D}$. Task dependencies affect object lifetimes, such that objects residing in shared-memory extend their lifetime as long as there are active threads using them. The extended object lifetimes eventually affect the amount of live memory an application needs that changes GC work load. Characterizing the real-time tasks dependency helps providing better options to obtain GC-related metrics in the automated RTGC approach. A discussion of estimating maximum live memory in section 3.4 shows how dependency affects the estimation.

3.3.2 `GCCM` component

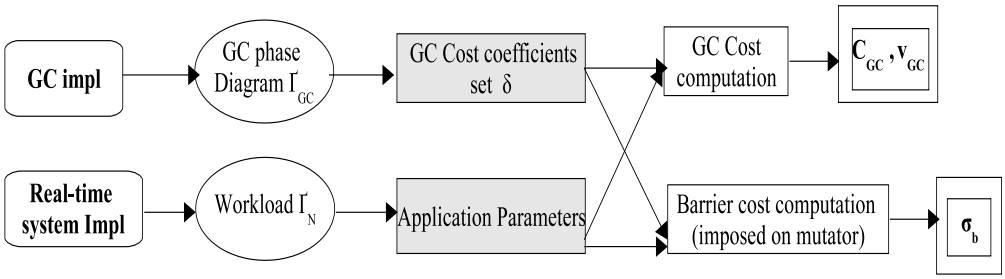
The `GCCM` component in the system model computes the GC-related cost along with the workload. In other words, it abstracts the computation of the GC-related cost, for instance, the worst-case GC cycle execution time, C_{GC}, in terms of (1) the performance of collector operations and (2) the garbage collection load offered by real-time tasks. Here,

$$\texttt{GCCM} = (In_{GC_{Impl}}, In_{App_{Impl}}, Out_{GC}, Out_{barrier})$$

Each element in the GCCM will be explained separately after presenting the basic GCCM architecture.

Separation of these aspects allows prediction of system behavior from knowledge of component behavior and environmental specifications. Figure 3.2 shows the basic architecture that constitutes a valid GCCM. It also depicts the flow of GC-related cost computation.

Figure 3.2: GC Cost model GCCM



In summary of the GC cost model architecture, the GCCM takes a real-time system with workload and GC implementation as inputs and calculates the GC-related costs for both the GC task and mutator tasks as outputs.

1. *GCCM* inputs:

   - **GC impl** $In_{GC_{Impl}}$: determines the performance of collector operations. A GC phase diagram is drawn to describe all the phases that form a GC cycle. The GC phase diagram helps identifying different GC operations so that a set of GC cost coefficients can be defined. This set of GC cost coefficients should be stable across a variety of applications on a targeted system.

   - **Real-time system impl** $In_{App_{Impl}}$: the real-time task set determines the GC load. Thus, the application-dependent parameters are the factors determining the GC-related cost. The parameters are L (the maximum amount of live memory of the real-time application), R (maximum root size), θ (an upper bound on reference density) *etc.* Section 3.4

discusses approaches to obtain these application-dependent parameters.

2. *GCCM* outputs:

   - **GC cost estimates** Out_{GC}: the major estimate is the worst-case GC cycle execution time, C_{GC}. Another estimate is the GC scheduling resolution, ν_{GC}, which affects the mutator pause time imposed by GC execution when GC is triggered. In other words, the mutator pause time from executing GC is no less than the GC scheduling resolution, ν_{GC}. The use of ν_{GC} will be given in section 5.2.3 in chapter 5 with the scheduling mechanisms. The sizes of GC increments are other interesting estimates.

   - **Barrier cost estimates** $Out_{barrier}$: estimates the GC barrier operations cost carried out by mutators. This overhead should be treated as a *tax* added to the mutator execution. For instance, a real-time task with an original worst-case execution time C_i needs be recalculated as $C_i * \sigma_i^{wb}$ when GC is integrated, where σ_i^{wb} is the barrier cost for real-time task τ_i computed in the `GCCM`.

Both the GC cost and barrier cost need knowledge of the *GC impl* and the application-dependent parameters to perform the cost calculation. A case study of the GC cost model with a copying collector (CC), $\texttt{GCCM}_{CC}$ is given in section 3.5.1. A detailed GC cost model for a mark-and-sweep collector (MS) is presented and implemented in chapter 4.

3.3.3 `SCHED` component

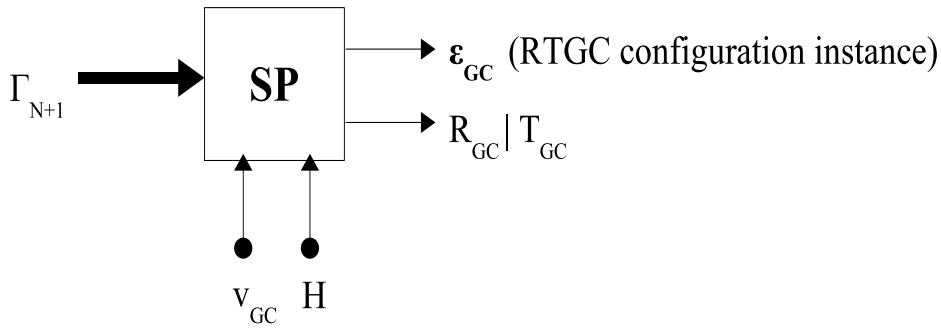
The `SCHED` component in the system model computes the GC scheduling parameters in terms of either *RTGC configuration* parameters described in section 3.2 in the RTGC framework or a classical real-time task's scheduling parameters.

$$\texttt{SCHED} = (SP, (\xi_{GC} | \overline{\xi}_{GC}))$$

where SP is the chosen real-time scheduling and schedulability analysis policy, ξ_{GC} is the *RTGC configuration* based GC scheduling parameters and $\overline{\xi}_{GC}$ is the classical real-time task's scheduling parameters, *i.e.*, T_{GC}, the period of the GC task; or R_{GC}, the worst-case GC response time. Each element in the SCHED will be explained separately after presenting the basic SCHED functionality.

The GC execution characterized by either ξ_{GC} or $\overline{\xi}_{GC}$ needs be scheduled with the real-time task set under the scheduling and schedulability policy described in SP. Figure 3.3 shows the basic functionality that a valid SCHED needs have.

Figure 3.3: The GC scheduling component *SCHED*



With the GC scheduling resolution, ν_{GC} and the heap requirement, H, *Sched* takes the knowledge of Γ_{N+1} as inputs, and outputs GC scheduling parameters, such as ξ_{GC} or $\overline{\xi}_{GC}$(R_{GC}, T_{GC}, *etc.*). Schedulability analysis in SP verifies the scheduling outputs feasibility. The ν_{GC} and the heapsize H are tunable with various GC scheduling parameters. (Tunable attributes have "black dots" associated with the arrow in the figure.)

The attributes are known or computed (by GCCM) *a priori* in Γ_{N+1}, for instance, the real-time task's worst-case execution time, C_i, the worst-case GC cycle time C_{GC}. ν_{GC}, the GC scheduling resolution and H, the system heapsize are two hardware platform-dependent parameters that are involved in the SCHED. During the scheduling and schedulability analysis process with SP, these two systematic parameters are tunable with GC execution parameters using either ξ_{GC} or $\overline{\xi}_{GC}$.

In summary, the SCHED takes mutator and GC's attributes as inputs and calculates the GC execution parameters in terms of ξ_{GC} or $\overline{\xi}_{GC}$.

- SP the space of scheduling here aims at two groups of real-time systems: *minimum mutator utilization* MMU-based systems,[3] [26] and fixed-priority systems with preemption.[4] For the fixed-priority systems, the scheduling policy that we consider handles arbitrary priority assigning methods, for instance, rate-monotonic scheduling (RMS) is the most common used one. When task synchronization occurs, existing protocol, for instance, priority inheritance protocol(PIP) or priority ceiling protocol(PCP), [86], can compute real-time tasks blocking factor so to support schedulability analysis.

- ξ_{GC} GC task is issued according to a RTGC configuration instance usually with higher priority than other mutator tasks. Recall that a particular set of parameterized rules and parameter values is termed a *RTGC configuration* in section 3.2. An RTGC configuration instance ξ_{GC} consists of three parts:

$$\xi_{GC} = (m, \epsilon_{GC}, (P_{GCI} \&\& P_{MI}))$$

where m is the GC execution mode that contains time-based (t), work-based (w) and hybrid (h) execution modes; ϵ_{GC} is the GC invocation boolean expression that evaluates to trigger GC cycles when the expression turns FALSE; and P_{GCI} and P_{MI} are GC increment and mutator increment quanta in CPU time or work. Three sample RTGC configuration instances ξ_{GC_h}, ξ_{GC_t} and ξ_{GC_w} are:

$$\xi_{GC_h} = ((m = h), (\epsilon_{GC} = F > H * \alpha), (P_{GCI} = T))$$
$$\xi_{GC_t} = ((m = t), (\epsilon_{GC} = U_{GC} > U_{GC_{min}}), (P_{GCI} = T))$$
$$\xi_{GC_w} = ((m = w), (\epsilon_{GC} = A_M), (P_{GCI} = N))$$

[3] *MMU* is the minimum mutator utilization over a given time interval Δt, denoted as μ_{MMU}.
[4] The scheduling considered here is *predictable* that is in contrast to *heuristic* scheduling.

ξ_{GC_h} shows a hybrid mode RTGC where GC cycle is triggered when free memory F drops under fraction α of total heap H, and each GC increment is assigned with minimum time T. ξ_{GC_t} is a time-based RTGC, GC cycle is triggered when GC CPU utilization U_{GC} drops under its expected minimum value $U_{GC_{min}}$, and each GC increment is assigned with minimum time T. ξ_{GC_w} shows a work-based RTGC, GC cycle is triggered when mutator allocation rate A_M goes beyond a preset value A_{safe}, with different GC implementation, GC increment is assigned N execution steps depending on which GC phase the GC cycle is undergoing.

RTGC configuration instance shows how GC is interleaved with mutators when GC is triggered, and GC cycles are scheduled as a periodic or aperiodic real-time task in order to perform the integrated schedulability analysis with the real-time task using existing approaches. Each RTGC configuration instance also has a scheduling parameter such as T_{GC} or R_{GC}. The choice of the scheduling parameters depends on the scheduling policy described in SP.

In chapter 5, two detailed instances of the RTGC configurations ξ_{GC_w} and ξ_{GC_t} are extensively designed and analyzed with the chosen scheduling policies. Algorithms are proposed to support the schedulability analysis and implemented to assist the use of our modified BDW collector.

- $\overline{\xi}_{RTGC}$ GC task is scheduled as a classical periodic or aperiodic task and GC's interleaving with mutators are implicitly taken care of by the scheduler. As a periodic task, GC period T_{GC} is the scheduling parameter.As an aperiodic task, GC response time R_{GC} is the scheduling parameter to calculate. Aperiodic tasks are usually served by periodic servers, characterized as server capacity C_S (maximum execution time) and server period T_S. There are various algorithms for these servers that differ in performance and computation complexity. There are polling server(PS), [61, 5], deferrable server(DS), [90], constant bandwidth server (CBS), [67], sporadic server (SS), [89] *etc.*

In section 2.4.2 in chapter 2, a list of existing work on GC scheduling mechanisms are presented. There approaches can be descried using our SCHED. For instance, Bacon's GC scheduling mechanism can be denoted as $\texttt{SCHED}_{Bacon} = (MMU, \xi_{GC_{Bacon}})$, where

$$\xi_{GC_{Bacon}} = (m = T \parallel W, \epsilon_{GC} = F > H{*}\alpha, (P_{GCI} = C_T \&\& P_{MI} = Q_T) \parallel (P_{GCI} = C_W \&\& P_{MI} = Q_W))$$

Robertz's GC scheduling mechanism, [83], can be denoted as $\texttt{SCHED}_{Robertz} = (EDF \parallel RMS, \overline{\xi}_{GC_{Robertz}})$, where GC is scheduled as a periodic task with the computed T_{GC}.

Low-latency requirement and RTGC-configuration based GC scheduling

Some real-time systems correctness depends not only on the logical result of computations, the real-time tasks deadlines but also on guaranteed low-latency [2]. In other words, the system latency needs be bounded. Examples include the mission critical real-time applications, for instance, given the speed of missiles, it is critical for the flight test operator to be given the position information as fast as it is acquired. In other words, the lowest possible latency of the telemetry processing system is essential. Other examples include audio applications that need system keeping up with the audio streaming or network communication applications, such as high-speed data transfer [10, 40] that need the time delay between receiving input data and responding to that data, desire low-latency. RTGC-configuration based GC scheduling ξ_{GC} is based on an ability to control GC incrementality explicitly. The possible mutator pause time imposed by the GC execution can be controlled by the explicit GC incrementality and schedulability analysis covers the latency concern as well.

The explicit GC incrementality controllability is an RTGC implementation choice. A prototyped RTGC introduced in section 4.4 in chapter 4 will show the detail implementation.

Table 3.3: Partial inputs for the system model

Input names	Element name in Ψ	Description
$\tau_i.A_i$	Γ_{N+1}	the maximum real-time task's allocation during its single invocation
L	GCCM	the maximum live memory at any point during an application's execution
R	GCCM	a root-set generally consists of global, static data, register data, and thread stacks.
θ	GCCM	the maximum number of references per unit of heap size, hence the maximum reference density in heap

3.4 Application-dependent parameters

In the system model, component Γ_{N+1}, GCCM and SCHED need application-dependent memory usage knowledge to perform the computations. Real-time tasks maximum allocation A_i in Γ_{N+1} should be obtained *a priori*. GCCM needs have the maximum live memory L, maximum root set size R, maximum reference density θ *etc.* to compute the GC-related cost. And SCHED needs use the GC cost estimates together with knowledge of real-time workload to calculate the GC scheduling parameters.

Modeling the interactions between the application, scheduler, and GC to derive values for A_i, L, R *etc.* is, at this time, an open problem. This section provide a brief discussion on the current available techniques to determine some of these values that are listed in table 3.3.

These metrics are common across various GC implementations. Different GC implementations may require additional application-specific memory usage metrics, for instance, a more complete set of metrics will be given in chapter 4 for a mark-and-sweep collection implementation. The metrics in table 3.3 are:[5]

- $\Gamma_{N+1}.\tau_i.A_i$: maximum real-time task's allocation during its single invocation.

  Worst-case execution time analysis already determines the maximum execution time that can be consumed over all possible paths in real-time tasks. It is straightforward to determine maximum amount of data allocated over any possible path for an request of task τ_i. How to statically obtain maximum allocation is demonstrated in [49, 72].

[5]Each metric is denoted in the form of $P.Q$, where P shows the component name in the system model and Q is the metric's name.

- `GCCM`.L: maximum live memory at any point during an application's execution.

  The maximum live memory L is statically undecidable. Type analysis or bounding annotation techniques can be applied so that L is determined for a subset of applications with constraints. Chin *et al.*, [27], proposed static memory usage analysis on the basis of type system with size properties that specifies the amount of memory required to execute a well-typed object-oriented program. Leena, Stoller and Liu, [91], demonstrated an automatic space usage analysis approach for their language named *MEMFL* which contains many features commonly found in high-level languages. Persson, [80], made an attempt to derive such bounds using annotations, such as specifying the maximum recursion depth. It is generally assumed that developers who execute Java applications know maximum live memory for a given application, [91, 49, 16, 72].

  For applications with a steady state of memory usage, maximum live memory can be approximated by the memory usage using profiling techniques, [37], when applications reach the steady state. Pessimistic estimation based on allocation with worst-case object lifetime gives maximum live memory. Task dependency Dep characteristics affect the maximum live memory estimation.

- `GCCM`.R: maximum root set during an application's execution.

  A root-set consists of global, static data, register data, and thread stacks. For a tracing collector, root size is estimated by the sum of the data segment size R_d processes and the maximum thread stack size R_t multiplied by total number of threads n, therefore, $R = R_d + nR_t$. High-level languages allow users to set the maximum thread stack size when executing an application, [37, 72, 49].

- `GCCM`.θ: the maximum reference density in heap during an application's execution.

Since liveness is determined by tracing references from a program's live root-set, reference density directly affects GC work load. Reference density differs in total reference density θ and non-null reference density θ_{nr}. Fortunately, in languages like Java or C#, reference fields are explicitly declared. Each object type's reference density can be determined as long as all object types are known *a priori*. Worst-case total reference density can be bounded by assuming worst-case reference density for all objects, [72]. Worst-cast non-null reference density can be bounded by assuming that all references are non-null.

Discussions for other application-dependent parameters can be found in [14, 35, 37].

3.4.1 Task dependency and application-dependent parameter estimates

Recall that the maximum live memory at any point during an application's execution is statically undecidable. Type analysis or bounding annotation techniques are applied in order to statically decide maximum live memory for a subset of applications with constraints. Another option is using maximum mutator allocation A_{max} to estimate maximum live memory L. The calculation of A_{max} for an application is affected by tasks execution dependency.

In this section, a task set having only independent tasks and one containing producer-consumer task dependency are analyzed separately. Assume that L_{init} is the maximum live memory during system initialization, including runtime initialization and application initialization before the real-time tasks are scheduled to execute..

- In the case that only independent mutator tasks in Γ_N, maximum live memory L is calculated with Equation (3.1).

$$L = \sum_{i=1}^{i=N} A_i + L_{init} \tag{3.1}$$

  where, the maximum live memory for the real-time task set are estimated using the task maximum allocation A_is.

- In the case that there are producer-consumer tasks in Γ_N: we only consider the one-to-many producer-consumer dependency here, where there are multiple consumers consuming the "products" produced by a single producer. A $1-m$ producer-consumer task group is denoted as $PC_{i_p} = (\tau_{i_p}, \tau_{i_{c_1}}, \cdots, \tau_{i_{c_n}})$, where τ_{i_p} is the producer task and $\tau_{i_{c_k}}$ with $k = 1 \cdots n_{i_p}$ are the consumer tasks. An example of this communication is a sensor driver thread that periodically collects data and puts them for further processing by one or more consumer threads.

  For a real-time task set Γ_N with $1-m$ producer-consumer task groups, the task dependency $Dep.D$ consists of $PC_{1_p}, \cdots, PC_{i_p}, \cdots, PC_{N_p}$. How much data will be alive in the worst case if producer-consumer task groups exist in Γ_N?

  The worst case happens when a consumer $\tau_{i_{c_k}}$ takes over(uses) all objects produced by producer τ_{i_p} at the start of its period, keeps them alive until the end of the period before freeing them. Therefore, the maximum amount of live data from the consumer $\tau_{i_{c_k}}$ is $\lceil \frac{T_{i_{c_k}}}{T_{i_p}} \rceil \cdot A_{i_p} + A_{i_{c_k}}$ and the maximum amount of live data from the producer is $\lceil \frac{T_{i_{c_k}}}{T_{i_p}} \rceil \cdot A_{i_p}$. The maximum live memory PC_{i_p} for a real-time producer-consumer group PC_{i_p} is calculated with Equation (3.4.1).

$$L_{PC_{i_p}} = n_{i_p} \cdot \lceil \frac{T_{i_{c_k}}}{T_{i_p}} \rceil \cdot A_{i_p} + \sum_{k=1}^{k=n_{i_p}} A_{i_{c_k}}$$

  Assume that real-time task set Γ_N contains N_{PC} real-time producer-consumer group PC_{i_p}, where $i = 1 \cdots N_{PC}$ and the rest of the real-time tasks are independent tasks. Therefore, the application maximum live memory L is calculated:

$$L = \sum_{i=1}^{i=N_{PC}} L_{PC_{i_p}} + \sum_{j \epsilon (\Gamma_N - Dep.D)} A_j + L_{init} \tag{3.2}$$

3.5 Case Study: model of a real-time system using a copying collector (CC)

This section models a copying collector using the system model described above. $\texttt{GCCM}_{CC}$ and $\texttt{SCHED}_{CC}$ used in this section denote the modeled component `GCCM` and `SCHED` for a copying collector using the system model.

3.5.1 $\texttt{GCCM}_{CC}$

Minsky is credited with the first copying garbage collector, [78], Cheney with an elegant 2-space collector, [24], and Baker with a real-time version of Cheney's algorithm, [8]. Here we consider a simple semi-space copying collector with bump pointer allocator, [60]. Incremental GC is supported by a write barrier as described in [26]. Write barrier cost is charged to the mutator in the way that mutator records relevant reference updating information in its thread's *write-log* during objects copying. These logs are processed by the GC after it finishes copying objects that can be traced from the root-set.

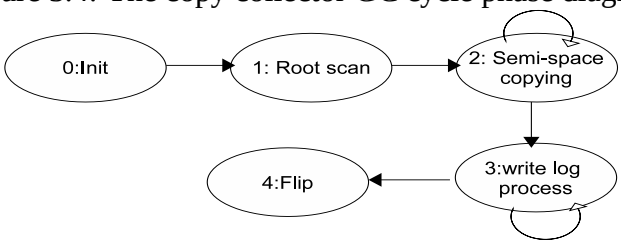
A GC cycle for this collector can be broken into five phases: *initialization*, *atomic root scan*, *semi-space copying*, *deferred write log processing* and finally *flip*. Each thread has its own deferred write log that stores information captured by the write barrier code executed by mutators. Some phases may be executed incremental, which is marked using a cyclic arrow. Figure 3.4 shows the copying collector's phase diagram, where the "semi-space copying" and "deferred write log processing" are made incremental.

For the *semi-space copying* phase, the atomic GC work is copying m bytes memory that takes t_m in time, and for the *deferred write log processing* phase, the atomic GC work is processing one thread's write log l that takes t_l in time. The other three GC phases execute atomically with processing time of t_{P_0}, t_{P_1} and t_{P_4}.

The cost coefficients needed are:

- δ_{r_1}: scan root set for references (μs/KB);

Figure 3.4: The copy collector GC cycle phase diagram



- δ_{r_2}: record one reference in root-scan phase (μs);
- δ_{c_1}: scan semi-space heap for references (μs/KB);
- δ_{c_2}: process a non-null reference in semi-space copy (μs);
- δ_{c_3}: object copying cost (μs/KB);
- δ_{wb}: record one "write" encountered by mutator (μs).[6]

With the set of copying GC cost coefficients, atomic states processing time and application-dependent parameters, GC cycle worst-case execution time is calculated:

$$C_{GC} = t_{P_0} + t_{P_1} + R \cdot (\delta_{r_1} + \theta_R \cdot \delta_{r_2}) + L \cdot (\delta_{c_1} + \theta \cdot \delta_{c_2} + \delta_{c_3}) + N \cdot t_l + t_{P_4} \tag{3.3}$$

The GC scheduling resolution ν_{GC} is no smaller than:

$$\nu_{GC} = max(t_{P_0}, t_{P_1}, t_{P_4}, t_m, t_l),$$

where $t_{P_1} = R \cdot (\delta_{r_1} + \theta_R \cdot \delta_{r_2})$.[7]

[6]Write barrier is a "tax" added to mutator execution by GC, therefore, with the worst-case write counts W and δ_{wb} the worst-case execution time $C_i = C_i^{orig} + W \cdot \delta_{wb}$.

[7]$theta_R$ is the maximum root set reference density.

3.5.2 $\texttt{SCHED}_{CC}$

GC task is scheduled as a traditional real-time task without explicit GC incrementality feature. We denote $\texttt{SCHED}_{CC} = (Po, \overline{\xi}_{RTGC})$ where Po considers two scheduling policies to schedule GC task as a periodic or an aperiodic task respectively.

Scheduling policy 1 (Po_1)

GC task is scheduled as a periodic task using rate-monotonic scheduler (RMS). Task synchronization is considered. The *generalized utilization bound theorem* of [69] shows that a set of n periodic tasks can be scheduled by RMS if:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + max\{\frac{B_i}{T_i}, \ldots, \frac{B_{n-1}}{T_{n-1}}\} \geq n(2^{1/n} - 1) \tag{3.4}$$

Assume that GC task has the highest priority, $T_{GC} \leq T_i$ for all mutator tasks in Γ_{N+1}, therefore, there is no blocking time for the GC task. Then:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_{GC}}{T_{GC}} + max\{\frac{B_i}{T_i}, \ldots, \frac{B_n}{T_n}\} \geq (n+1)(2^{1/(n+1)} - 1) \tag{3.5}$$

From Equation (3.5), GC task period is constrained by

$$T_{GC} \geq \frac{C_{GC}}{(n+1)(2^{1/(n+1)} - 1)} - \sum_{i=1}^{n} \frac{C_i}{T_i} - max\{\frac{B_i}{T_i}, \ldots, \frac{B_n}{T_n}\} \tag{3.6}$$

which is a necessary condition for the real-time task set Γ_{N+1} to be schedulable.

The value of T_{GC} needs also satisfy $T_{GC} \leq T_i$ to preserve that GC task has the highest priority in Γ_{N+1}. The scheduling and schedulability analysis above might not be able to find a feasible T_{GC}. In that case, the assumption that GC task has the highest priority could be removed, schedulability analysis considering the GC task's blocking time with all mutator tasks can find GC a necessary constrained condition for Γ_{N+1} to be schedulable if there is *any*.

If a feasible T_{GC} is found with Equation (3.6), the heap requirement H can be determined. The allocated heap will reach the maximum just before collector flips the two semi-spaces. Implicitly the worst-case time duration between the GC cycle start and GC cycle end (when flipping the semi-spaces) is its period T_{GC}. Therefore, the semi-space should be no smaller than maximum live memory plus the amount of memory allocated during the GC period T_{GC}. The heap requirement H is:

$$H \geq 2(L + \sum_{i=1}^{n} \lceil \frac{T_{GC}}{T_i} \rceil A_i) \tag{3.7}$$

Scheduling policy 2 (Po_2)

GC task is scheduled as an aperiodic task using polling server(PS), [5] using RMS. A polling server is defined by a three-tuple (P_{ps}, C_{ps}, T_{ps}) with the server priority P_{ps}, its maximum capacity C_{ps} and its period T_{ps}. The server is assumed to be launched at the initialization of the system. A worst-case response time analysis of τ_{GC} is extensively discussed in [5] and τ_{GC} can have arbitrary priority. If τ_{GC} has the highest priority using an RMS scheduler, in other words, $T_{GC} \leq T_i$ for all mutator tasks in Γ_{N+1}, the worst-case response time R_{GC} is constrained by:

$$R_{GC} \leq C_{GC} + (T_{ps} - C_{ps})Q \tag{3.8}$$

where Q is:

$$Q = \lceil \frac{C_{GC}}{C_{ps}} \rceil - \lceil \frac{(\lceil \frac{C_{GC}}{C_{ps}} \rceil - 1)C_{ps} + 1 - C_{GC}}{C_{ps}} \rceil$$

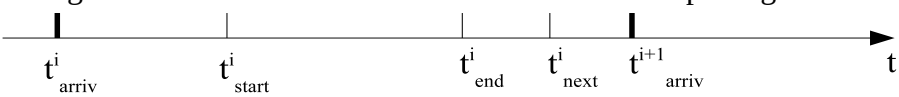
Before determining the heap requirement, the execution of the aperiodic task τ_{GC} by the polling server T_{ps} is characterized by $(t_{arriv}, t_{start}, t_{end}, t_{next})$ similarly to the definition in [5], where

- t_{arriv} is the GC triggering time meaning that GC is ready to execute. It is then waiting for

tasks with higher priorities than the polling server to complete. It may also wait for its server capacity to be available.

- t_{start} is the time when GC task starts its execution.
- t_{end} is the GC cycle completion time when GC flips the two semi-spaces.
- t_{next} is the next available time, *i.e.*, the earliest time at or after t_{end} when the polling server capacity is not zero, and when the CPU is available to serve the polling server(there is no higher priority tasks occupying the CPU). In other words, it is the time when GC task would be able to start its execution.

Figure 3.5: The GC execution characteristics with polling server



We need count the maximum amount of memory allocated by each real-time task during $[t_{start}, t_{next})$ to determine the heap requirement. It has been shown in [5] that:

$$\max_i(t^i_{next} - t^i_{start}) \leq \max_i(t^i_{end} - t^i_{arriv}) = R_{GC}$$

for all i, in other words, for all the GC cycles. The upper bound for the duration $[t_{start}, t_{next})$ is at most $R_{GC} - 1$. The heap requirement is constrained by:

$$H \geq 2(L + \sum_{i=1}^{n} \lceil \frac{R_{GC} - 1}{T_i} \rceil A_i) \tag{3.9}$$

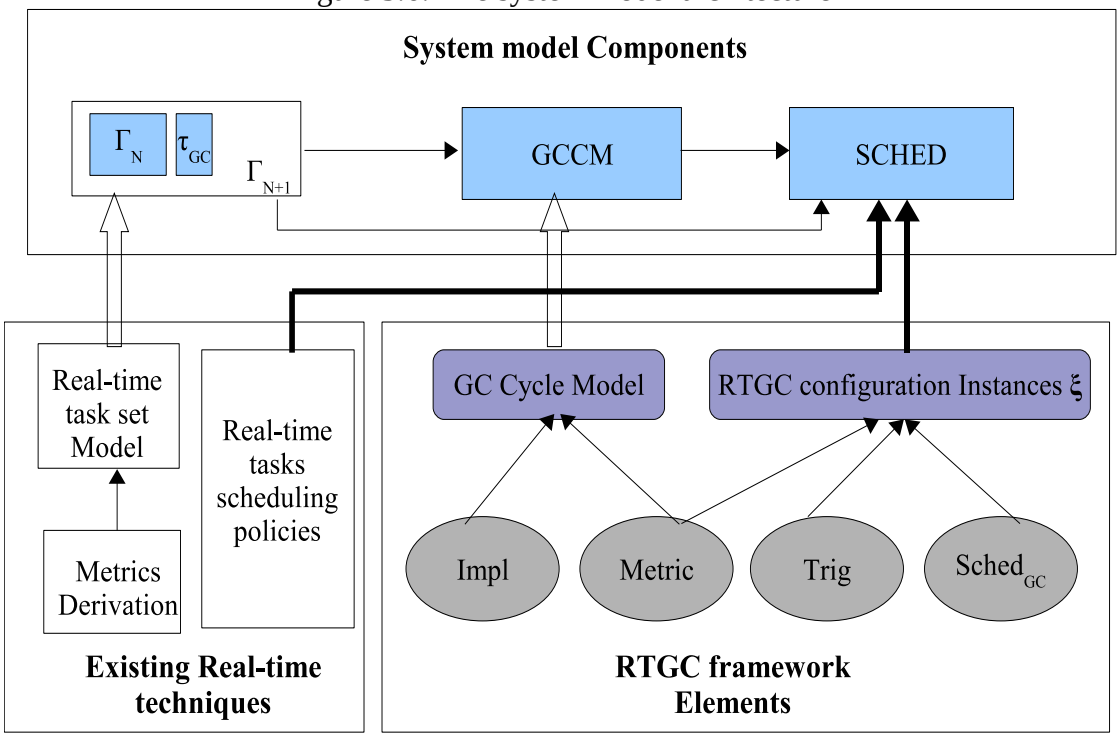
3.6 Summary

The RTGC framework breaks down the description of real-time garbage collectors into description of their implementation, metrics, triggering rules and scheduling rules. The framework captures

essential characteristics of real-time collectors that use incremental copying and mark-and-sweep techniques. The framework also underlies the idea of an RTGC configuration which describes the rules and parameters used for garbage collection in any particular real-time system.

The system model provides users a uniform way to model the GC-integrated real-time systems, where the GC is modeled by the RTGC framework. The elements in the RTGC framework and the components in the system model are dependent on each other as shown in figure 3.6.

Figure 3.6: The system model architecture



The RTGC framework models the garbage collector and provides interfaces to support the `GCCM` and `SCHED` components. These interfaces are: the GC cycle model that assists the generation of the GC cost model and the RTGC configuration that allows explicit control of GC interleaving with the mutators The existing real-time techniques including real-time task models and scheduling policies are included in the system model as well. Within the system model, data flows

from Γ_{N+1}, `GCCM` to `SCHED` to determine the GC scheduling parameters.

In the following chapters, a detailed `GCCM` and `SCHED` components modeled using the system model are presented and implemented for a mark-and-sweep collector. Our RTGC implementation supports the explicit GC incrementality for the RTGC-configuration based GC scheduling.

CHAPTER 4

MODELING GC COST

The existing research work on modeling GC cost is discussed in previous section 2.4.1. Compared to the Bengtsson, Henriksson, Kim, Bacon, and Goh approaches, our approach to modeling GC cost strives to separately model the effects of the GC implementation, the mutator implementation, the offered load from the environment, and the effects of scheduling on the total costs of GC. Like Bacon and Goh, the thesis work seeks to use a granularity just fine enough that the part of the model pertaining to the GC implementation (the GC cost coefficients) is stable across applications and workloads. The model also accounts for the cost of write barriers required by the incremental collection approach but executed by the mutators.

This chapter describes a controllable, incremental, real-time garbage collector that was implemented for Mono 1.1.16 [96], an open source implementation of CLI (Common Language Infrastructure) [39] [1]. A complete GC cost model was built for the implementation. The model consists of a set of GC cost coefficients and a set of *application-dependent parameters*. The GC cost coefficients are measured based on the GC implementation to achieve GC cost model accuracy. The application-dependent parameters that characterize the effects of mutator behavior are defined in a way that existing programming analysis techniques [27, 72, 80] can be directly used to provide the estimation.

A GC cost model should be pessimistic so that it does not promise what cannot be delivered.

[1] The Common Language Infrastructure (CLI) is the ECMA standard that describes the core of the .NET Framework world.

To be useful for real-time systems, a model's predicted WCET must be no less than the actual WCET. However, unreasonably large overestimates are also problematic as they may lead to the incorrect conclusion that the application cannot be feasibly scheduled. A *pessimism* metric of a model assesses the degree to which the model's predictions exceed the measured worst-case cost in particular situations. A pessimism of 1.0 corresponds to perfect estimates. Conservative estimates have larger pessimism. The experimental validation of our model suggests that removing low-frequency outliers from the measured costs of individual operations provides significantly less pessimistic overall cost estimates. While this approach is dubious for hard real-time systems, it's use in soft real-time systems improves schedulability in resource-constrained systems. For hard real-time, a more direct attack on the sources of the outliers will be needed.

The remainder of this chapter is organized as follows. Section 4.1 illustrates our mark-and-sweep *GC impl* upon which the GC cost model will be built. The mark-and-sweep GC implementation holds the RTGC characteristics in the RTGC framework. Section 4.2 presents a detail GC cost model for the mark-and-sweep *GC impl*. It calculates the GC-related cost, for instance, the worst-case GC cycle execution time C_{GC} for the GC task and write barrier cost for the mutator task. Section 4.3 extends the GC cost model with the real-time tasks. Section 4.4 illustrates the RTGC implementation that is adapted from Boehm-Demers-Weiser(BDW) GC [21] in Mono [96]. Our RTGC implementation supports the explicit GC incrementality. Experiments performed in section 4.5 show that the GC cost coefficients are stable across applications and workloads. To assist GC incrementality, atomic GC work for each GC phase is defined and calculated using the GC cost model. The atomic GC work is the minimum uninterruptible GC work in the GC phase. Section 4.6 shows how eliminating outlier measurements costs reduces the pessimism. The GC cost estimates are improved with adjusted worst-case cost coefficients, which covers over 99%th percentile measurements. The use of adjusted worst-case cost coefficients are studied and validated with the estimates of the GC cycle cost, GC increment costs and write barrier costs. Section

4.7 gives the summary.

4.1 The mark-and-sweep GC *Impl*

The mark-and-sweep GC implementation follows the RTGC characteristics discussed in the RTGC framework. The RTGC framework is also used to assist the build of a GC cost model.

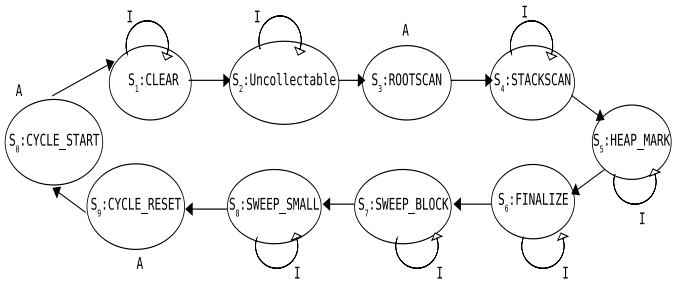
- For the GC impl, the RTGC framework identifies important characteristics and parameters of each GC component that have to be modeled and controlled in order to meet the goal of applying GC in real-time systems. A valid RTGC implementation needs to add three characteristics to the classical garbage collectors: incrementality, measurability and schedulability. For real-time systems, a complete garbage collection (a GC *cycle*) cannot be executed as one atomic action, while the mutator does nothing. Fine-grained interleaving of small units of garbage collection work (GC *increments*) with small units of real-time task execution is required.

- For the GC cost model, recall that the Real Time Garbage Collector (RTGC) framework introduced in chapter 3 contains four elements: *Impl*, *Metrics*, *Trig* and *Sched*. In general, a GC work-metric must be compatible with the Impl, Trig and Sched. That is, it must provide the information that Trig and Sched need to ensure that deadlines are met. The GC cost model presented in this chapter deals with a group of GC work-metrics. They are the number of references checked by thread stack scan, the amount of live heap scanned by heap marker and the amount of empty blocks reclaimed by heap sweeper *etc.* Decomposing these GC work-metrics into (1) the performance of collector operations and (2) the GC load offered by a real-time task generates a set of GC cost coefficients and a set of application-dependent parameters constituting the GC cost model. For instance, the work-metric *the number of references checked by thread stack scan*'s cost leads to the need for a cost coefficient *examine one reference in thread stack* and an application-dependent parameter *stack reference*

density. There are eventually used to determine the GC cost for the GC impl.

In this section, the Mark-and-sweep (MS) collector is used as the GC *Impl*. For which a GC cost model is built. To support incrementality, *all* phases that can take arbitrarily long have been incrementalized in the GC *Impl*, which makes the MS collector more nearly a real-time collector. A 2-level hierarchical allocation mechanism is used. as the allocator. It separately allocates large objects (bigger than one half of the allocator page size) and small memory objects.. Objects maybe made uncollectable by registering them with the collector. The root scan phase, before heap marking, is further divided into two phases: (nonthread)root scan and thread stack scan so that thread scan phase can be done incrementally. Thus, each GC cycle consists of ten phases illustrated in the phase diagram in Figure 4.1. To characterize GC incrementality, if a phase is atomically (uninterruptibly) executed, it is labeled A, otherwise, it is labeled I. Incremental phases contain multiple, controllable sub-phases.

Figure 4.1: GC phase diagram for the incremental mark-and-sweep collector



The ten GC phases are:

- GC_start(P_0): Initialization of a new GC cycle.
- GC_clear(P_1): Clear mark bits for all allocated heap blocks (I).
- GC_uncollectable(P_2): Mark uncollectable objects in allocated heap and push them on the

mark stack.[2] (I)

- GC_rootscan(P_3): Scan non-thread-stack roots including static, global and register data, and GC internal roots.

- GC_stackscan(P_4): Scan thread stacks to mark heap references reachable directly from the stacks. (I)

- GC_mark_heap(P_5): Scan heap memory and mark all accessible references starting from marked references in root set. (I)

- GC_finalization(P_6): Run registered finalizers on objects about to be collected. (I)

- GC_sweep_large(P_7): Reclaim unmarked large objects and entirely empty small-object blocks to the heap block free list; during this process, enqueue small-object blocks containing both marked and unmarked objects for processing in next phase. (I)

- GC_sweep_small(P_8): Reclaim unmarked small objects in the nonempty blocks, which are enqueued in P_7 phase, to the appropriate free list. (I)

- GC_reset(P_9): Reset global variables and (optionally) gather statistics data.

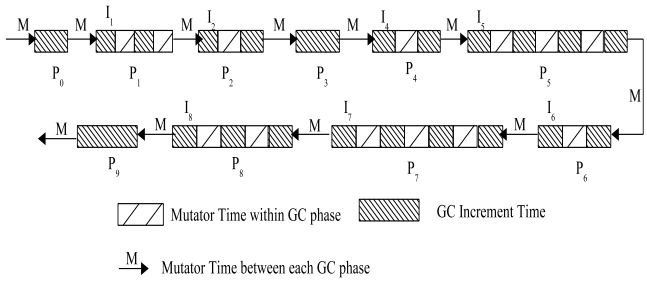
In order to enable non-trivial finalization in garbage collection, finalization should be carried out incrementally. However, non-trivial finalization, [60], supports application-specific finalization routines in addition to nulling disappearing links. It is hard to model arbitrary application-specific finalization routines. So for now, it is assumed that finalization is limited to nulling disappearing links. With this assumption, finalization phase P_6 is executed incrementally on the basis of processing one disappearing link.

[2]The allocator allows mutators to designate some allocated objects as uncollectable. These essentially become additional roots for marking.

RTGC incrementality: the mutator and GC execution model

Figure 4.1 lists the ten GC phases that constitute one GC cycle. Incremental phases contain multiple and controllable GC increment I_i, where $i = 1, 2, 4, 5, 6, 7, 8$. The interleaving of mutator and GC in time is illustrated in figure 4.2. The execution of GC phases P_0 through P_9 forms one GC cycle. GC increments I_i and atomic GC phase P_0, P_3, P_9 interleave with mutator execution.

Figure 4.2: RTGC incrementality for a mark-sweep collector



For the GC *Impl*, the worst-case GC cycle execution time C_{GC} is the basic GC cost calculation in the GC cost model. Since, when GC is integrated scheduled with real-time tasks, C_{GC} needs computed *a priori* to perform schedulability analysis. To assist GC incrementality(have explicit control over the interleaving between mutator and GC), atomic GC work for each GC phase is defined and calculated using the GC cost model. This will be extensively discussed in section 4.5.4 after presenting the detail GC cost model.

Barrier Algorithms

Section 2.3.1 in chapter 2 gives an overview of the barrier techniques to support the incremental garbage collector. In page (20) two tricolor invariants are introduced that were described in [81] as:

- *The weak tricolor invariant*: All white objects pointed to by a black object are reachable from some grey object through a chain of white objects that is Inv_1;

- *The strong tricolor invariant*: There are no pointers from a black pointer to a white object that is Inv_2.

Barrier techniques are divided into *snapshot-at-the-beginning* and *incremental-update* algorithms according to the two barrier invariants. For the mark-and-sweep GC *impl*, a write barrier needs trap reference updates when GC is executed in stack scan, P_4, and heap mark, P_5, to ensure that GC's view of data connectivity is consistent. Here the incremental stack scan allows one thread's stack scanned as one GC increment. A finer-granularity of stack scan is scanning one stack frame [3] as in [100, 101].

Two write barrier algorithms are used:

- Dijkstra's *incremental-update*, [36], write barrier algorithm: when hit, it "shade" the new reference grey if the new reference is white. The "shade" here means marking and pushing the references into a mark stack.

- Yuasa's *snapshot-at-the-beginning*, [99], write barrier algorithm: when hit, it "shade" the old reference grey if the reference being updated is not yet marked.

Yuasa's *snapshot-at-the-beginning* algorithm is not sufficient in order to support incremental stack scan, P_4. Considering the case:

(a) GC finishes scanning thread 1's stack TS_1 and switches to mutator execution;

(b) mutator pops out the old reference R_{old} to object obj that resides in thread 2's stack TS_2 and assigns a new reference R_{new} to obj that resides in thread 1's stack TS_1;

(c) GC resumes and scans thread 2's stack TS_2.

[3] A *stack frame* is allocated on the stack when a procedure is called and is removed upon return from the procedure. In general, the stack frame for a procedure contains all necessary information to save and restore the state of a procedure.

Now object obj is reachable through the local variable residing in thread 1's stack TS_1. If R_{new} is not recorded, GC will lose reachability to object obj, since thread 1's stack TS_1 has already been scanned (will not be scanned again in the current GC cycle). Therefore, when reference update occurs in stack scan phase, P_4, R_{new} needs "shade" the new reference in addition to Yuasa's write barrier.

4.2 GC cost model

The GC cost model supports RTGC measurability that is characterized in the RTGC framework in chapter 3. In principle, such a model could be built for any GC implementation. (Which is not to say that every GC is suitable for real-time use. Its cost model, should, however, make that clear.) Our goal of the GC cost model is separately characterizing (1) the performance of collector operations and (2) the garbage collection load offered by a real-time task. It allows prediction of system behavior from knowledge of component behavior and environmental specifications that is a step along the road to an engineering approach to garbage collection in real-time systems.

The details of a GC cost model depend on how GC-related work is carried out. For example, if some GC work is carried out directly by mutator the model needs to account for this work as part of the mutator's worst-case execution time. On the other hand, GC work that occurs outside the mutator thread can be modeled as a concurrent thread with its own worst-case execution time, latency and period. The complete GC cost model provides input to the real-time scheduling of the application, such as C_{GC} (the worst-case GC cycle execution time), ν_{GC} (the GC scheduling resolution introduced in page (45))*etc.* The GC implementation and the model may both be parameterized, for example to allow control of the uninterruptible incremental work activities undertaken by the GC. The GC cost model handles the implicit translation of controllable GC increments from GC work into CPU time. Thus, scheduling and schedulability analysis can be performed with the controllable GC incrementality that determines how mutator and GC are interleaved when GC cycle

is invoked.

The GC cost model has three parts:

1. A model of the GC implementation expressed as a set of constants and processing rate coefficients

2. A model of the way that mutators influence the amount of work done by the GC, expressed as a set of application-dependent parameters

3. A model of GC-related overhead incurred by the mutators, in particular the cost of barrier execution to allow the GC to execute incrementally

which will be addressed in turn.

4.2.1 GC cost coefficients

At the coarsest level the GC cycle worst-case execution time C_{GC} is needed for real-time scheduling analysis. A GC cycle execution time is the sum of all GC phases time, $C_{GC} = \sum_{i=0}^{i=9} T_{P_i}$. The execution time depends on both the GC implementation and applications' memory usage. For each incremental GC phase, there are one or more GC cost coefficients. Atomic GC phases take short constant time and are measured as a whole phase. There are no GC cost coefficients defined for atomic GC phases. GC cost coefficients for incremental phases and fixed costs for atomic phases are grouped below:

P_0: GC start

- T_{P_0}: fixed cost for atomic GC start phase (μs);

P_1: GC clear

- δ_C: Clear one block's mark bits (μs);

P_2: GC uncollectable

- δ_{U_e}: Get one allocated block and check whether it is uncollectable (μs);
- δ_{U_p}: Push pointers to all objects in one uncollectable block (μs);

P_3: GC root scan

- T_{P_3}: fixed cost for atomic GC nonthread stack root set scan phase (μs);

P_4: GC stack scan

- δ_{R_r}: Scanning rate of thread stacks for pointers ($\mu s/$Byte), which is the rate of fetching next available slot in thread stack to process;
- δ_{R_e}: Examine one pointer in thread stack (μs);
- $\delta_{R_{mp}}$: Mark and push a valid pointer to the main marking stack (μs);
- $\delta_{R_{bl}}$: Record a value in blacklist[4] to prevent it being misinterpreted as a valid pointer (μs);

P_5: GC heap mark

- $\delta_{M_{sr}}$: Scanning cost of an entry in main mark stack when marking the heap (μs/Byte);
- $\delta_{M_{hr}}$: Scanning rate of live heap for pointers (μs/Byte);
- δ_{M_e}: Examine one pointer in heap (μs);
- $\delta_{M_{mp}}$: Mark and push a valid object in heap to the marking stack (μs);
- $\delta_{M_{bl}}$: Record a value in the blacklist to prevent it being misinterpreted as a valid pointer (μs);

[4]Any main mark stack entry that fails the reference validity test is pushed onto an allocation blacklist to avoid allocation at addresses that are known to be subject to false retention.

P_6: GC finalize

- T_{fin}: fixed cost for atomic GC finalizing initialization sub-phase (μs);
- $\delta_{F_{dl}}$: process one disappearing link in heap (μs);

P_7: GC sweep large

- T_{sp}: fixed cost for atomic GC sweep initialization sub-phase (μs);
- δ_{S_r}: Scanning rate of heap to reclaim garbage for both P_7 and P_8 (μs/Byte).
- δ_{S_l}: Reclaim one empty block (μs);
- $\delta_{S_{enq}}$: Enqueue small object reclamation work into reclaim list to be processed in P_8 (μs);

P_8: GC sweep small

- δ_{S_s}: Reclaim unused small objects in one block (μs);

P_9: GC reset

- T_{P_9}: fixed cost for atomic GC reset phase to be ready for the next GC cycle (μs);

4.2.2 Application-dependent Parameters

The GC cost coefficients are attributes of GC implementation itself. Alone they are insufficient to characterize the GC costs for a particular application. Application behavior is incorporated into the model with a set of application-dependent parameters.

Application-dependent parameters are categorized into two groups: one group depends only on the application. The other depends on both the application and the GC execution pattern.

Group 1: Purely application-dependent parameters

- θ_t: upper bound on reference density in thread stack per byte;
- θ_h: upper bound on reference density in live heap per byte;
- θ_{h_v}: upper bound on valid reference density in live heap per byte;[5]
- θ_{dl}: upper bound on disappearing link density in live heap per byte
- L: maximum live memory size in bytes;
- U_{max}: maximum uncollectable memory in bytes during application execution.

Group 2: Application and GC execution pattern-dependent parameters

Several parameters depend on both the application and the pattern of GC executions.

- A_{max}: maximum memory allocated in bytes during application execution with GC;
- G_l: maximum garbage in bytes needing to be reclaimed during phase P_7;
- G_s: maximum garbage in bytes needing to be reclaimed during phase P_8.

The maximum memory GC needs to reclaim is the difference between maximum allocated memory and minimum live memory at any point of application's execution. A naive minimum live memory approximation is 0. Then the amount of garbage to reclaim is the maximum memory allocation A_{max}, which means $G_l + G_s \leq A_{max}$.

A_{max} depends on the GC execution patterns with the application. For instance, if GC is triggered when the amount of free memory becomes too small, then $H - F$ (F is the free memory

[5]Heap reference density is differentiated between reference density, θ_h, and valid reference density, θ_{h_v}, because the conservative BDW collector tests reference validity during application execution. Any main mark stack entry that fails the reference validity test is "blacklisted" to avoid false retention.

threshold to trigger GC and H is the heapsize) plus maximum allocation during one GC cycle execution is an approximation of the maximum allocated memory.

Modeling the interactions among the application, scheduler, and GC to derive values for A_{max}, G_l, and G_s is, at this time, an open problem. For the time being, there values are captured by sampling runs of the application with code instrumentation in the collector.

4.2.3 The calculation of C_{GC}

The most basic calculation supported by the model is that of C_{GC}, the actual processing time required to complete a GC cycle from the beginning of P_0 to the end of P_9. Besides the GC cost coefficients of Section 4.2.1 and the application-dependent parameters of Section 4.2.2, three global system-specific parameters are required:

- B: the block size in bytes
- M_t: the maximum thread stack size in bytes
- N_T: the maximum number of active threads in system.

The seven incremental GC phases P_1, P_2, P_4, P_5, P_6, P_7, and P_8 are calculated according to (4.1) through (4.7) below.

$$T_{P_1} = \frac{A_{max}}{B} \cdot \delta_C \tag{4.1}$$

$$T_{P_2} = \frac{A_{max}}{B} \cdot \delta_{U_e} + \frac{U_{max}}{B} \cdot \delta_{U_p} \tag{4.2}$$

$$T_{P_4} = M_t \cdot N_T \cdot \theta_t \cdot (\delta_{R_e} + max(\delta_{R_{mp}}, \delta_{R_{bl}})) + M_t \cdot N_T \cdot \delta_{R_r} \tag{4.3}$$

$$T_{P_5} = L \cdot \theta_h \cdot (\delta_{M_e} + \delta_{M_{mp}}) + L \cdot \theta_{h_v} \cdot (\delta_{M_e} + \delta_{M_{bl}} + \delta_{M_{sr}}) + L \cdot \delta_{M_{hr}} \tag{4.4}$$

$$T_{P_6} = T_{fin} + L \cdot \theta_{dl} \cdot \delta_{F_{dl}} \tag{4.5}$$

$$T_{P_7} = T_{sp} + M \cdot \delta_{S_r} + \frac{G_l}{B} \cdot \delta_{S_l} \tag{4.6}$$

$$T_{P_8} = \frac{G_s}{B} \cdot (\delta_{S_s} + \delta_{S_{enq}}) \tag{4.7}$$

Equations (4.1) through (4.7) combine to give Equation (4.8) for the worst-case GC cycle execution time.

$$C_{GC} = \sum_{i=0}^{i=9} T_{P_i} = (T_{P_0} + T_{P_3} + T_{P_9}) + \tag{4.8}$$

$$\begin{aligned} M_t \cdot N_T \cdot \theta_t \cdot (\delta_{R_e} + max(\delta_{R_{mp}}, \delta_{R_{bl}})) + M_t \cdot N_T \cdot \delta_{R_r} + \\ L \cdot \theta_h \cdot (\delta_{M_e} + \delta_{M_{mp}}) + L \cdot \theta_{h_v} \cdot (\delta_{M_e} + \delta_{M_{bl}} + \delta_{M_{sr}}) + \\ L \cdot \delta_{M_{hr}} + \frac{A_{max}}{B} \cdot (\delta_C + \delta_{U_e}) + \frac{U_{max}}{B} \cdot \delta_{U_p} + \\ T_{fin} + \delta_{F_{dl}} \cdot \theta_{dl} \cdot L + \\ T_{sp} + H \cdot \delta_{S_r} + \frac{G_l}{B} \cdot \delta_{S_l} + \frac{G_s}{B} \cdot (\delta_{S_s} + \delta_{S_{enq}}) \end{aligned}$$

Equation (4.9) is used to calculate sweeping cost instead of Equations (4.6) and (4.7) by substituting cost coefficients both δ_{S_l} and $\delta_{S_s} + \delta_{S_{enq}}$ with $max(\delta_{S_s} + \delta_{S_{enq}}, \delta_{S_l})$.

$$T_{P_{sweep}} = T_{S_7} + T_{S_8} \tag{4.9}$$

$$\leq T_{sp} + M \cdot \delta_{S_r} + \frac{G_s + G_l}{B} \cdot max(\delta_{S_s} + \delta_{S_{enq}}, \delta_{S_l})$$
$$\leq T_{sp} + M \cdot \delta_{S_r} + \frac{A_{max}}{B} \cdot max(\delta_{S_s} + \delta_{S_{enq}}, \delta_{S_l})$$

A *pessimism* metric assesses the degree to which the model's predictions exceed the measured worst-case cost in particular situations. The model's *pessimism* for a cost m, denoted by ρ^m is the ratio between the calculated and measured worst-case values of that cost. To be correct, the pessimism for each modeled cost should be greater than 1; to be useful, the pessimism should be as close to 1 as possible. For instance, $\rho^{C_{GC}} = \frac{C_{GC}}{\widehat{C}_{GC}}$ is the pessimism of the calculated C_{GC}. $\rho^{C_{GC}} > 1$ is necessary for a valid C_{GC} calculation.[6]

4.2.4 Write Barrier Cost

The non-moving mark-and-sweep GC *Impl* needs mutator pointer updates protected by a write barrier when GC is in incremental thread stack scan phase P_4 or heap marking phase P_5. Barrier algorithms are discussed in section 4.1.

Algorithm 1 demonstrates the implementation of a four-step *snapshot-at-the-beginning* write barrier, [99]. The algorithm follows closely to the write barrier implemented by Goh, [52].

Algorithm 2 demonstrates the implementation of a three-step *incremental-update* write barrier, [36]. Whenever there is a pointer update, the writer barrier implemented using Algorithm 1 or

[6]For the value of a metric m, $\widehat{m}$ is the measured value of the metric.

Algorithm 1 Algorithm 1: write-barrier mutation using modified Yuasa's snapshot-at-the-beginning for MS collector

Require: *This function is called upon reference update* $r_{old} = r_{new}$

```
1: if GC_state is P_4 or P_5 then
2:     if r_old is NOT marked then
3:         push r_old into mark stack
4:         if GC_state in P_4 then
5:             push r_new into mark stack
6:         end if
7:     end if
8: end if
9: r_old = r_new
```

Algorithm 2 is invoked by the mutator. Detail models of write barrier cost regarding the implementations are provided, which is not presented in [52] with the write barrier.

Algorithm 2 Algorithm 2: write-barriered mutation using Dijkstra's incremental-update for MS collector

Require: *This function is called upon reference update* $r_{old} = r_{new}$

```
1: r_old = r_new
2: if GC_state is P_4 or P_5 then
3:     if r_new is NOT marked then
4:         push r_new into mark stack
5:     end if
6: end if
```

Write barrier cost calculation

In order to calculate write barrier cost, first a set of write barrier cost coefficients is derived, where step numbers refer to Algorithm 1 and Algorithm 2:

- $\delta_{WB_{C_1}}$: Check GC status in lines (1) and (4) in Algorithm 1, and in line (1) in Algorithm 2;
- $\delta_{WB_{C_2}}$: Check whether the pointer is marked or not in line (2) in Algorithm 1, and in line (2) in Algorithm 2, where two consecutive pointer checking operations execute for both references r_{old} and r_{new};

- δ_{WB_P}: Push a pointer to the mark stack in line (3) and (5) in Algorithm 1, and in line (3) in Algorithm 2.

When write barrier execution is carried out by a mutator, the worst-case mutator execution time C_m needs to recalculated to include the write barrier overhead. A set of application-dependent parameters are defined before calculating the write barrier cost for mutator:

- W_{total}: the maximum number of total pointer updates during a complete mutator execution;
- W_r: the maximum number of pointer updates when GC is in P_4 during a complete mutator execution;
- W_m: the maximum number of pointer updates when GC is in P_5 during a complete mutator execution;
- W_p: the maximum number of old pointers in pointer update that are not marked during a complete mutator execution;
- W_q: the maximum number of new pointers in pointer update that are not marked during a complete mutator execution;

Worst-case extra execution cost C_{wb1} resulting from write barrier execution using Algorithm 1 is calculated (4.10).

$$C_{wb1} = \delta_{WB_{C_1}} \cdot W_{total} + \delta_{WB_{C_2}} \cdot (2W_r + W_m) + \delta_{WB_P} \cdot (W_p + W_r) \tag{4.10}$$

Worst-case extra execution cost C_{wb2} resulting from write barrier execution using Algorithm 2 is calculated (4.11).

$$C_{wb2} = \delta_{WB_{C_1}} \cdot W_{total} + 2\delta_{WB_{C_2}} \cdot (W_r + W_m) + \delta_{WB_P} \cdot W_q \tag{4.11}$$

Among the five parameters, W_{total} depends solely on the application implementation, the other four parameters have additional dependency on GC execution pattern. Approximation of W_r and W_m as functions of W_{total} is done in Equation (4.12) with the knowledge of maximum GC utilization U_{GC}^{max} calculated over a GC cycle. The number of pointer updates that are trapped by write barrier in line (1) in Algorithm 1 and Algorithm 2 depends on how often GC thread stack root scan P_4 and heap marking P_5 is ongoing within a GC cycle and how much CPU the GC utilizes during that GC cycle.

$$W_r = \frac{T_{P_4} \cdot U_{GC}^{max}}{C_{GC}} \cdot W_{total} \tag{4.12}$$

$$W_m = \frac{T_{P_5} \cdot U_{GC}^{max}}{C_{GC}} \cdot W_{total}$$

A naive worst-case estimation of W_p is $W_r + W_m$ comes from assuming that none of r_{old} in the pointer updates trapped by write barrier in line (2) in Algorithm 1 is marked, therefore, all trapped pointer updates from line (2) proceed to line (3). Similarly, W_q can be estimated using $W_r + W_m$ assuming that none of r_{new} in the pointer updates trapper by write barrier in line (2) in Algorithm 2 is marked, therefore, all trapped pointer updates from line (2) proceed to line (3).

Hence, the write barrier cost C_{wb1} for the mutator is recalculated:

$$C_{wb1} = \delta_{WB_{C_1}} \cdot W_{total} + (\delta_{WB_{C_2}} + \delta_{WB_P}) \cdot (2W_r + W_m) \tag{4.13}$$

$$= ((\delta_{WB_{C_2}} + \delta_{WB_P}) \cdot \frac{(2T_{P_4} + T_{P_5}) \cdot U_{GC}^{max}}{C_{GC}} + \delta_{WB_{C_1}}) \cdot W_{total}$$

And the write barrier cost C_{wb2} for the mutator is recalculated:

$$C_{wb2} = \delta_{WB_{C_1}} \cdot W_{total} + (2\delta_{WB_{C_2}} + \delta_{WB_P}) \cdot (W_r + W_m) \tag{4.14}$$

$$= ((2\delta_{WB_{C_2}} + \delta_{WB_P}) \cdot \frac{(T_{P_4} + T_{P_5}) \cdot U_{GC}^{max}}{C_{GC}} + \delta_{WB_{C_1}}) \cdot W_{total}$$

The worst-case execution time of the application including write barrier overhead is $C_m^{wb} = C_m + C_{wb}$, where C_m is the application's execution time without the existence of GC, C_m^{wb} is the application's execution time with the GC and C_{wb} is the write barrier cost calculated using Equation (4.13) or (4.14). With the definition of $\sigma^{wb} = \frac{C_m^{wb}}{C_m} = \frac{C_m + C_{wb}}{C_m}$, a real-time task's worst-case execution time including write barrier cost is recalculated using $\sigma^{wb} \cdot C_m$.

4.3 GC cost model with multiple real-time tasks

The GC cost model for the mark-and-sweep *GC impl* presented calculates the GC-related cost, for instance, the worst-case GC cycle execution time C_{GC} and write barrier cost for a single real-time task.

The GC cost model is extended with a task set Γ_N in this section considering both independent and dependent task cases. In page (43), real-time tasks are differentiated by independent or dependent based on whether there are precedence and synchronization constraints, [43] among tasks or not.

The GC cost model has three parts that model (1) GC implementation; (2) real-time workload; and (3) barrier cost incurred by mutator. Part (2) and (3) that are application-dependent are extended with real-time tasks and part (1) stays the same.

4.3.1 Application-dependent parameters for multiple tasks

Depending on whether there is precedence or non-CPU resources sharing, real-time tasks are characterized as independent or dependent.

Case 1: Independent real-time tasks

They do not share any non-CPU resources or have any precedences, each mutator thread works as if there is a private heap, which implies a need for the eight application-dependent parameters defined in section 4.2.2 for each mutator thread. They are

- θ_t^i: upper bound of reference density in real-time task τ_i's thread stack per byte;
- θ_h^i: upper bound of reference density in τ_i's live heap per byte;
- $\theta_{h_v}^i$: upper bound of valid reference density in τ_i's live heap per byte;
- θ_{dl}^i: upper bound on disappearing link density in τ_i's live heap per byte;
- L_i: maximum live memory in bytes of τ_i's heap;
- U_{max}^i: maximum uncollectable memory in bytes from τ_i;
- G_l^i: maximum garbage in bytes to reclaim in P_7 from τ_i;
- G_s^i: maximum garbage in bytes to reclaim in P_8 from τ_i.

The A_i, maximum allocation of τ_i is defined in the task model Γ_{N+1}.

The GC cost imposed by real-time task τ_i, denoted as C_{GC}^i, can be calculated in Equation (4.8) by substituting the application-dependent with the real-time task τ_i-dependent parameters.

$$C_{GC}^i = \sum_{i=0}^{i=9} T_{P_i} = (T_{P_0} + T_{P_3} + T_{P_9}) + \tag{4.15}$$

$$
\begin{aligned}
M_t \cdot N_T \cdot \theta_t^i \cdot (\delta_{R_e} + max(\delta_{R_{mp}}, \delta_{R_{bl}})) + M_t \cdot N_T \cdot \delta_{R_r} + \\
L_i \cdot \theta_h^i \cdot (\delta_{M_e} + \delta_{M_{mp}}) + L_i \cdot \theta_{h_v}^i \cdot (\delta_{M_e} + \delta_{M_{bl}} + \delta_{M_{sr}}) + \\
L_i \cdot \delta_{M_{hr}} + \frac{A_i}{B} \cdot (\delta_C + \delta_{U_e}) + \frac{U_{max}^i}{B} \cdot \delta_{U_p} + \\
T_{fin} + \delta_{F_{dl}} \cdot \theta_{dl}^i \cdot L_i + \\
T_{sp} + H \cdot \delta_{S_r} + \frac{G_l^i}{B} \cdot \delta_{S_l} + \frac{G_s^i}{B} \cdot (\delta_{S_s} + \delta_{S_{enq}})
\end{aligned}
$$

Therefore, C_{GC} for Γ_N when it is an independent task set, is calculated using $C_{GC} = \sum_{i=1}^{i=N} C_{GC}^i$, where C_{GC}^i is calculated in Equation (4.15).

Case 2: Dependent real-time tasks

They share non-CPU resources such as memory and may have precedences, such as synchronization using concurrency control protocols. Unlike independent real-time tasks working in a "private" heap, all mutator threads work in the same heap, which implies a need for the application-dependent parameters remaining defined for the whole application as in section 4.2.2. The calculation of the worst-case GC cycle execution is with Equation (4.8).

In section 3.4.1, calculating application's maximum allocation, A_{max} is given for independent and dependent tasks, where A_{max} is a function of real-time tasks' maximum allocations, A_i. A one-to-many producer-consumer dependency of real-time tasks is modeled and A_{max} is calculated using Equation (3.4.1) and (3.2). The maximum application allocation A_{max} with GC can be used to estimate maximum live memory L compensating the fact that maximum live memory at any point during an application's execution is statically undecidable.

4.3.2 Write barrier cost with multiple tasks

The worst-case execution time C_i needs recalculated to include write barrier overhead, which implies a need for the five application-dependent parameters defined in section 4.2.4 for each

mutator thread. They are

- W^i_{total}: the maximum number of total pointer updates within one invocation of real-time task τ_i;
- W^i_r: the maximum number of pointer updates when GC is in P_4 within one invocation of τ_i;
- W^i_m: the maximum number of pointer updates when GC is in P_5 within one invocation of τ_i;
- W^i_p: the maximum number of old pointers in pointer update that are not marked within one invocation of τ_i;
- W^i_q: the maximum number of new pointers in pointer update that are not marked within one invocation of τ_i;

Using the set of write barrier cost coefficients derived for the write barrier Algorithm (1) in page (76) and Algorithm (2) in page (76) in section 4.2.4, the worst-case write barrier cost using Algorithm (1), C^i_{wb1}, for real-time task τ_i is calculated as:

$$C^i_{wb1} = \delta_{WB_{C_1}} \cdot W^i_{total} + \delta_{WB_{C_2}} \cdot (2W^i_r + W^i_m) + \delta_{WB_P} \cdot (W^i_p + W^i_r) \tag{4.16}$$

The worst-case write barrier cost using Algorithm (2), C^i_{wb2}, for real-time task τ_i is calculated as:

$$C^i_{wb2} = \delta_{WB_{C_1}} \cdot W^i_{total} + 2\delta_{WB_{C_2}} \cdot (W^i_r + W^i_m) + \delta_{WB_P} \cdot W^i_q \tag{4.17}$$

Among the five real-time task-dependent parameters, W^i_{total} depends solely on the implementation of real-time task, τ_i; the other four parameters have additional dependency on GC execution

pattern. Same approximation of W_r^i, W_m^i, W_p^i and W_q^i is applied here as in section 4.2.4. They are:

$$W_r^i = \frac{T_{P_4} \cdot U_{GC}^{max}}{C_{GC}} \cdot W_{total}^i$$
$$W_m^i = \frac{T_{P_5} \cdot U_{GC}^{max}}{C_{GC}} \cdot W_{total}^i$$
$$W_p^i = W_r^i + W_m^i$$
$$W_q^i = W_r^i + W_m^i$$

where U_{GC}^{max} is the maximum GC utilization calculated over a GC cycle.

Thus, write barrier cost C_{wb1}^i and C_{wb2}^i for real-time task τ_i is recalculated as:

$$C_{wb1}^i = ((\delta_{WB_{C_2}} + \delta_{WB_P}) \cdot \frac{(2T_{P_4} + T_{P_5}) \cdot U_{GC}^{max}}{C_{GC}} + \delta_{WB_{C_1}}) \cdot W_{total} \tag{4.18}$$
$$C_{wb2}^i = ((2\delta_{WB_{C_2}} + \delta_{WB_P}) \cdot \frac{(T_{P_4} + T_{P_5}) \cdot U_{GC}^{max}}{C_{GC}} + \delta_{WB_{C_1}}) \cdot W_{total}^i \tag{4.19}$$

Similarly, write barrier overhead, σ_i^{wb}, for real-time task τ_i, is calculated using $\sigma_i^{wb} = \frac{C_i + C_{wb}^i}{C_i}$. A real-time task's worst-case execution time including write barrier cost is computed using $\sigma_i^{wb} \cdot C_i$.

When there are multiple tasks invoke the write barrier that accesses the mark stack, write barrier must be concurrency-safe. Lock-based synchronization is considered in the write barrier implementation.

4.4 Application of the RTGC cost model in Mono

The GC cost model built for the mark-and-sweep collector has been applied using Boehm-Demers-Weiser (BDW) GC library, [21], in Mono 1.1.16, [96], the Common Language Infrastructure virtual machine, [39]. The BDW collector was modified to improve its real-time abilities.

4.4.1 The BDW collector and Mono runtime platform

The BDW library, [21], is a freely available library that provides C and C++ programs with garbage collection capabilities. The algorithm it employs belongs to the family of mark and sweep collectors. The 2-level allocation mechanism and the ten phase GC cycle diagram of the BDW collector are given in section 4.1.

The BDW GC has also evolved to a 'mostly parallel GC' to reduce pause time by making the mark and sweep were partially incremental, [19]. The mark phase proceeds incrementally at memory allocation. A virtual memory protection scheme using the operating system *mprotect* system call realizes the write barrier. The sweep phase immediately reclaims memory blocks containing only garbage objects. It reclaims objects in partially unused blocks on-demand when a memory allocation for the same size cannot be satisfied incrementally.

The original stand-alone BDW collector is a conservative collector, which treats any word that it encounters as a potential pointer unless it can prove otherwise, [60]. In Mono, [96], the BDW GC is almost always supplied with precise pointer information. Hence, false retention due to pointer mis-identification is rare, which makes the BDW collector mostly-precise. In detail, Mono's scanning for different types of memory proceeds as follows, [96]:

- The heap (where managed objects are allocated) is currently handled in precise mode: the GC considers memory words that only contain heap references.
- Thread stacks and registers are always scanned conservatively. There are Mono runtime data structures involved that are implemented using C without type information.
- The static data area is scanned in precise mode with pointer information.

The Mono virtual machine provides a generalized GC interface so that different GC implementations can be accessed in a uniform way. Currently Mono has two GC engines: the BDW collector

and a compacting generational collector. This makes Mono a platform to study various RTGC implementations, for instance, a detail GC cost model for the compacting generational collector shall be built.

Fragmentation is a concern when using the BDW collector for real-time. However, this dissertation work aims at proposing a systematic RTGC approach rather than seeking a decent real-time GC implementation. Techniques that handles fragmentation may apply, for instance, in Metronome, [7], the mostly non-moving collector moves objects to remedy fragmentation when the number of free pages falls below a preset threshold.

4.4.2 Collector modifications

We have modified the Mono GC implementation to facilitate modeling and measurement. There are three design goals of the GC modifications: moving GC work out of the mutator; making GC increments controllable through GC configuration parameters; and instrumenting GC work to build an accurate GC cost model. Additional GC implementation changes help to achieve a more accurate cost model.

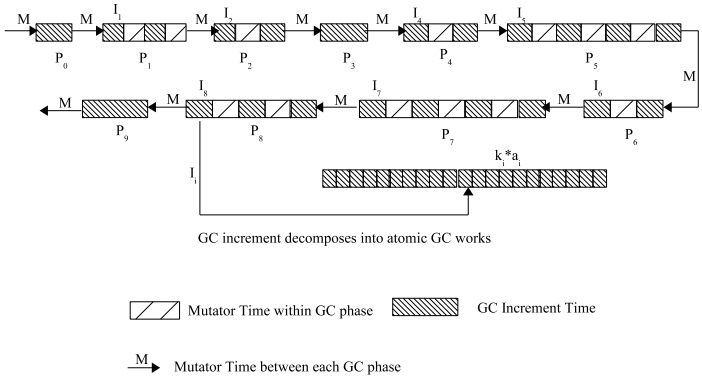
Mostly-parallel BDW's incrementality is insufficient because it does not incrementalize *all* phases that can take arbitrarily long. GC work in our modified collector is carried out in a single Mono run-time thread. A *GC cycle* consists of ten phases. Fine-grained interleaving of small units of garbage collection work (GC *increments*) with small units of real-time task execution is required in order to support real-time applications. Modifications therefore include incrementalizing all such phases as shown in figure 4.1 in page 66.

GC incrementality knobs k_i are specified for the GC increments I_i, where $i = 1, 2, 4, 5, 6, 7, 8$. The knobs control the size of uninterruptible GC work I_i in each incremental GC phase P_i in terms of the amount of atomic GC work a_i. The GC incrementality knobs provide us explicit ability to control GC execution, in other words, it supports the RTGC-configuration based GC scheduling

given in the system model in section 3.3.3 in chapter 3.[7]

Figure 4.3 shows the relationship among GC increments I_i, atomic GC work a_i and the incremental GC phase P_i in a GC cycle.

Figure 4.3: GC cycle, GC phase, GC increment and Atomic GC work



In order to achieve an accurate GC cost model, implementation modifications are made to BDW collector. Heap marking is simplified as scanning main mark stack and calling a uniform heap marking function[8] on each mark stack entry until stack becomes empty.

The modifications presented here follow closely the directions established by Goh in modifying the BDW collector for Mono 0.25, [52]. The major differences are that additional phases have been made incremental – particularly, uncollectable object marking, finalization, and sweeping small objects. Exposure of the GC interface in the more recent Mono allowed the write barrier to be more transparently integrated into the system. Our modifications support scheduling GC work using RTGC configurations. In section 3.1 in chapter 3, the modified RTGC was described using the RTGC framework.

[7]Experimental study of the RTGC configuration based GC scheduling algorithms and the schedulability analysis will be chapter 5 using the set of GC incrementality knobs.

[8]Heap marking in the original BDW collector used in Mono contains all sorts of self-admitted ”ugly hacks” [20] to speed things up. In particular, it “avoids procedure calls on the common path, takes advantage of the mark descriptor encoding, and optionally maintains a cache for the block address to header mapping or prefetchs when an object is ”grayed”, *etc.*”. In order to have a predictable GC, in our RTGC implementation, these ”hacks” are disabled.

Write barrier implementation in Mono

In adapting Mono's non-moving BDW collector for real time, the write barrier implementation is given in Algorithm 1 in section 4.2.4. A lock-based synchronization is used to make write barrier concurrency safe.

In detail, this kind of write barrier makes the BDW stand-alone library unsuitable for use with general C/C++ code. However, in the context of Mono, the centralized runtime support for writes provides a hook where the necessary write barrier code can be installed to support arbitrary C# code. Using the write barrier (Algorithm 1) in Mono 1.1.16 involves addressing these kinds of updates:

- updates to Mono internal data structures such as MonoHash, MonoThread, *etc.* These are protected directly by installing write barrier code at the site of reference updates in Mono runtime code.

- low-level explicit reference updates: Mono provides a set of low-level user-callable BDW APIs. The ones containing reference updates are write-barrier protected by wrapping them with a call to the write barrier.

- CIL opcode reference updates: *stfld, stsfld, stelem.ref* and *stind.ref*[9] are the set of reference update opcodes in Mono. They are implemented using helper functions. In order to write-barrier protect them, a set of helper functions contain write barrier calls are invoked when GC is ongoing.

[9] *stfld* replaces the value of a field of an object with a given value; *stsfld* replaces the value of a static field with a value from the stack; *stelem.ref* replaces the value of the element at the supplied index in the one-dimensional array with the reference value pushed onto the stack and *stind.ref* stores value of type object's reference into memory at a specific address.

Table 4.1: Applications Description

Application	Description
GC-stress	A gc stress testing program written to test Mono's GC engine, which executes a function allocating lots of object and performing some trivial computation repeatedly.
GCbench	This application is adapted from one written by John Ellis and Pete Kovac of Post Communications. It was modified by Hans Boehm of Silicon Graphics.It was ported to C# by Daniel Spoonhower of Carnegie Mellon University. It allocates and collects balanced binary trees of various sizes.
BinaryTree	This is our implemented sample application. It creates Binarytree with a specified depth, performs tree node checking computation and collects the tree repeatedly. In section 4.2.1, 4.2.2 and 4.2.3 16 is used as binary tree depth input.
Graph	This is our implemented sample application. It constructs and restructs a directed graph with a specified number of nodes. In section 4.2.1, 4.2.2 and 4.2.3 2000 is used as the number of nodes.

4.5 Experiments

The experiments described in this section focus on showing that the part of the GC cost model pertaining to the GC implementation (the GC cost coefficients) is stable across applications and workloads. Another goal of the experiments is to determine the *pessimism* of the model's worst-case GC cost predictions.

All the experiments were conducted on a PC (IBM Thinkpad T60) with a Genuine Intel(R) CPU T2400 at 1.83GHz and 1.5GBytes physical memory running Linux 2.6.17-gentoo-r4 with real-time kernel patch-2.6.17-rt8. The real-time kernel provides a high resolution timer. SMP is disabled for better real-time kernel stability. Physical heap memory is locked using *mlock()* to prevent paging. For the experiments, 64M memory is locked during Mono runtime start-up by setting the *GC_INITIAL_HEAP_SIZE* environment variable to 64. Runlevel 1 (single user mode) was used to minimize kernel activity influences. All cost coefficient rates have units of μs per byte and all other cost coefficients have units of μs. The CPU time stamp counter is read using the *rdtsc* instruction in instrumented code to obtain CPU cycles consumed for each cost coefficient, [1]. Table 4.1 describes the applications used in the experiments.

4.5.1 Measuring GC cost coefficients

Table 4.2 presents measured worst-case GC cost coefficients of processing rates (μs/Byte). Table 4.3 and 4.4[10] presents measured worst-case GC cost coefficients of processing units (μ). These

[10]"CPU_C" is short for "CPU cycles".

cost coefficients are defined in section 4.2.1.

Table 4.2: Worst-case RTGC cost coefficients – processing rate

Application	P_4		P_5				P_7,P_8	
	δ_{R_r}		$\delta_{M_{sr}}$		$\delta_{M_{hr}}$		δ_{S_r}	
	CPU cycles/B	μs/B	CPU cycles/B	μs/B	CPU cycles/B	μs/B	CPU cycles/B	μs/B
GC-stress	27.64	0.01512	19.93	0.0109	36.38	0.0199	0.0500	0.000027345
GCBench	28.30	0.01548	17.18	0.0094	36.74	0.0201	0.0526	0.000028758
BinaryTree	28.59	0.01564	16.63	0.0091	36.19	0.0198	0.0598	0.000032750
Graph	28.30	0.01548	16.81	0.0092	37.29	0.0204	0.0620	0.000033971

Table 4.3: Worst case RTGC cost coefficients (a)

Application	P_1		P_2				P_4					
	δ_C		δ_{U_e}		δ_{U_p}		δ_{R_e}		$\delta_{R_{mp}}$		$\delta_{R_{bl}}$	
	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs
GC-stress	35101	19.193	28699	15.692	7194	3.933	29205	15.969	27068	14.807	2552	1.395
GCBench	35981	19.675	28820	15.758	6864	3.753	28959	15.842	26540	14.519	2464	1.347
BinaryTree	36982	20.222	28996	15.834	6512	3.561	29880	16.346	28816	15.764	2453	1.341
Graph	34507	18.869	28820	15.758	6622	3.621	29101	15.920	26470	14.480	2508	1.371

Table 4.4: Worst case RTGC cost coefficients (b)

Application	P_5						P_6		P_7, P_8					
	δ_{M_e}		$\delta_{M_{mp}}$		$\delta_{M_{bl}}$		$\delta_{F_{dl}}$		δ_{S_l}		δ_{S_s}		$\delta_{S_{enq}}$	
	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs
GC-stress	32494	17.768	30668	16.769	2574	1.407	1139	0.623	35123	19.205	29513	16.138	4070	2.225
GCBench	32989	18.039	32340	17.683	2474	1.353	1254	0.686	35189	19.242	33429	18.279	5269	2.881
BinaryTree	32186	17.598	32131	17.568	2274	1.244	1111	0.607	36477	19.955	33986	18.592	5071	2.772
Graph	34320	18.766	29700	16.240	2650	1.450	1098	0.601	37356	20.426	30877	16.884	4895	2.678

Each coefficient takes to be the maximum value of all the measurements encountered during 100 application executions for the coefficient operations defined in section 4.2. Table 4.2, 4.3 and 4.4 show that the GC cost coefficients are stable across these applications with less than 1.6% difference. The scanning rate of the live heap $\delta_{M_{hr}}$ is slightly larger than scanning rate of thread stack for pointers δ_{R_r} shown in table 4.2 due to different heap and stack scanning implementation.

We also conducted another experiment on the *BinaryTree* application with various workloads by building different depth of trees. Table 4.5 and 4.6 display a subset of the cost coefficient measurements and shows that they are stable across various workloads for the application with less than 1.8% difference.

Table 4.5: Worst case RTGC cost coefficients for *BinaryTree*

Workload	P_5						P_6		P_7, P_8					
	δ_{M_e}		$\delta_{M_{mp}}$		$\delta_{M_{bl}}$		$\delta_{F_{dl}}$		δ_{S_l}		δ_{S_s}		$\delta_{S_{enq}}$	
(Tree Depth)	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs
15	32098	17.550	31098	17.012	2178	1.191	1100	0.601	36289	19.842	34199	18.700	4167	2.280
16	32186	17.598	32131	17.568	2274	1.244	1111	0.607	36477	19.955	33986	18.592	5071	2.772
17	31613	17.294	32472	17.754	2361	1.292	1166	0.638	37961	20.756	32417	17.725	5225	2.857
18	31989	17.500	32329	17.676	2112	1.155	1098	0.601	36381	19.902	30877	16.884	5095	2.787

Table 4.6: More Worst case RTGC cost coefficients for *BinaryTree*

Workload	P_1		P_2				P_4					
	δ_C		δ_{U_e}		δ_{U_p}		δ_{R_e}		$\delta_{R_{mp}}$		$\delta_{R_{bl}}$	
(Tree Depth)	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs
15	35465	19.401	27819	15.218	6812	3.726	28117	15.382	26068	14.260	2422	1.325
16	35981	19.675	28820	15.758	6864	3.753	28959	15.842	26540	14.519	2464	1.347
17	36011	19.700	28100	15.372	6797	3.718	29109	15.924	27109	14.839	2409	1.318
18	35787	19.577	28080	15.361	6812	3.726	29651	16.220	26891	14.711	2501	1.368

4.5.2 Measuring application-dependent parameters

Table 4.7 presents example application-dependent GC cost parameters obtained for applications that have no uncollectable objects (*i.e.*, $U_{max} = 0$ and $T_{P_2} = 0$).

Profiling code was included in our modified BDW collector in Mono to output the set of application-dependent parameters. For instance, upper bound on various metric density such as reference density in thread stack θ_t, reference density in live heap θ_h, valid reference density in live heap θ_{h_v} and disappearing link density in live heap θ_{dl} are derived by profiling the application execution to record the number of references in thread stack, the number of references and valid references in live heap and the number of disappearing link in live heap traced by GC in each GC cycle. The density parameter candidates are then calculated for each GC cycle and the maximum candidates are output as the density parameters of the application. A discussion on the current available techniques to derive some of these application-dependent parameters can be found in section 3.4 in chapter 3.

Table 4.7: Application-dependent GC cost parameters(number of references/byte, bytes)

Application	Application-dependent Parameters							
	θ_t	θ_h	θ_{h_v}	θ_{dl}	L	A_{max}	G_l	G_s
GC-stress	0.00548	0.06176	0.04269	0.000514	13456	28836544	28649352	99984
GCbench	0.00760	0.04175	0.04166	0.000002	8889592	28694472	21427480	155944
BinaryTree	0.00768	0.05885	0.04170	0.000002	9662424	28618320	25451472	160624
Graph	0.00580	0.15509	0.06611	0.000001	32587196	33974912	4080416	136816

Table 4.8: Atomic GC phase execution time (μs)

Application	Atomic GC phase name				
	GC_start (P_0)	Rootscan_nonthreadstack (P_3)	Finalizing initialization (in P_6)	Sweep initialization (in P_7)	Reset (P_9)
	T_{P_0}	T_{P_3}	T_{fin}	T_{sp}	T_{P_9}
GC-stress	3.44	32.39	10.11	87.21	55.10
GCbench	3.80	53.44	12.22	76.06	54.66
BinaryTree	3.49	50.04	12.27	61.49	63.67
Graph	3.68	66.57	10.75	86.90	52.92

4.5.3 Calculating C_{GC} using GC cost measurements

With the set of worst-case cost coefficients, atomic GC phase execution time and application-dependent parameters, the worst-case GC execution time C_{GC} is calculated with Equation (4.8) and compared to the actual measured worst-case value through the GC profiling. GC cost C_{GC}'s pessimism $\rho^{C_{GC}}$ is determined.

In these experiments, $B = 4096$ bytes and $M_t = 24$ kilobytes. The modified runtime system provides environment variable *MAX_THREAD_STACKSIZE* to specify the maximum thread stack size. By default, maximum stack size is the default Linux 2.6.17-rt8 maximum stack size, $8M$bytes. Before running an application, thread stack size analysis should be performed to set *MAX_THREAD_STACKSIZE* so that the application does not experience stack overflow. These applications are each single-threaded. In addition, there are three system threads: the Mono runtime thread, the Mono finalization thread and the RTGC thread. Thus, the total number of active threads in each example is $N_T = 4$.

Table 4.8 presents measurements of the three atomic GC phases and the atomic finalization, sweep initialization time for the example applications.

Table 4.9: The calculation of the worst-case $C_{GC}(\mu s)$

Application	Measured and calculated worst-case GC cycle execution time												
	$\widehat{T}_{P_1}$	T_{P_1}	$\widehat{T}_{P_4}$	T_{P_4}	$\widehat{T}_{P_5}$	T_{P_5}	$\widehat{T}_{P_6}$	T_{P_6}	$\widehat{T}_{P_{sweep}}$	$T_{P_{sweep}}$	$\widehat{C}_{GC}$	C_{GC}	$\rho^{C_{GC}}$
GC-stress	2484	146849	545	4236	378	9960	11.01	14.42	3651	140458	7184	301807	42.01
GCBench	2173	139096	512	3960	301162	13380464	14.25	24.42	3574	123621	306507	13647541	44.52
BinaryTree	2517	139044	495	3372	366049	14592596	14.36	25.40	3699	134980	372943	14872992	39.88
Graph	2358	169168	533	4122	1939567	191343008	12.43	30.34	2922	82772	1945509	191599368	98.48

Table 4.9 shows that the complete set of GC cost coefficients and application-dependent parameters predicts the worst-case C_{GC} with pessimism values in the range of approximately 40 to 100 for these applications.

4.5.4 Calculating the atomic GC work a_i cost

In section 4.4.2, the GC is made incremental and the GC incrementality is controllable by the GC incrementality knobs to set the *time-based* or *work-based* GC increment sizes. The GC increment is denoted as I_i, where $i = 1, 2, 4, 5, 6, 7, 8$ for each incremental phase.

- The time-based GC increment size expresses all the GC increments size in CPU time, T_{I_i}.
- The work-based GC increment size expresses the size of each GC increment, I_i, by giving the number, k_i, of the atomic GC work, a_i. The atomic GC work a_i takes t_i in time, therefore, the work-based GC increment size in time, t_{I_i}, is expressed as $t_{I_i} = k_i \cdot t_i$.

GC cycle, GC phase P_i and GC increment I_i are shown in figure 4.2 in page (66). Work-based GC increment I_i is decomposed into GC atomic work a_i, where $i = 1, 2, 4, 5, 6, 7, 8$ in figure 4.4. Time-based GC increment I_i size is the CPU time T_{I_i}.

Figure 4.4: GC increment vs. Atomic GC work

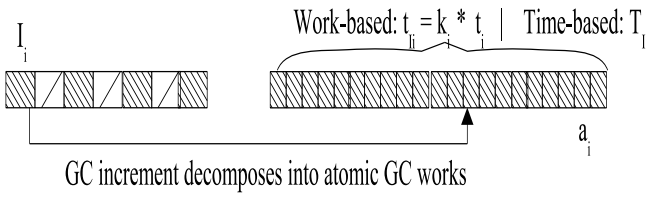


Table 4.10: The atomic GC work description and calculation

GC increment	**Atomic GC Work a_i**	**Description**	**Calculation formula for t_i**
I_1	a_1	Clearing one block in GC phase P_1	$t_1 = \delta_C^{wc}$
I_2	a_2	Processing one block in GC phase P_2	$t_2 = \delta_U^{wc} + \delta_P^{wc}$
I_4	a_4	Scan one thread stack in GC phase P_4	$t_4 = M_t \cdot \theta_t \cdot (\delta_{R_e} + max(\delta_{R_{mp}}, \delta_{R_{bl}})) + M_t \cdot \delta_{R_r}$
I_5	a_5	Process half block of live heap in heap marking GC phase P_5	$t_5 = 0.5 \cdot B \cdot (\theta_h \cdot (\delta_{M_e} + \delta_{M_{mp}}) + \theta_{h_v} \cdot (\delta_{M_e} + \delta_{M_{bl}} + \delta_{M_{sr}}) + \delta_{M_{hr}})$
I_6	a_6	Process one disappearing link in GC phase P_6	$t_6 = \delta_{F_{dl}^{wc}}$
I_7	a_7	Reclaim one empty block in GC phase P_7	$t_7 = \delta_{S_l^{wc}} + \delta_{S_{enq}^{wc}}$
I_8	a_8	Reclaim unused small objects in one nonempty block in GC phase P_8	$t_8 = \delta_{S_s^{wc}}$

Table 4.11: The atomic GC work estimates for *GCBench*

Time cost t_i (μs) of atomic GC work a_i	t_1	t_2	t_4	t_5	t_6	t_7	t_8
	20.22	19.52	990.00	311.52	0.686	22.12	18.28

What are the atomic GC work a_i? Table 4.10 lists the definitions for each atomic GC work a_i in each incremental phase I_i.

The atomic GC works a_i are calculated using GC cost coefficients and application-dependent parameters in table 4.10. As an example in table 5.1, the worst-case atomic GC works, a_i, are estimated for the application *GCBench* using the formulas provided in table 4.10, where the GC cost coefficients are given in table 4.2, 4.3 and 4.4 and *GCBench* application-dependent parameters are given in table 4.7.

4.5.5 Calculating Write barrier cost

Experiments were also done to obtain the write barrier cost coefficients and calculate the write barrier pessimism for all the applications. The same experimental setting is used as introduced at the beginning of this section. Each application contains a single real-time task that is mapped into

Table 4.12: The calculation of write barrier cost C_{wb} (μs, $\#/\mu s$)

Application	W_{total}	U_{GC}^{max}	$\frac{2T_{P_4}+T_{P_5}}{C_{GC}}$	C_m	$\widehat{C}_{wb}$	C_{wb}	$\widehat{\sigma}_{wb}$	σ_{wb}	r_M	$\rho^{\sigma_{wb}} = \frac{\sigma_{wb}}{\widehat{\sigma}_{wb}}$
GC-stress	401	0.583	0.161	3290105	56	11887	1.000018	1.0036	0.00013	1.0036
GCBench	1452115	0.635	0.981	2467593	1920789	133195528	1.778	54.98	0.588	30.92
BinaryTree	1591477	0.847	0.982	5313741	3239997	184974212	1.609	35.81	0.300	22.26
Graph	3244416	0.716	0.998	8001384	1558434	332479346	1.195	42.55	0.405	35.61

one Mono thread.

Three write barrier cost coefficients $\delta_{WB_{C_1}}$, $\delta_{WB_{C_2}}$ and δ_{WB_P} are defined in section 4.2.4 for the write barrier implemented with Algorithm 1. The worst-case write barrier cost coefficients are the maximum values observed among the complete sets of measurements giving $\delta_{WB_{C_1}} = 18.63\mu s$, $\delta_{WB_{C_2}} = 39.28\mu s$ and $\delta_{WB_P} = 78.06\mu s$ with number of CPU cycles 34055, 71804 and 142694 respectively. The write barrier cost coefficient measurements are stable across the applications with a less than 2.1% difference.

For C_{wb} and σ_{wb}, both measured and calculated values are given. r_M is the application's mutation rate calculated over a complete application execution. In table 4.12, write barrier overheads are measured and calculated using Equation (4.13).

Each application in table 4.12 represents a task, and a complete execution of the application is treated as one invocation. Recall that $\sigma_{wb} = \frac{C_m^{wb}}{C_m} = \frac{C_m + C_{wb}^i}{C_m}$ is the write barrier overhead, and a real-time task's write-barrier included worst-case execution time is calculated as $\sigma_{wb} \cdot C_m$, here, C_m is the worst-case execution time of the real-time task, C_{wb} is the write barrier overhead and C_m^{wb} is the worst-case execution time of the real-time task including write barrier overhead. Pessimisms of the calculated write barrier cost $\rho^{\sigma_{wb}}$ relative to actual measurement range from 1.0036 to 35.61 for these applications. The small write barrier overhead for the application *gc-stress* is due to the application's small mutation rate.

Table 4.13: The C_{GC} calculation with Γ_3 (μs, bytes)

Real-time tasks		C_i	C_i^{wb}	A_i	θ_t^i	θ_h^i	θ_{hv}^i	θ_{dl}^i	L_i	G_l	G_s
Γ_3	τ_1	894	895	3632	0.00242	0.0613	0.0522	0.000511	11832	25191672	87917
(Ind.)	τ_2	21262	36932	96520	0.00135	0.0897	0.0424	0.000002	31024	81720	516
	τ_3	14290	27008	15464	0.00244	0.0709	0.0547	0.000001	18200	32279	760
Calculation		$\widehat{C}_{GC} = 3068$		$C_{GC} = 324082$		$\rho^{C_{GC}} = 105.63$		$C_{GC}^{awc} = 63128$		$\rho^{C_{GC}^{awc}} = 20.58$	

4.5.6 A sample calculation of GC cost for multiple real-time tasks

Table 4.13 presents the data gathered to calculate a worst-case GC execution time C_{GC} with task set Γ_3 of three independent real-time tasks, where their periods $T_1 = 4000\mu s$, $T_2 = 50000\mu s$ and $T_3 = 40000\mu s$. Task τ_1, τ_2, τ_3 correspond to the application *GC-stress*, *BinaryTree* and *Graph* respectively in [46]. The write barrier costs for these applications are $\sigma_1^{wb}, \sigma_2^{wb}, \sigma_3^{wb}$ are 1.0001, 1.737, 1.890. Then $C_i^{wb} = \sigma_i^{wb} \cdot C_i$ is used to compute the task's worst-case execution time with GC write barrier overhead included. $\widehat{C}_{GC}$ is the measured worst-case GC execution time, C_{GC} and C_{GC}^{awc} are the estimates calculated with the worst-case cost coefficients and the adjusted worst-case cost coefficients covering 99.9% percentile according to [46]. $\rho^{C_{GC}}$ is the pessimism metric assessing the degree to which C_{GC} prediction exceeds the measured cost $\widehat{C}_{GC}$.

4.6 Improving GC cost estimates with adjusted cost coefficients

The worst-case cost coefficient rates are stable and consistent across various applications as in table 4.2. However, the observed cost coefficients in table 4.3 and 4.4, taken as the maximum of the observed values, are much larger than the vast majority of the observed values. In other words, the worst-case cost coefficients very rarely occur.

4.6.1 Examining GC cost coefficients measurements distribution

The worst-case cost coefficients rarely occur in each GC cycle. Similarly, if the GC increments have sizes that cover enough number of cost coefficient measurements, the worst-case cost coefficients rarely occur in each GC increment. There is no bursty distribution of the worst-case cost coefficient in GC increments.

Cost coefficients distribution in GC cycle

Figure 4.5 plots value distributions of the measurements for the clearing cost coefficient δ_C, rootscan cost coefficient $\delta_{R_{mp}}$, marking cost coefficient $\delta_{M_{mp}}$ and sweeping cost coefficient δ_{S_l}. The complete cost coefficients measurement set is gather for one GC cycle encountered by 100 application runs. The point (x, y) in each curve means that among the complete set of cost coefficient measurements, there are y percent having values larger than x.

Figure 4.5: A set of cost coefficient Measurement (CPU cycle #) vs. Percentage (%)

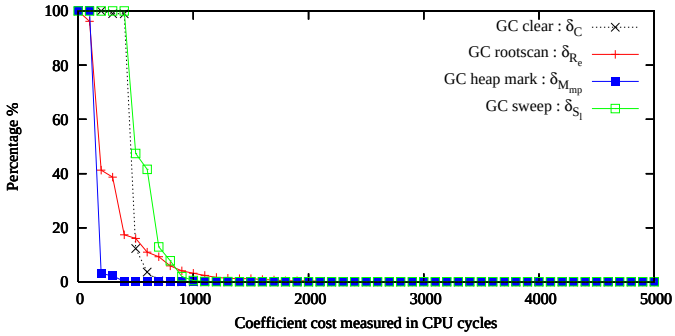
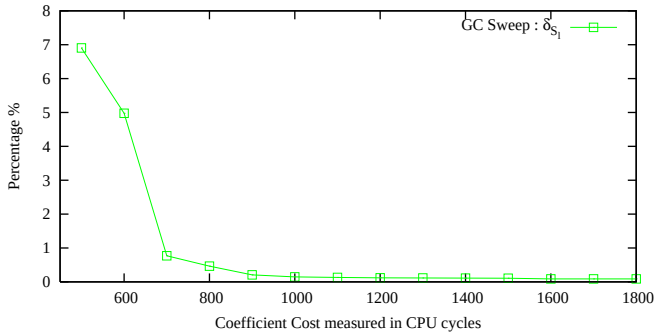


Figure 4.6 gives a closer look at the sweep cost coefficient measurement δ_{S_l} value distribution where y is between 0% and 8%. The extreme point $(1800, 0.107)$ in the plot shows that the fraction of measurements that are larger than 1800 reduces to only 0.107%. In other words, 1800 is the 99.9th percentile for cost coefficient δ_l of the complete measurement set. Comparing to the worst-case value of 35123 for δ_{S_l} in table 4.4, 1800 will give a less pessimistic calculation in the GC cost model.

Other cost coefficients are observed to have the similar distribution characteristic. Adjusted worst-case cost coefficient measurements are given in table B.2 and B.3 in the Appendix, where awc_1 and awc_2 are determined at 99.9th and 99.8th percentiles of the measurements respectively. These adjusted cost coefficients are also stable across these applications.

Figure 4.6: Sweeping Cost coefficient δ_{S_l} Measurement (CPU cycle #) vs. Percentage (%)

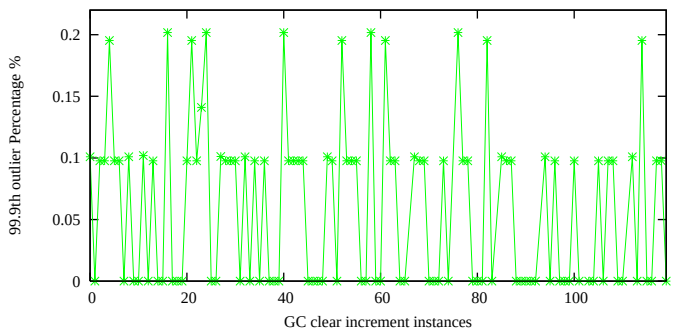


Cost coefficients distribution in GC increments

Using a set of GC cost coefficients measurements gathered in one GC increment, we calculate the fraction of measurements that are larger than the 99.9th percentile of the complete measurement set gather in one GC cycle in the previous section 4.6.1. (Adjusted cost coefficients, awc_1 determined by the 99.9th percentile of the measurements are reported in table B.2 and B.3 in the Appendix.)

Figure 4.7 plots the calculated fractions for GC clear cost coefficient δ_C observed over 100 GC clear increments. The GC clear increment is time-based with size of $T_{I_1} = 1000\mu$s. The maximum fraction is 0.22%, which means there is only at most 0.22% measurements larger than the 99.9th percentile for all these observed GC clear increments.

Figure 4.7: Cost coefficient δ_C Measurements Distribution in GC increments I_1: GC clear increment instances I_1 vs. Calculated fraction of the 99.9th percentile in I_1 (%) ($T_{I_1} = 1000\mu$s)



Other GC cost coefficients have similar distribution property. Figure B.1, B.2 and B.3 in the

Table 4.14: The GC cost coefficient distribution in GC increments

GC Increment	I_1	I_2		I_4			I_5			I_6	I_7		I_8
GC Cost Coefficient	δ_C	δ_{U_e}	δ_{U_p}	δ_{R_e}	$\delta_{R_{mp}}$	$\delta_{R_{bl}}$	δ_{M_e}	$\delta_{M_{mp}}$	$\delta_{M_{bl}}$	$\delta_{F_{dl}}$	δ_{S_l}	$\delta_{S_{enq}}$	δ_{S_s}
GC increment size T_{I_i} (μs)	1000	1000		2000			2500			500	1500		1500
Fraction of the 99.9th percentile	0.22	0.20	0.12	0.21	1.14	0.12	0.24	1.20	0.14	0.21	0.32	0.16	0.22

Appendix show similar distributions for δ_{U_e} with uncollectable increments, $\delta_{M_{mp}}$ with heap mark increments and δ_{S_l} with sweep large increments.

Table 4.14 lists their calculated fraction of measurements that are larger than the 99.9th percentile observed over the corresponding GC increments.

Experiments show that there is no bursty distribution of rarely occurred large outliers in GC increments and each GC increment may encounter rarely occurred large outliers (that is consistent across all GC increments).

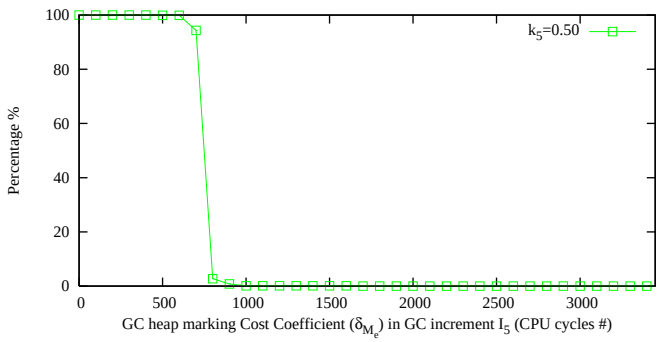
What magnitudes of percentile that the adjusted worst-case cost coefficients awc_1 in table B.2 and B.3 in the Appendix are when the complete set of measurements are derived from one GC increment? We define a *worst-case GC increment* to be an observed GC increment that takes the maximum time to execute the same amount of GC work. The percentile derived using a worst-case GC increment covers all other GC increments.

In figure 4.8, GC cost coefficient δ_{M_e} for the incremental GC_mark phase P_5 is measured using the work-based GC increment I_5 of size $k_5 = 0.5$. The measurement sets are gathered using the worst-case GC increments and the GC cycle as well. The case when the measurement set is gathered using the GC cycle is shown in figure 4.5. Again, the point (x, y) in each curve means that among the complete set of cost coefficient measurements (either using worst-case GC increments or GC cycle), there are y percentage measurements having values larger than the value of x.

Figure 4.8 plots value distributions of the measurements for the GC cost coefficient δ_{M_e} in heap

marking increment I_5 with size k_5: $0.5, 1, 1.25$ and 2[11]. The complete cost coefficients measurement set is gather for the worst-case GC heap marking increment encountered by 100 application runs. The point (x, y) in each curve means that among the complete set of cost coefficient measurements, there are y percent having values larger than x.

Figure 4.8: Cost coefficient δ_{M_e} Measurement using the worst-case GC increment I_5 (CPU cycle #) vs. Percentage (%)



The data show that the adjust worst-case cost coefficients awc_1 are approximately the 99th percentile of the measurements in the worst-case GC heap marking increment. Other cost coefficients have observed to have similar distribution characteristic using the worst-case GC increments as shown in figure B.4 and B.5 in the Appendix for δ_C and δ_{S_l}.

4.6.2 Pessimism of C_{GC} calculation

The calculated worst-case C_{GC} in table 4.9 has observed pessimism, $\rho^{C_{GC}}$, that ranges from 40 up to 100. The worst-case cost coefficients directly affect the calculation of C_{GC}. Using the adjusted worst-case cost coefficients, the worst-case GC execution time C_{GC} is recalculated using Equation (4.8) in table 4.15.

The resulting GC cost calculations are less pessimistic: $\rho^{C_{GC}}$, is less than 10 in all cases. Figure 4.9 shows a comparison of pessimism among the adjusted worst-case awc_1, the adjusted worst-case

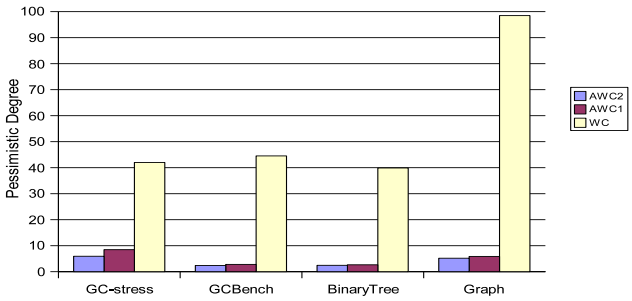
[11]Recall that the atomic GC work a_5 is processing half block of live heap in heap marking GC phase P_5. 0.5 stands for processing half block of live heap.

Table 4.15: The calculation of adjusted worst-case $C_{GC}(\mu s)$

Benchmark		Measured and calculated worst-case GC cycle execution time						
		T_{P_1}	T_{P_4}	T_{P_5}	T_{P_6}	$T_{P_7} + T_{P_8}$	C_{GC}	$\rho^{C_{GC}}$
gc-stress	Measured	2484	545	378	11.01	3651	7184	
	Calculated awc_1	3820	645	1158	13.21	55197	60917	8.48
	Calculated awc_2	2668	582	984	12.11	38174	42649	5.94
GCBench	Measured	2173	512	301162	14.25	3574	306507	
	Calculated awc_1	3745	742	813021	17.10	41798	859420	2.81
	Calculated awc_2	2649	716	706782	15.68	26637	736913	2.41
BinaryTree	Measured	2517	495	366049	14.36	3699	372943	
	Calculated awc_1	3790	668	938437	17.23	47376	990408	2.66
	Calculated awc_2	2650	540	880671	15.80	25080	918979	2.46
Graph	Measured	2358	533	1939567	12.43	2922	1945509	
	Calculated awc_1	4390	635	11393644	14.92	30475	11429274	5.87
	Calculated awc_2	3143	620	10078529	13.67	17028	10099459	5.19

awc_2 and the worst-case for the calculation of C_{GC}.

Figure 4.9: Pessimism $\rho^{C_{GC}}$ using different C_{GC} calculations



4.6.3 *Pessimism of GC increment cost calculation*

With the cost coefficient distribution characteristic observed in GC increments, GC increment estimates can be calculated with the adjusted worst-case cost coefficient, for instance, the adjusted worst-case cost coefficient set awc_1 in table B.2 and B.3. In figure 4.10, the GC increments are work-based with $k_1 = 3000$, $k_2 = 3000$, $k_4 = 2$, $k_5 = 6$, $k_6 = 600$, $k_7 = 2000$ and $k_8 = 2000$, where k_i is the number of atomic GC work a_i that form an uninterruptible GC increment[12].

In figure 4.10, the pessimism of the GC increments range from 1.24 to 1.81 that shows the adjust worst-case cost coefficients can be used to calculate the GC increment costs

[12]The variations of k_i are due to the definitions of the atomic GC works, a_i for each GC increments, I_i, given in table 4.10 in section 4.5.

Figure 4.10: Pessimism ρ^{I_i} of GC increments estimates ($i = 1, 2, 4, 5, 6, 7, 8$)

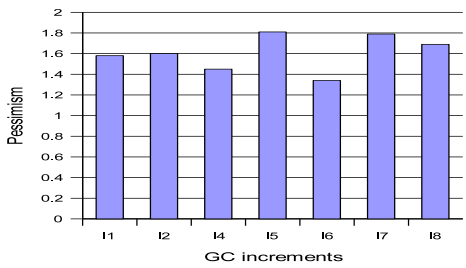


Table 4.16: The calculation of write barrier cost C_{wb} (μs, $\#/\mu s$)

Application	W_{total}	U_{GC}^{max}	$\frac{2T_{P_4}+T_{P_5}}{C_{GC}}$	C_m	$\widehat{C}_{wb}$	C_{wb}	$\widehat{\sigma}_{wb}$	σ_{wb}	r_M	$\rho^{\sigma_{wb}} = \frac{\sigma_{wb}}{\widehat{\sigma}_{wb}}$
GC-stress	401	0.583	0.161	3290105	56	321	1.000018	1.00002	0.00013	1.000018
GCBench	1452115	0.635	0.981	2467593	1920789	2888419	1.778	2.171	0.588	1.221
BinaryTree	1591477	0.847	0.982	5313741	3239997	3912040	1.609	1.737	0.300	1.079
Graph	3244416	0.716	0.998	8001384	1558434	7121232	1.195	1.890	0.405	1.582

4.6.4 Pessimism of write barrier cost calculation

During the experiments, write barrier cost coefficients, $\delta_{WB_{C_1}}$, $\delta_{WB_{C_2}}$ and δ_{WB_P}, have the similar distribution characteristics that the maximum of the observed values are much larger than the vast majority of the observed values across a variety of applications with various workloads. Adjusted write barrier cost coefficients are measured among 99.9%th percentile that $\delta_{WB_{C_1}} = 0.59\mu s$, $\delta_{WB_{C_2}} = 0.935\mu s$ and $\delta_{WB_P} = 1.311\mu s$ with number of CPU cycles 1080, 1710, 2398 respectively. Write barrier cost is recalculated for C_{wb} and σ_{wb} with the set of adjusted cost coefficients in table 4.16.

Table 4.12 and 4.16 shows that the write barrier cost estimates are improved from the maximum pessimism of 35.61 with the worst-case cost coefficients to 1.582 with the adjusted cost coefficients.

4.7 Summary

The real-time GC cost model described in this chapter factors the contributions of the GC implementation and the application-imposed workload. GC cost coefficients together with application-dependent parameters allow computation of the GC cycle time. Separation of these aspects is a step along the road to an engineering approach to garbage collection in real-time systems, allowing prediction of system behavior from knowledge of component behavior and environmental specifications.

The coefficients defined in this model have been found to be stable when measured across a variety of applications. The pessimism, however, is largely due to a small number of outlier measurements used in computing the coefficients. We believe, but cannot prove (yet), that this is due to variations in the execution by either the operating system or the hardware cache subsystem during the measurement process. The dramatic improvement in the pessimism suggests that effort to identify and remove this source of variability would be well worthwhile. A large overestimation of GC cost reduces the chance for a real-time application to pass a schedulability analysis, especially with constrained amount of resources. Thus, the derivation of adjusted worst-case GC cost is proposed to find a tighter estimate. Several experiments have been done to demonstrate such an improvement by eliminating corresponding outliers from cost coefficient measurements.

Thus far, the applications used to validate the model have been more along the lines of benchmarks than actual real-time applications. As a first application, the GridStat status router was re-implemented in chapter 6 to show the whole process of automated RTGC integration in the real-time systems.

The modified BDW collector for Mono is parameterized by a number of “knobs” that control the size of the uninterruptible increments in the various phases. The model coefficients and parameters described here were chosen to support scheduling GC and mutator activity at the level of these increments. The next chapter presents the scheduling rules and algorithms that can take

advantage of that capability.

CHAPTER 5

SCHEDULING MECHANISMS USING RTGC INCREMENTALITY

This chapter presents GC-included real-time scheduling and schedulability analysis using the RTGC configurations, which are a particular set of parameterized rules and parameter values for a collector. We propose two RTGC configuration plans: work-based ξ_W and time-based ξ_T; and illustrate the scheduling mechanisms with two types of real-time systems. They are (1) a guaranteed minimum mutator utilization, μ_{MMU} over time duration Δt; or (2) fixed-priority systems with specific deadline requirements respectively. The scheduling mechanisms derive constraints on the RTGC configuration plans' scheduling parameters and calculate the heapsize requirement H as functions of GC cost and the GC scheduling parameters.

In chapter 4, our RTGC implementation – the modified BDW collector in Mono supports the controllable RTGC incrementality. It allows users to configure GC trigger condition and specify GC increment sizes. The modified collector is parameterized by a number of "knobs" that control the size of the uninterruptible increments in various GC phases. These GC invocation configurations and GC increment size parameters are computed by the scheduling and schedulability analysis module, which is this section's primary focus. Meeting real-time timing requirements are relied on proper scheduling and schedulability analysis techniques, [70, 4, 6, 15, 33], with the underlying task model. The RTGC scheduling mechanisms presented in this section incorporate these existing scheduling techniques.

The remainder of this chapter is organized as follows: section 5.1 gives an overview of scheduling and schedulability analysis techniques. It also outlines the groups of RTGC scheduling mechanisms and the objectives of the RTGC scheduling mechanisms. Section 5.2 proposes two RTGC configuration plans to schedule GC work. Section 5.3 and 5.4 illustrate the algorithms associated with the RTGC scheduling using *MMU* and *Cspace* scheduling policies. Section 5.5 draws a conclusion for this chapter.

5.1 Scheduling preliminary

In our modified RTGC implementation, the GC work is carried out in a single Mono run-time thread, different from the mutator threads. This allows the integration of garbage collection into the current scheduling and schedulability analysis framework to see whether the GC-enabled system is schedulable prior to execution. This section gives the state-of-the-art in scheduling and schedulability analysis and outlines the objectives of the RTGC scheduling mechanisms.

5.1.1 Scheduling and schedulability analysis

Meeting real-time timing properties, such as deadline requirements has relied on proper scheduling and schedulability analysis techniques. We categorize the existing analysis techniques into: *analytical*, [15, 70, 6, 33, 4] and *model-based*, [48, 71, 97, 42, 44, 28, 13] according to the type and scope of the analysis.

1. *Analytical scheduling theory*

   This group is constituted by formal and informal notations that use the scheduling results for a periodic task model to statically assess timing requirements. The analysis usually deals with the deadline satisfaction requirements without taking programs' functionality into account.

2. *Model-based scheduling*

This group uses verification techniques that are based on state space exploration, such as reachability analysis, and model checking techniques, [29]. Compared to analytical approaches, model-based techniques are capable to solve a broader family of real-time applications by modeling more general timing satisfaction requirements than deadlines. Their hypothesis on the real-time task model are less restrictive than the analytical ones. Their major drawback is the fact that they suffer from state space explosion problem, thus the associated verification problem has high time and space complexity.

Model-based approaches can be further divided into two subcategories: *algebra-based* [44, 28, 13] and *automata-based* [48, 71, 97, 42]. The algebra-based approaches, for instance, *RTSL*, [44], use formal semantics and search for timing exceptions in the automatically generated finite state systems. The automata-based approaches model the problem (tasks and scheduling policy) in a general mathematical framework, *e.g.* timed automata with priorities, where schedulability is formulated as a property of the model and schedulability analysis checks whether the model satisfies the property or not.

Scheduling techniques can also divided into *deterministic* and *stochastic* scheduling, [73, 88], where stochastic scheduling is concerned with scheduling problems in which the processing times of tasks are modeled as random variables. Recall in chapter 3, task model Γ_{N+1} considered in the system model is a periodic or sporadic task model that is tailored to fit the GC-integrated real-time systems. Each task in Γ_{N+1} is characterized with $\tau_i = (\Phi_i, C_i, \sigma_i^{wb}, T_i, D_i, A_i, B_i)$ (see page 42 for detail). Our RTGC scheduling mechanisms are integrated with the analytical deterministic scheduling policies for priority-driven real-time systems.

Schedulability analysis of fixed-priority systems

Fixed-priority systems schedulability analysis has been widely studied in real-time literature. Many acceptance tests have been developed to guarantee real-time timing properties for a set of periodic

tasks. They fall in two classes: *sufficient tests*, [68, 63]) and *exact tests*, [64, 15, 70].

Sufficient tests usually have polynomial time complexity and can be used for on-line timing analysis for real-time applications when real-time tasks are admitted dynamically. However, providing only a sufficient schedulability condition may cause a poor processor utilization.

Exact (precise) tests provide a necessary and sufficient schedulability condition, but are too complex to be executed on-line for large task sets. It is because that many of these tests require the task response time to be calculated iteratively in the presence of higher priority tasks. Lehoczky *et al.*, [64], were among the first to propose the exact test for rate-monotonic analysis (RMA), [70], where a set of time points defined for schedulability analysis is verified that before each time point (in the set), the total computation for a given task set completes before that time point. Audsley *et al.*, [6], propose another original exact test for RMA, in which each task's worst-case response time R_i is calculated using an iterative formula that they derived. Schedulability analysis checks to make sure that each task's worst-case response time R_i is less than or equal to its deadline.s

5.1.2 RTGC scheduling mechanisms groups

The existing research work on RTGC scheduling is discussed in section 2.4.2 in chapter 2. In most approaches, GC work is carried out as an individual real-time task. It allows better control over GC, since GC can be scheduled together with real-time tasks so that the schedulability tests verify real-time predictability. Baker's approach, [8], is an exception to this strategy where GC work is incrementally invoked and executed upon mutator allocations.

In chapter 3, when scheduling GC as an individual task, `SCHED` component in the system model defines two ways to schedule GC using: (1) ξ_{GC} – the *RTGC configuration* based GC scheduling parameters and (2) $\overline{\xi}_{GC}$ – the classical real-time task's scheduling parameters, *i.e.*, T_{GC}, the period of the GC task; or R_{GC}, the worst-case GC response time[1]. The existing GC scheduling

[1]Recall that *RTGC configuration* describes the rules and parameters used for garbage collection in any particular real-time system.

approaches can be categorized into two groups according to `SCHED`:

1. Group 1 using ξ_{GC}, [7, 51]: Scheduling GC using explicit RTGC configuration instances. The RTGC configuration tells how GC increments are issued and how big GC increment sizes are in terms of GC work or CPU time, for instance, in [51], a fixed execution time quanta is assigned to each GC increment when GC cycles are triggered. Explicit RTGC configurations should be computed by integrated scheduling module with real-time task set. These GC scheduling parameters can be tuned with heap requirement and GC triggering rules during scheduling analysis.

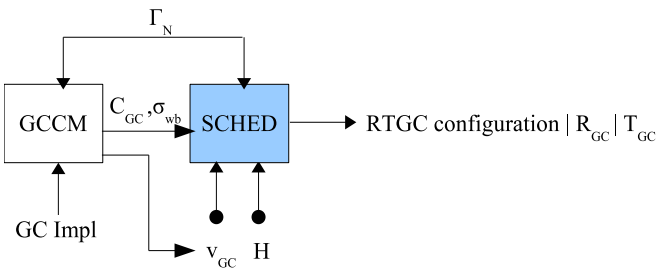
2. Group 2 using $\overline{\xi}_{GC}$, [54, 61, 83]: Scheduling GC with the traditional real-time task's scheduling parameters. The schedulers, earliest-deadline-first (EDF), sporadic server (SS) *etc.* are used among these approaches. The GC increments are issued implicitly by the real-time schedulers. These GC scheduling parameters can be tuned with heap requirement as well.

These two methods must schedule GC work together with real-time task set. The difference lies in the GC-mutator interleaving is determined by either explicit RTGC configurations or implicit real-time scheduler features. The explicit RTGC configurations supports a broader class of real-time applications, for instance, those applications where guaranteed low-latency (hence, the maximum mutator pause time) is critical, [2]. Examples include audio applications that need system keeping up with the audio streaming or network communication applications, such as high-speed data transfer, [10, 40], that need small time delay between receiving input data and responding to that data. Since GC increment sizes affect the maximum mutator pause time, using the explicit RTGC configurations, GC incrementality is computed to guarantee real-time low-latency requirements. This provides a tailored GC for real-time applications.

5.1.3 *The objective of RTGC scheduling mechanisms*

Recall that our overarching goal toward RTGC is to pursue the systematic approach of automating garbage collection configurations for a real-time system with its workload to meet platform-independent timing properties by analyzing platform-dependent timing-related memory-management behavior. It leads to two topics of interests: GC cost estimates, `GCCM` – the GC cost model, and GC scheduling mechanisms, `SCHED`. Figure 5.1 depicts the dependency between these two parts and shows the objective of our RTGC scheduling mechanisms – computing RTGC configuration parameters using the knowledge of the real-time system and GC implementation.

Figure 5.1: RTGC scheduling objective



With heap memory requirement H, `SCHED` takes the knowledge of real-time task set, Γ_N, the worst-case GC cycle execution time, C_{GC}, and GC scheduling resolution, ν_{GC} (estimates from `GCCM`) as inputs, and outputs GC scheduling parameters, such as GC cycle response time, R_{GC}, GC cycle period, T_{GC}, RTGC configuration instances with GC incrementality *etc.* Heap size, H is tunable to produce different GC scheduling parameters. (Tunable attributes have "black dots" associated with the arrow in the figure.)

5.2 RTGC scheduling mechanisms

A particular set of parameterized rules and parameter values is termed a *RTGC configuration* in section 3.2 in chapter 3, which determines the ways that an RTGC algorithm is controlled in a particular system. It characterizes the GC execution pattern. With GC scheduling using the rules in the RTGC configuration, schedulability analysis computes feasible parameter values in the RTGC configuration for a given task set Γ_{N+1} in the system model.

The modified RTGC implementation allows GC to be scheduled by work or by time. Work-based RTGC configurations specify the parameterized GC increment sizes using the set of GC incrementality knobs introduced in section 4.5.4. Time-based RTGC configurations issue GC increment in terms of CPU time.

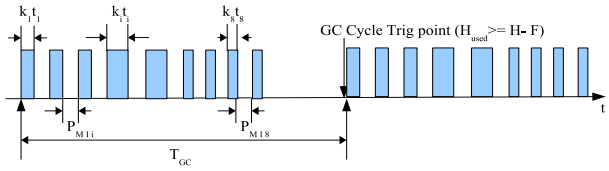
5.2.1 Work-based RTGC configuration

- *GC Trigger* - GC is triggered aperiodically when free heap memory drops below the preset threshold F. For this case, the minimum GC cycle inter-arrival time T_{GC} needs be derived.

- *GC Interleaving (with mutator)* - Whenever a GC cycle is triggered, GC increments and mutators are interleaved in a way that depends on which phase P_i GC is in. GC expends k_i amount of collection work according to the settings given by RTGC incrementality before yielding to the mutator and then mutator expends P_{MI_i} CPU time before yielding back to the GC. The interleaved execution pattern proceeds until the current GC cycle completes.

The GC execution pattern is shown in figure 5.3. The RTGC configuration parameters include T_{GC} and the set of k_i, where $i = 1, 2, 4, 5, 6, 7, 8$, which will be functions of heap memory requirement H, preset free memory threshold F and P_{MI_i} together with all other timing properties given in the task model Γ_{N+1}. Again, the derivation of these RTGC configuration parameters depends on the scheduling and schedulability analysis applied on the whole task set Γ_{N+1}.

Figure 5.2: Work-based RTGC configuration ξ_W



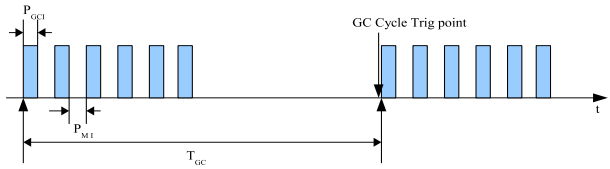
5.2.2 Time-based RTGC configuration

- *GC Trigger* - GC is triggered periodically with period T_{GC}

- *GC Interleaving (with mutator)* - Whenever GC cycle is triggered, GC increments and mutator are interleaved in a way that GC expends P_{GCI} CPU time before yielding to the mutator and then mutator expends P_{MI} CPU time before yielding back to the GC. The interleaved execution pattern proceeds until the current GC cycle completes.

The GC execution pattern is shown in figure 5.3. The RTGC configuration parameters include T_{GC} and P_{GCI}, which will be functions of heap memory requirement H and P_{MI} together with all other timing properties given in the task model Γ_{N+1}. The derivation of these RTGC configuration parameters depends on the scheduling and schedulability analysis applied on the whole task set Γ_{N+1}.

Figure 5.3: Time-based RTGC configuration ξ_T



5.2.3 GC Scheduling Resolution ν_{GC}

The scheduling and schedulability analysis computes the RTGC configuration parameters. The computation needs the knowledge of the GC scheduling resolution ν_{GC}. The GC scheduling resolution ν_{GC} is defined as the maximum GC execution latency due to the GC incrementality that imposes on mutator execution. It is the maximum execution time cost among all the atomic GC executions:

$$\nu_{GC} = MAX(T_{P_0}, T_{P_3}, T_{P_9}, T_{sp}, T_{fin}, t_1, t_2, t_4, t_5, t_6, t_7, t_8) \tag{5.1}$$

where T_{P_j} ($j = 0, 3, 9$), T_{sp} and T_{fin} are atomic GC phase execution time and t_i ($i = 1, 2, 4, 5, 6, 7, 8$) are the atomic GC work in incremental GC phases that are defined in table 4.10 in page 93. Some

Table 5.1: Calculating the GC scheduling resolution ν_{GC} for *GCBench*

Time cost t_i (μs) of atomic GC work a_i	t_1	t_2	t_4	t_5	t_6	t_7	t_8
	20.22	15.83	990.00	311.52	0.686	22.12	18.28

Atomic phase cost T_{P_j} (μs)	T_{P_0}	T_{P_3}	T_{P_9}	T_{sp}	T_{fin}
	3.80	53.44	12.22	76.06	54.66

of the atomic GC work a_i are application-dependent, for instance, the atomic GC work a_4 in incremental stack scan phase is *Scan one thread stack in GC phase*. The estimate of a_4's cost is application-dependent (depending on thread stack reference density *etc.*). Therefore, the GC scheduling resolution ν_{GC} is also application-dependent.

As an example, in table 5.1, the GC scheduling resolution ν_{GC} for the application *GCBench* is estimated to be 990μs due to t_4.

5.3 Work-based RTGC configuration schedulability analysis using *MMU*

The following two sections demonstrate scheduling and schedulability analysis with both the work-based, ξ_W and time-based, ξ_T RTGC configuration instances given in section 5.2. Different scheduling policies are used for these two RTGC configuration instances.

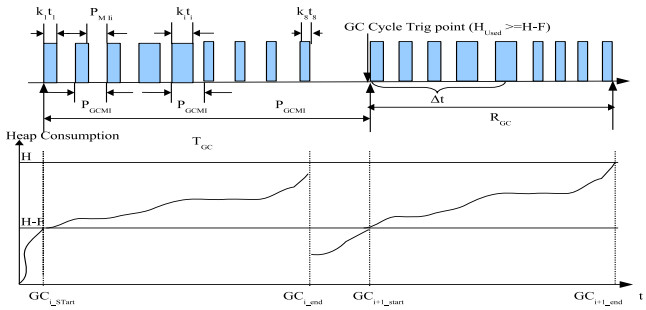
In this section, the work-based RTGC configuration instance, ξ_W is scheduled to meet real-time constraint that is specified by *minimum mutator utilization*(MMU) over a given time interval Δt, denoted as μ_{MMU}. It was defined by Cheng and Blelloch, [25], which is the minimum CPU utilization by the mutator over all intervals of width Δt. The minimum mutator utilization, μ_{MMU} is a function of window size Δt.

For example, play station game pad might require a response time of 50ms and the game pad controlling code might take 2ms to execute. It is sufficient to have a minimum mutator utilization of 4% at a granularity of 50ms. Therefore, for this game pad real-time application, the real-time constraint is to achieve $\mu_{MMU} = 4\%$.

5.3.1 RTGC configuration ξ_W scheduling parameters constraints

Besides the GC execution characteristics described in ξ_W, one constraint is added that $P_{MI_i} + k_i \cdot t_i = P_{GCMI}$, where P_{GCMI} is fixed for all i. The constraint simplifies the calculation of mutator utilization as a function of RTGC configuration scheduling parameters. Figure 5.4 illustrates the RTGC scheduling and the heap consumption, F is the preset free memory threshold triggering the GC cycles and H is the heapsize requirement.

Figure 5.4: Work-based RTGC configuration ξ_W with MMU (case 1)



Determining work-based GC increment sizes

During time duration Δt, there are $\lfloor \frac{\Delta t}{P_{GCMI}} \rfloor$ complete GC-mutator increment pairs. The mutator utilization over Δt can be calculated as

$$\mu(\Delta t) \geq \frac{\sum_i n_i \cdot (P_{GCMI} - k_i \cdot t_i) + \sum_j n_j \cdot (P_{GCMI} - T_{P_j})}{\Delta t} \tag{5.2}$$

Where $(\sum_i n_i + \sum_j n_j) \leq \lfloor \frac{\Delta t}{P_{GCMI}} \rfloor$.

Here, $i = 1, 2, 4, 5, 6, 7, 8$ represent the incremental GC phases and n_i is the number of GC-mutator increment pair $(P_{MI_i}, k_i \cdot t_i)$ in time interval Δt. $j = 0, 3, 9, sp, fin$ [2] represent the atomic GC phases and n_j is the number of GC-mutator increment pair (P_{MI_j}, T_{P_j}) in time interval Δt and $n_j \leq 1$, since these constant-time atomic GC phase executions occur only once in each GC cycle.

[2] $j = sp, fin$ represent the atomic GC phase T_{sp} and T_{fin}.

We define

$$\overline{kt}_{max} = \max_i(k_i \cdot t_i) \text{ and } \overline{kt}_{min} = \min_i(k_i \cdot t_i)$$

where k_i must be chosen to satisfy real-time properties during schedulability analysis. First they satisfy $\overline{kt}_{max} \geq \overline{kt}_{min} \geq \nu_{GC} \geq T_{P_j}$, where ν_{GC} is the GC scheduling resolution discussed in section 5.2.3 and $j = 0, 3, 9, sp, fin$ represent the five atomic GC phases. Then we have

$$P_{GCMI} - k_i \cdot t_i \geq P_{GCMI} - \overline{kt}_{max} \text{ and } P_{GCMI} - T_{P_j} \geq P_{GCMI} - \overline{kt}_{max} \text{ for all i,j}$$

The calculation in Expression (5.2) is simplified using the above expressions:

$$\mu(\Delta t) \geq \frac{\lfloor \frac{\Delta t}{P_{GCMI}} \rfloor \cdot (P_{GCMI} - \overline{kt}_{max})}{\Delta t} \geq \mu_{MMU} \tag{5.3}$$

$$\Longrightarrow \overline{kt}_{max} \leq P_{GCMI} - \frac{\mu_{MMU} \cdot \Delta t}{\lfloor \frac{\Delta t}{P_{GCMI}} \rfloor}$$

Determining the free memory threshold F

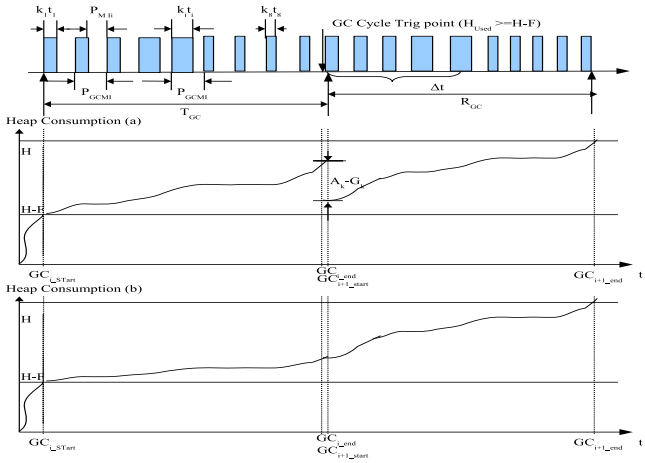
Using the work-based RTGC configuration, ξ_W, GC is triggered when free memory goes below the preset threshold F. A good choice of the free memory threshold F should avoid triggering GC when mutator executes in a state when it still keeps allocating without deallocating. In this case, GC is invoked without reclaiming any garbage, and CPU time is wasted.

An application's allocation patterns affect the choice of F. To be feasible F must be above the amount of free memory at the point when the application just enters its steady state. A real-time application's steady state is when the application enters its repeatable execution pattern, *i.e.*, the periodic invocations of the periodic task set during application's execution.

In figure 5.4, heap consumption is plotted by time with GC cycles. Every time GC is triggered, it reclaims a certain amount of garbage to bring the used memory below $H - F$. The next GC

cycle will not be triggered upon the completion of the first GC cycle, where $T_{GC} > R_{GC}$. In figure 5.5, heap consumption in (a) and (b) shows that GC is invoked collecting some garbage (or no garbage) and the next GC cycle immediately follows the completion of the previous GC cycle, where $T_{GC} = R_{GC}$. In this case, GC is invoked frequently while not reclaiming much garbage (there is no garbage reclaimed in figure 5.5 (b)) due to the inappropriate choice of F.

Figure 5.5: Work-based RTGC configuration ξ_W with MMU (case 2,3)



For the e-th GC cycle during application's execution, we define: M_e^G, the garbage reclaimed in e-th GC cycle, M_e^A is the mutator allocation in the duration of R_{GC} in e-th GC cycle, and E_{max} is the maximum total number of GC cycles during a complete application execution. Ensuring that applications never run out of memory require that $H - F + \sum_{e=1}^{E}(M_e^A - M_e^G) \leq H$ for any E ($1 \leq E \leq E_{max}$)). Thus, the choice of F should satisfy:

$$F \geq \sum_{e=1}^{E}(M_e^A - M_e^G) \text{ for all E that } 1 \leq E \leq E_{max} \tag{5.4}$$

Determining the heap requirement H

So far, we derive the constraints for the RTGC configuration ξ_W scheduling parameters: (i) the GC increment sizes need to satisfy Expression (5.3) and (ii) for GC triggering the preset free memory

threshold F needs to satisfy Expression (5.4). These constraints are a result of real-time timing requirement in terms of the minimum mutator utilization, μ_{MMU}, and the application's memory usage behaviors. Here, the heapsize requirement for RTGC configuration ξ_W is studied.

For the e-th GC cycle, let L_e be the live memory at the start of the cycle, FG_e be the floating garbage at the start of the cycle (it is generated from the previous $(e-1)$-th GC cycle), and M_e^A be memory allocated during the current GC cycle. The heap memory required during the cycle, H_e, satisfies: $H_e = L_e + FG_e + M_e^A$. The worst-case scenario for floating garbage is that all memory allocated during $(e-1)$-th GC cycle, M_{e-1}^A, becomes floating garbage of the current e-th GC cycle that $FG_e \leq M_{e-1}^A$.

Lemma 1. *For the e-th GC cycle, the amount of heap memory required H_e is bounded by the maximum live memory L_e^{max} at the start of the cycle, the maximum memory allocated in e-th GC cycle, $M_e^{A_{max}}$ and the maximum memory allocated in $(e-1)$-th GC cycle, $M_{e-1}^{A_{max}}$, [74].*

$$H_e \leq L_e + M_e^{A_{max}} + M_{e-1}^{A_{max}} \tag{5.5}$$

$$\Longrightarrow H = L + 2 \cdot A_{max}$$

since the maximum live memory during application execution, L, satisfies $L \geq L_e$ and the maximum memory allocation during GC cycles, A_{max}, satisfies $A_{max} \geq M_e^{A_{max}}$ and $A_{max} \geq M_{e-1}^{A_{max}}$.

With the knowledge of real-time tasks' allocation metrics, A_{max} is calculated as

$$A_{max} = \sum_{i=1}^{N} \frac{R_{GC}}{T_i} \cdot A_i \tag{5.6}$$

where R_{GC} is the worst-case GC cycle response time.

Let NI_i be the number of GC increments occurring in each incremental GC phase, and $\sum_{i=1}^{N} NI_i \cdot$

$k_i \cdot t_i + T_{P_0} + T_{P_3} + T_{P_9} + T_{sp} + T_{fin}$ constitutes one GC cycle execution time. Also let T_a^{sum} be the sum of all the atomic GC phase execution time, and $T_a^{sum} = T_{P_0} + T_{P_3} + T_{P_9} + T_{sp} + T_{fin}$. The worst-case GC response time R_{GC} is calculated as:

$$R_{GC} = \sum_{i=1}^{N} NI_i \cdot P_{GCMI} + T_a^{sum} \leq \frac{C_{GC}}{\overline{kt}_{min}} \cdot P_{GCMI} + T_a^{sum} \tag{5.7}$$

$$\text{where } \overline{kt}_{min} = \min_i (k_i \cdot t_i)$$

With R_{GC} satisfying Expression (5.7) and the heapsize calculated in Equation (5.3), the heap requirement H can calculated using:

$$H = L + 2 \cdot \sum_{i=1}^{N} \frac{\frac{C_{GC}}{\overline{kt}_{min}} \cdot P_{GCMI} + T_a^{sum}}{T_i} \cdot A_i \tag{5.8}$$

The heapsize H calculated using Equation (5.8) is a function of the RTGC configuration ξ_W scheduling parameters, for instance, P_{GCMI}, $\overline{kt}_{min}$ and GC cost estimate C_{GC} from `GCCM` component. On page 74, C_{GC} is calculated using Equation (4.8), which is a function of heapsize H with the GC cost coefficients and application-dependent parameters known *a priori*. It is rewritten as:

$$C_{GC} = \sum_{i=0}^{i=9} T_{P_i} = \Upsilon + \delta_{S_r} \cdot H \tag{5.9}$$

$$\text{where } \Upsilon = (T_{P_0} + T_{P_3} + T_{P_9}) +$$
$$M_t \cdot N_T \cdot \theta_t \cdot (\delta_{R_e} + max(\delta_{R_{mp}}, \delta_{R_{bl}})) + M_t \cdot N_T \cdot \delta_{R_r} +$$
$$L \cdot \theta_h \cdot (\delta_{M_e} + \delta_{M_{mp}}) + L \cdot \theta_{h_v} \cdot (\delta_{M_e} + \delta_{M_{bl}} + \delta_{M_{sr}}) +$$
$$L \cdot \delta_{M_{hr}} + \frac{A_{max}}{B} \cdot (\delta_C + \delta_{U_e}) + \frac{U_{max}}{B} \cdot \delta_{U_p} +$$
$$T_{fin} + \delta_{F_{dl}} \cdot \theta_{dl} \cdot L +$$
$$T_{sp} + \frac{G_l}{B} \cdot \delta_{S_l} + \frac{G_s}{B} \cdot (\delta_{S_s} + \delta_{S_{enq}})$$

5.3.2 *Algorithms of ξ_W scheduling and schedulability analysis using MMU*

Using Equation (5.9) where $C_{GC} = \Upsilon + \delta_{S_r} \cdot H$, the heapsize H is recalculated as

$$H = \frac{L + 2\sum_{i=1}^{N} \frac{A_i}{T_i} \cdot (T_a^{sum} + \frac{\Upsilon \cdot P_{GCMI}}{\overline{kt}_{min}})}{1 - 2\sum_{i=1}^{N} \frac{A_i}{T_i} \cdot \frac{\delta_{S_r} \cdot P_{GCMI}}{\overline{kt}_{min}}} \tag{5.10}$$

In Equation (5.10), the heapsize H monotonically increases with the value of P_{GCMI}, therefore, Algorithm 3 is proposed to find the minimum valid P_{GCMI}. It takes the inputs: a user-specified value of $\overline{kt}_{min}$ and the real-time timing property in terms of μ_{MMU} and Δt, and finds the minimum valid P_{GCMI}. The choice of $\overline{kt}_{min}$ needs to satisfy $\overline{kt}_{min} \geq \nu_{GC}$ and $\overline{kt}_{max} \geq \nu_{GC}$, where ν_{GC} is the GC scheduling resolution.

Algorithm (4) calculates the heapsize requirement H as a function of P_{GCMI}.

To summarize, when real-time timing property is specified with the minimum mutator utilization μ_{MMU} over the given Δt, GC scheduling using the RTGC configuration ξ_W in the task model Γ_{N+1} consists of three steps:

- Determine the GC incrementality parameters k_i using the constraint given in Expression (5.3);

Algorithm 3 ξ_W GC: Determine the minimum valid P_{GCMI} value with MMU requirements

Require: *This function is called upon a task set modeled in* Γ_{N+1} *and returns the minimum valid* P_{GCMI}

1: **double Min**_P_{GCMI}**(**μ_{MMU}**,** Δt**,** $\overline{kt}_{min}$**, retry)**{ {There are one chosen value $\overline{kt}_{min}$ and two initially set values$P_{GCMI} = \overline{kt}_{max} = \nu_{GC}$}
2: **while** (retry) **do**
3: **if** $(P_{GCMI} - \frac{\mu_{MMU} \cdot \Delta t + T_a^{max}}{\lfloor \frac{\Delta t}{P_{GCMI}} \rfloor}) > \overline{kt}_{min}$ **then**
4: set $\overline{kt}_{max}$ to $P_{GCMI} - (\mu_{MMU} * \Delta t + T_a^{max} \lfloor \frac{\Delta t}{P_{GCMI}} \rfloor)$;
5: return P_{GCMI};
6: **else**
7: $P_{GCMI} += \nu_{GC}$;
8: **end if**
9: decrement retry quanta by 1;
10: **end while**
11: return 0; {Failed to find P_{GCMI}}
12: }

Algorithm 4 ξ_W GC: Determine GC scheduling parameters together with heapsize requirement H with MMU requirements

Require: *This function is called upon a task set modeled in* Γ_{N+1} *and returns the heapsize requirement* H

1: **long Sched_Analysis_MMU(**μ_{MMU}**,** Δt**)**{ {There are two preset values: initial $P_{GCMI} = \overline{kt}_{max} = \nu_{GC}$}
2: $P_{GCMI} =$ Min_P_{GCMI}(μ_{MMU},Δt,retry)
3: **if** ($P_{GCMI} == 0$) **then**
4: increase retry quanta {Algorithm 3 failed to calculate the valid P_{GCMI} within the preset retry quanta.}
5: **else**
6: Calculate the heapsize requirement H using Equation (5.10)
7: **end if**
8: return 0;
9: }

- Choose a valid free memory threshold F, given in Equation (5.4);
- Derive a minimum heap requirement H, given in Equation (5.8).

Algorithm 3 and Algorithm 4 calculate the k_i, F and H. These derived scheduling parameters guarantee that a task set Γ_{N+1} is schedulable to meet the specified μ_{MMU} over Δt.

5.4 Time-based RTGC configuration schedulability analysis using the exact *C-space* feasibility test for fixed-priority systems

Fixed priority scheduling is widely used in modern real-time systems and many schedulability analysis have been extensively studied in real-time literature. As proven by Liu and Layland [68], the worst-case scenario for a periodic task set scheduled by RM occurs when all the tasks are simultaneously activated. Further, under arbitrary fixed priorities (non-RM) this scenario of simultaneous activation is still the worst-case even with the presence of shared resources, [86]. Without loss of generality, in the task model Γ_{n+1} given in section 3.3.1, assume $\Phi_i = 0$ for all the tasks including the GC, where Φ_i is an initial activation time for real-time task τ_i.

In this section, a real-time task set Γ_{n+1} is scheduled with fixed priorities (FP) and an exact schedulability analysis proposed in [15] is used. It is called the *C-space* feasibility test: a task set is viewed as a point in a space specified by the task set parameters, tasks periods T_i and deadlines D_i. The feasibility test verifies whether this point (representing the task set) belongs to a schedulable region in *C-space*. A space where every coordinate is represented by a task execution time C_i is called a *C-space*. In other words, the *C-space* feasibility test considers periods T_i and deadlines D_i as parameters, whereas worst-case execution times C_i are the "free variables", [15]. A constraint on the C_i variables is a function of all the T_i and D_i, which identifies a schedulable region in the *C-space*. The feasibility test checks to see whether the task set falls in the schedulable region or

not. In particular, a schedulable region is formulated as follows in [15]

$$M_n(T_1, \cdots, T_n, D_1, \cdots, D_n) = \{(C_1, \cdots, C_n \in \Re_+^n : \Gamma_n \text{ is schedulable by FP}\} \tag{5.11}$$

The *C-space* feasibility test is an exact schedulability analysis for a task set Γ_n with an arbitrary fixed priority (FP) assignment. The formulation of the schedulability analysis problem precisely specifies the feasibility(schedulability) region in the space of task execution time (the *C-space*).

5.4.1 GC-included C-space Formulation and the schedulable region

The original region formulation, [15], is extended to include the GC task, denoted as M_{N+1} as follows:

$$M_{N+1}((T_1, D_1), \cdots, (T_N, D_N), (T_{GC}, D_{GC})) \tag{5.12}$$

$$= \{(C_1, \cdots, C_n, C_{GC} \in \Re_+^{N+1} : \Gamma_{N+1} \text{ is schedulable by FP}\}$$

Similarly, in the formulation of M_{N+1}, periods T_i, T_{GC} and deadlines D_i, D_{GC} are considered as parameters, whereas C_i, C_{GC} are the free variables. A constraint on the C_i, C_{GC} variables, which is a function of all the T_i, T_{GC} and the D_i, D_{GC}, gives the schedulable region in the *C-Space* for the task set Γ_{N+1}. The schedulability check is simply to make sure that the C_i, C_{GC} given in Γ_{N+1} fall in this schedulable region.

The C-space feasibility analysis theorem in [15] is applied in this section. It is restated using the extended region formulation of Equation (5.12) below:

Theorem. *Given a periodic task set with GC,* $\Gamma_{N+1} = \{\tau_1, \cdots, \tau_i, \tau_{GC}, \tau_{i+1}, \cdots, \tau_N\}$*, where tasks priorities are in decreasing order and the schedulable region of the task set* M_{N+1} *(defined in Equation (5.12)), is given by*

$$M_{N+1}(T_1, \cdots, T_{GC}, \cdots, T_N, D_1, \cdots, D_{GC}, \cdots, D_N) \tag{5.13}$$

$$= \{\forall i = 1, \cdots, N, GC, C_i \cdot \sigma_i + B_i + W(D_i) \leq D_i\}$$

Where $W(D_i)$ and $P_{i-1}(D_i)$ are further defined as:

$$W(D_i) = \tag{5.14}$$

$$\begin{cases} \min_{t \in P_{i-1}(D_i)} \sum_{j=1}^{i-1} \lfloor \frac{t}{T_j} \rfloor \cdot C_j + (D_i - t) & (i > 0) \\ 0 & (i = 0) \end{cases}$$

and

$$P_i(t) = \begin{cases} P_{i-1}(\lfloor \frac{t}{T_i} \rfloor \cdot T_i) \bigcup P_{i-1}(t) & (i > 0) \\ t & (i = 0) \end{cases}$$

Our schedulability analysis algorithm for a given GC-included task set Γ_{N+1} is based on the above Theorem to check whether worst-case task execution time C_i and worst-case GC cycle execution time C_{GC} fall in the schedulable region in the *C-space*.

5.4.2 RTGC configuration ξ_T scheduling parameters constraints

Constraints are derived when GC is scheduled using the RTGC configuration instance ξ_t in this section. First, for the GC task, we have $D_{GC} \leq T_{GC}$, D_{GC} is the relative GC task deadline, thus:

$$D_{GC} = \lfloor \frac{C_{GC}}{P_{GCI}} \rfloor \cdot P_{MI} + C_{GC} \leq T_{GC} \tag{5.15}$$

where P_{GCI} is the GC execution time quantum and P_{MI} is the mutator execution time quanta assigned when GC increments and mutator are interleaved when GC is triggered at intervals of T_{GC}.

Lemma 1 tells a heap requirement as a function of the maximum live memory and maximum mutator allocation. Here, we have $A_{max} = \sum_{i=1}^{n} \frac{T_{GC}}{T_i} \cdot A_i$, therefore, the heap requirement using

the RTGC configuration ξ_T is calculated as:

$$H = L + 2 \cdot \sum_{i=1}^{n} \frac{T_{GC}}{T_i} \cdot A_i \tag{5.16}$$

Recall that Equation (5.9) calculates C_{GC} as a function of heapsize H, where $C_{GC} = \Upsilon + \delta_{S_r} \cdot H$. Using Equation (5.15) and (5.16), the constraint on GC period T_{GC} is expressed as:

$$T_{GC} \geq \frac{\Upsilon + \delta_{S_r} \cdot L + \dfrac{P_{MI} \cdot P_{GCI}}{P_{MI} + P_{GCI}}}{\dfrac{1}{1 + \dfrac{P_{MI}}{P_{GCI}}} - 2 \cdot \delta_{S_r} \cdot \sum_{i=1}^{N} \dfrac{A_i}{T_i}} \tag{5.17}$$

All these constraints of ξ_T scheduling parameters need scheduling and schedulability analysis using the *C-space* technique.

5.4.3 Algorithms of ξ_T scheduling and schedulability analysis using C-space

Algorithm 5 takes a GC-included task set Γ_{N+1} and tells whether this task set is schedulable or not according to the exact *C-space* schedulability analysis given in the Theorem on page 121. The function, *calcWorkload*, recursively computes $W(D_i)$ defined in Equation (5.14) in the Theorem. Figure C.1 in the Appendix C lists the algorithm implementation in C#.

Algorithm 5 ξ_T GC: Fixed-Priority schedulability test with *C-space*

Require: *This function is called upon the task set modeled as Γ_{N+1} and returns whether the task set is schedulable or not according to the exact C-space schedulability analysis*

```
bool FPTest(Γ_{N+1}){
for (i = 1; i <= N + 1; i++) do
    if (τ_i.C_i + calcWorkload(i-1,τ_i.D_i) > τ_i.D_i) then
        return FALSE;
    end if
end for
return TRUE;
}
```

Using Algorithm 5 to check the schedulability of a given task set Γ_{N+1}, for time-based GC scheduling ξ_T, given the chosen mutator and GC increment sizes, P_{MI} and P_{GCI} satisfying constraint in Expression (5.15), the algorithm (6) computes the GC period T_{GC} and the heap size requirement H iteratively until the given Γ_{N+1} (with GC included) succeeds in the exact schedulability analysis. In the Algorithm 6, RMS is used to assign the fixed-priority to the tasks in Γ_{N+1}. Other priority assignment policies can be programmed in Algorithm 6. This algorithm takes care of the GC period constraint T_{GC} by using Expression (5.17) to calculate the initial value of T_{GC}.

Algorithm 6 ξ_T GC: Finding T_{GC}, H with RMS

Require: *This function computes the GC scheduling parameter T_{GC} for the time-based GC scheduling ξ_T*

```
Finding_GCSched(Γ_{N+1}){
Compute T_GC = T_GC^init with Equation (5.17)
while (T_GC > C_GC) do
    Recompute heapsize H using Equation (5.16) for the current T_GC
    Recompute C_GC using Equation (5.9) with the new H
    Reorder the task set Γ_{N+1} with the new calculated T_GC according to RMS policy
    if FPTest(Γ_{N+1}) == TRUE then
        return the current T_GC and H that succeed schedulability test
    else
        T_GC+ = T_GC^INC {T_GC^INC is a preset constant and with the new T_GC, the next iteration of
        the computation starts.}
    end if
end while
}
```

To summarize, the scheduling of RTGC configuration ξ_T with a task set Γ_{n+1} using fixed-priority assignment in the formulation of M_{n+1} consists of three steps:

- Determine the GC incrementality parameters P_{GCI} and P_{MI} constraint, given in Expression (5.15);
- Derive the minimum heap requirement H, given in Equation (5.16).

- Perform schedulability analysis by checking whether the C_i and C_{GC} fall in the schedulable *C-space* given in Theorem 5.4.1.

With the P_{GCI}, P_{MI}, T_{GC} and H chosen with the above steps, the task set Γ_{N+1} is schedulable with fixed-priority in the descending order as in Γ_{N+1} verified by C-space schedulability analysis.

5.5 Summary

In this chapter, we analyzed two examples of RTGC configuration plans: work-based ξ_W and time-based ξ_T. These RTGC configurations are scheduled with two types of real-time systems that have: (1) a guaranteed minimum mutator utilization, μ_{MMU} over time duration Δt; or (2) specific deadline requirements and fixed-priority assignments. The work-based ξ_W is applied to the *MMU*-based real-time system. The time-based ξ_T is utilized in the fixed-priority real-time system with either RM (rate monotonic) or non-RM priority assignments. Schedulability analysis are performed, where the *C-space* feasibility analysis is used for time-based RTGC configuration ξ_T.

Constraints on both RTGC configuration plans' scheduling parameters are derived. Heapsize requirement H is calculated as a function of GC cost estimate C_{GC} from `GCCM` component and the RTGC configurations scheduling parameters: for instance, P_{GCMI}, $\overline{kt}_{min}$ in work-based RTGC configuration, ξ_W and P_{GCI}, P_{MI} in time-based RTGC configuration, ξ_T. Algorithms of schedulability analysis compute and check the GC scheduling parameters to make sure that the real-time task set Γ_{N+1} is schedulable if the feasible scheduling parameters are found.

CHAPTER 6

EXPERIMENTAL DEMONSTRATION

This chapter demonstrates our systematic RTGC approach using the system model for GC-enabled real-time systems:

- Using the task model Γ_{N+1} to express a real-time application with workload. The metrics that should be known *a priori* for real-time tasks are estimated;
- Using the GC cost model `GCCM` in chapter 4 to calculate GC-related cost;
- Using the two RTGC-configuration based GC scheduling modules (the scheduling instances in the `SCHED`) in chapter 5 to determine the GC scheduling parameters.

As a test platform, a real-time application – *status router* was implemented to illustrate the use of our RTGC approach using the system model. The status router application (SR) is a multi-casting packet forwarder with real-time processing requirements, [50]. It needs be scheduled during operation as the offered load pattern changes.

This chapter is organized as follows: section 6.1 introduces the execution context of the status router and presents its implementation. Section 6.2 models the status router with its publication-subscription workloads and the GC work using Γ_{N+1}. The worst-case real-time task execution time C_i, task period T_i and maximum task allocation A_i are calculated as functions of the status router's attributes. Section 6.3 estimates the GC-related cost using the GC cost model. The status router application-dependent parameters are measured and calculated. For instance, the maximum

memory allocation, A_{max}, and maximum memory deallocation(garbage), G_{max}, are calculated as functions of the status router's attributes as well. Section 6.4 runs the GC scheduling modules implemented according to the algorithms given in chapter 5. GC work is scheduled using the calculated GC scheduling parameters. Different publication and subscription workloads are used. The experiments show that the feasible GC scheduling parameters can be used to schedule GC work in executing the status router, which demonstrates the utility of our systematic RTGC approach using the system model. Furthermore, heap memory consumption is visualized and the status router packet forwarding latency is examined to evaluate the performance. Section 6.5 draws a conclusion.

6.1 An overview of the status router

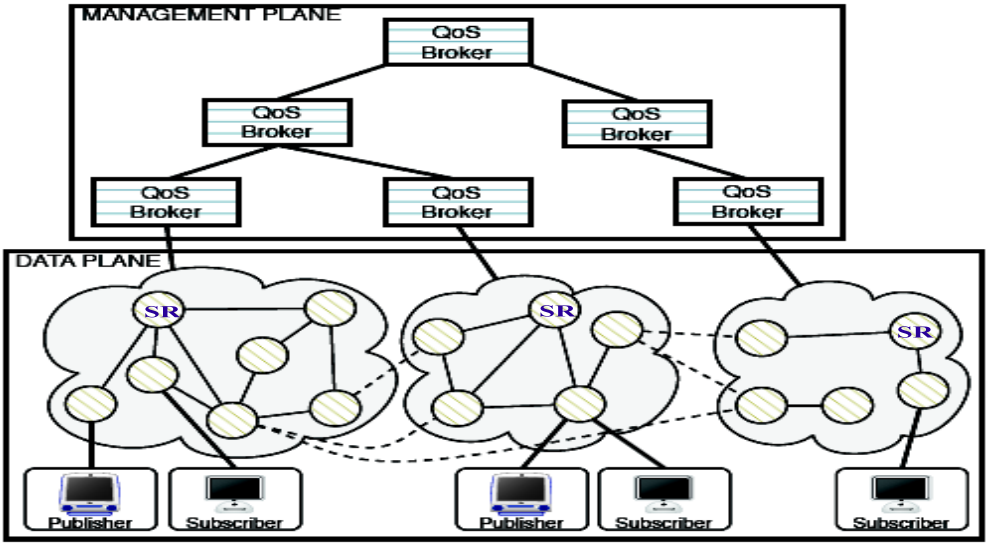
The GridStat status router (SR) is a multi-casting packet forwarder with real-time processing requirements, [50]. The original prototype status router was built in Java, ignoring the real-time requirement with the result that GC-induced delays occasionally disrupt its operation. In this section the status router's execution context and implementation are presented to provide necessary knowledge before expressing the application using the task model Γ_{N+1} in the next section. Our discussion starts with the background of GridStat architecture, where the status router is deployed.

6.1.1 The execution context of status router

GridStat was developed with the needs of electric power grid monitor and control in mind. It targets an application domain where the vast majority of data is published periodically, while allowing only a small amount of aperiodic data to be published. "The major communication requirements were to provide end-to-end reliability and timeliness for event streams scalable into the wide-area setting," [50]. GridStat implements the publisher-subscriber paradigm and hence, its architecture is appropriately composed of the key entities: publisher(**P**), subscriber(**S**), status router(**SR**) and QoS broker(**B**), which are divided into two planes: the *management plane* and *data plane* according

to the functionality. The role of the management plane is to control the resources in the data plane in order to provide QoS guarantees and scale to the wide-area setting. The role of the data plane is to forward events from the publishers to the subscribers, as efficiently as possible, so as to provide end-to-end timeliness and redundancy guarantees. Figure 6.1 shows the GridStat architecture which is included in [50].

Figure 6.1: GridStat Architecture



The QoS brokers preallocate paths between publishers and subscribers in order to guarantee delivery of the requested information. Dynamic subscription requests can also be handled by the brokers when there are available resources. The status routers forward events from publishers to subscribers at the specified intervals without dropping them.

6.1.2 Status router implementation

The status router for our experiments consists of a simplified re-implementation of publisher, subscriber, event channel and status router. In the GridStat framework, *status variables* represent some device which has a state that changes over time. *Status events* are an updated value for a status variable at a given time and are encapsulated into messages that flow from publishers to subscribers.

The re-implemented entities are:

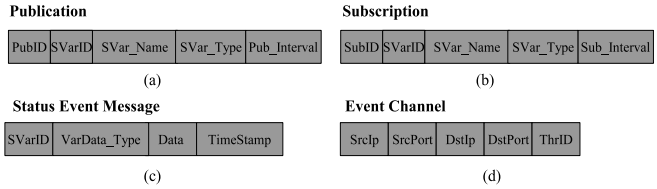
- *Publisher (push mode):* provides the abstraction of publishing streams of status events. Each stream published by a publisher is called a *publication*. One publisher can publish multiple publications. The data structures of the publication and the published message are in figure 6.2 (a). Each publication specifies that a status variable is published at certain time interval. The publishers are implemented as the active entities that push events at the specified interval.

- *Subscriber (push mode):* provides the abstraction of subscribing to a status event stream. One subscriber can make multiple subscriptions. The data structure representing a subscription is in figure 6.2 (b). Each subscription specifies that a status variable is subscribed at certain time interval. The subscribers are implemented as entities that are signaled when status events arrive.

- *Event Channel:* provides the abstraction of the communication links that send and receive messages. The status event message format and event channel data structure are in figure 6.2 (c),(d). The current implementation uses the UDP protocol as the transport layer. The field *ThrID* is the ID of the thread generated in the status router that handles the connection (event channel) between status router and publisher or subscriber.

- *Status Router:* provides the part of the framework that routes status event streams from the publishers to the subscribers. It can be viewed as an extended router that forwards events streams by having mechanisms to take advantage of the semantics of the status variables. The multi-cast rate-filtering routing mechanism is implemented, which is given in Algorithm 7. Step (1) in Algorithm 7 performs packet *filtering* and step (2) performs multi-cast routing.

Figure 6.2: Status Router Data Structures



Algorithm 7 Sparse Multi-cast rate-filtering routing Algorithm

Require: *This function is called upon message arrival in status router, the sparse routing algorithm performs **filtering** first and the **routing** afterwords.*

1: **if** (StatusVar.TimeStamp $+\lceil pubInterval/2 \rceil$ % subInterval[0..i] $<$ pubInterval) **then**
2: Obtain the list of event channel indexes, ECIndex[], which the message surviving the filtering needs be enqueued to $\{i$ (in the if statement) is the number of subscriptions with unique subInterval for the status variable SVar$\}$
3: **for** $j = 0$ to ECIndex.Length **do**
4: Enqueue the message to the outgoing queue $ECOut[ECIndex[j]]$ that subscribe to the status variable at the matching intervals
5: **end for**
6: **end if**

In this demonstration the management plane of the GridStat framework is simulated by a set of configuration files to preallocate resources for the publishers and subscribers, and represent publications, subscriptions and event channels. Figure 6.3 summarizes the layout of the status router demonstration, where configuration files simulating the management plane are shared by the publishers, subscribers, event channels and status router.

Figure 6.4 shows the components inside the status router.

6.2 Status router task model

The status router with publication-subscription workloads is modeled using the task model Γ_{N+1}, which is the input of both the *GCCM* and *SCHED* in order to determine GC scheduling parameters for a specific workload. There are three types of tasks τ_p, τ_c and τ_{GC} in the status router (SR):

- *Producer tasks*: serve the incoming connection between SR and publishers. Since they read

Figure 6.3: The Layout of the status router demonstration

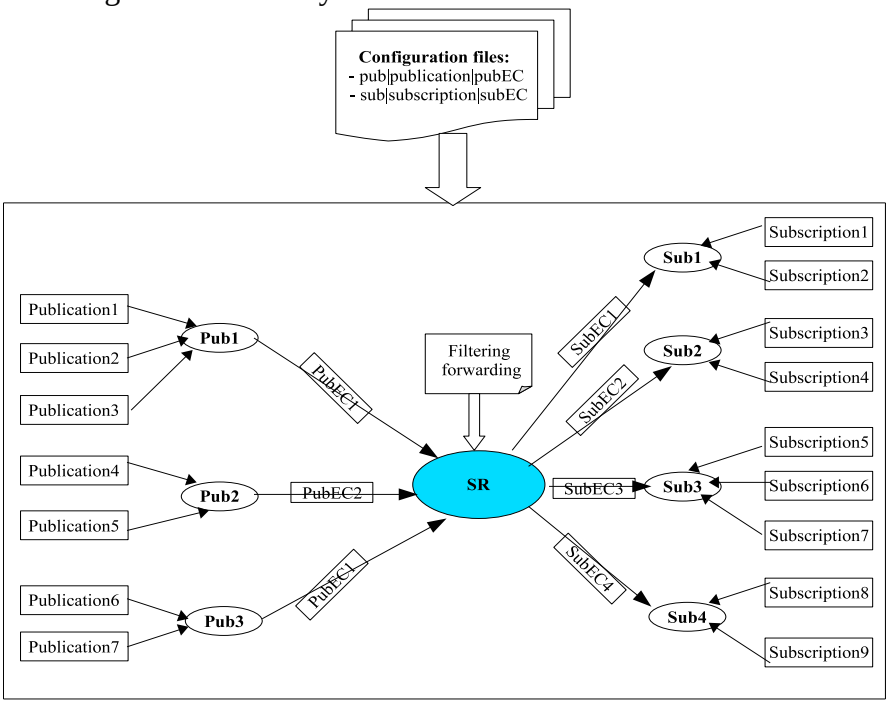
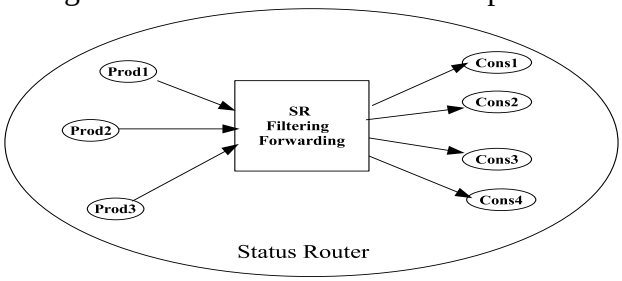


Figure 6.4: The Status Router Components



in status variable messages from sockets and produce data in status router, they are called producer tasks, τ_{p_m}, where $1 \leq m \leq P$ and P is the total number of incoming connections served in the SR. Each producer task is served by one thread and owns one incoming event channel in the SR.

- *Consumer tasks*: serve the outgoing connection between SR and subscribers. Since they write out status variable messages to sockets and consume data in status router, they are called consumer tasks, τ_{c_l}, where $1 \leq l \leq C$ and C is the total number of outgoing connections served in SR. Each consumer task is served by one thread and owns one outgoing queue and one outgoing event channel in the SR.

- *RTGC task*: the real-time garbage collection task, τ_{GC}, which will be scheduled using the work-based ξ_W or time-based ξ_T RTGC configurations.

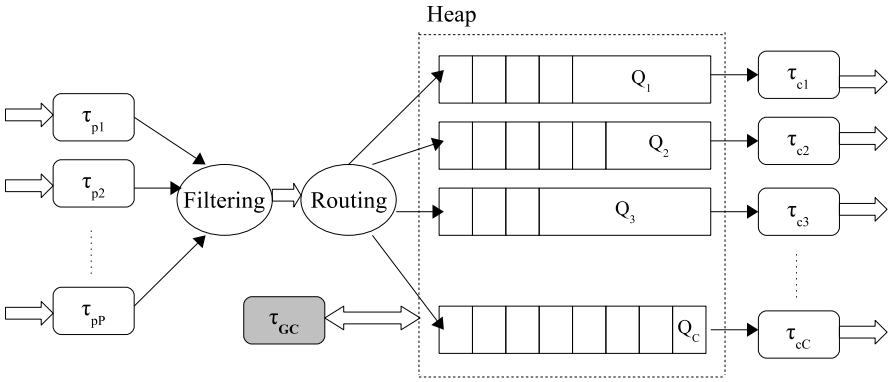
Thus, the task model for the status router is

$$\Gamma_{N+1}^{SR} = \{\tau_{p_1}, \cdots, \tau_{p_P}, \tau_{c_1}, \cdots, \tau_{c_C}, \tau_{GC}\},$$
$$\text{where} P + C = N$$

The interaction among tasks in Γ_{N+1}^{SR} is depicted in figure 6.5. Each publisher or subscriber is served by one thread in the status router.

The producer task τ_{p_m} publishes (generates) publications $P_m^1 \cdots P_m^M$ at publishing intervals of $pubInt_m^1 \cdots pubInt_m^M$ and the consumer task τ_{c_l} pushes subscriptions $S_l^1 \cdots S_l^L$ upon message arrivals at subscribed intervals of $subInt_l^1 \cdots subInt_l^L$. Their periods are calculated below, where

Figure 6.5: The Status Router task model



LCM means the *Least Common Multiple*:

$$T_{p_m} = LCM(pubInt_m^1 \cdots pubInt_m^M) \tag{6.1}$$

$$T_{c_l} = LCM(subInt_l^1 \cdots subInt_l^L). \tag{6.2}$$

For heap memory usage, the producer task τ_{p_m} allocates memory while the consumer tasks τ_{c_l} deallocates (garbage) memory for GC to reclaim. We define the following attributes that are unique to the status router application before calculating real-time tasks' parameters in Γ_{N+1} known *a priori*:

- S is the message size consumed in heap (Bytes);
- S_P is the memory consumption needed to process one message upon its arrival (Bytes);
- t_I is the time to read in one message by the producer task from incoming sockets (ms);
- t_F is the time to perform filtering check on one message by the producer task (ms);
- t_R is the time to route one message and enqueue it to the right outgoing link after the message passes the filtering test by the producer task (ms) during multi-casting;

- t_O is the time to write one message to the outgoing sockets by the consumer task (ms).

- $c_{sub_m^i}$ is the total number of subscriptions that subscribe to the publication P_m^i handled by the publisher task τ_{p_m}.

The worst-case execution time C_i for each producer task and consumer task are calculated:

$$C_{p_m} = \sum_{i=1}^{M} \frac{T_{p_m}}{pubInt_m^i} \cdot (t_I + t_F) + \sum_{i=1}^{M} \frac{T_{p_m}}{pubInt_m^i} \cdot \sum_{k}^{c_{sub_m^i}} \frac{pubInt_m^i}{subInt_m^i} \cdot t_R \tag{6.3}$$

$$C_{c_l} = \sum_{j=1}^{L} \frac{T_{c_l}}{subInt_l^j} \cdot t_O$$

Due to the periodic behavior of the real-time tasks (producer and consumer tasks), the status router execution has a periodic pattern with the period calculated as:

$$T_{\Gamma_{N+1}^{sr}} = LCM(T_{p_1}, \cdots, T_{p_M}, T_{c_1}, \cdots, T_{c_L}) \tag{6.4}$$

Based on the implementation characteristics of the status router, the real-time tasks' memory usage in SR – maximum memory allocation A_i and maximum memory deallocation G_i [1] are calculated:

$$A_{p_m} = \sum_{i=1}^{M} \frac{T_{p_m}}{pubInt_m^i} \cdot \sum_{k}^{c_{sub_m^i}} \frac{pubInt_m^i}{subInt_m^i} \cdot S + \sum_{i=1}^{M} \frac{T_{p_m}}{pubInt_m^i} \cdot S_P \tag{6.5}$$

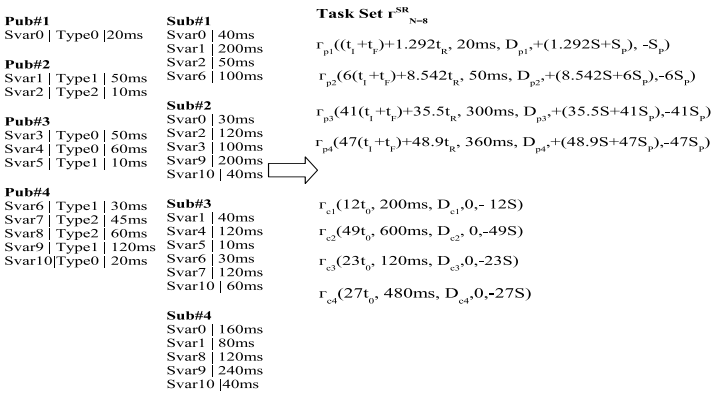
[1] Introducing the maximum memory deallocation G_i here is to support calculating the maximum garbage need be reclaimed in the SR.

$$G_{p_m} = \sum_{i=1}^{M} \frac{T_{p_m}}{pubInt_m^i} \cdot S_P$$

$$G_{c_l} = \sum_{j=1}^{L} \frac{T_{c_l}}{subInt_l^j} \cdot S$$

A sample workload configuration of the publishers with publications and subscribers with subscriptions in the SR and the modeled real-time task set Γ_N is shown in figure 6.6, where the memory usage is calculated using Equation (6.5) and the worst-case execution time is calculated using Equation (6.3).

Figure 6.6: A sample Publisher and Subscriber workload configuration and its task model Γ_N



Since the status router's execution is repeatable with period $T_{\Gamma^{sr}_{N+1}}$ calculated in Equation (6.4), GC scheduling parameters are calculated for a complete periodic status router's execution. In order to calculate the worst-case GC cost, the maximum memory allocation A_{max} and maximum garbage G_{max} are calculated for the period $T_{\Gamma^{sr}_{N+1}}$ using the real-time tasks' memory usage parameters: A_{p_i}, G_{p_i} and G_{c_j}:

$$A_{max} = A_{init} + \sum_{i=1}^{M} \frac{T_{\Gamma_{sr}}}{T_{p_i}} \cdot A_{p_i} \tag{6.6}$$

$$G_{max} = \sum_{i=1}^{M} \frac{T_{\Gamma_{sr}}}{T_{p_i}} \cdot G_{c_i} + \sum_{j=1}^{L} \frac{T_{\Gamma_{sr}}}{T_{c_j}} \cdot G_{c_j}$$

Table 6.1: Status Router Publisher-Subscriber Workload specifications (byte, μs)

Task set	task	C_i	$T_i(D_i \leq T_i)$	A_i	G_i
TS_1	τ_{p_1}	$(t_I + t_F) + 1.292t_R$	20	$1.292S + S_p$	S_p
	τ_{p_2}	$6(t_I + t_F) + 8.542t_R$	50	$8.542S + 6S_p$	$6S_p$
	τ_{p_3}	$41(t_I + t_F) + 35.5t_R$	300	$35.3S + 41S_p$	$41S_p$
	τ_{p_4}	$47(t_I + t_F) + 48.9t_R$	360	$48.9S + 47S_p$	$47S_p$
	τ_{c_1}	$12t_O$	200	0	$12S$
	τ_{c_2}	$49t_O$	600	0	$49S$
	τ_{c_3}	$23t_O$	120	0	$23S$
	τ_{c_4}	$27t_O$	480	0	$27S$
TS_2	τ_{p_1}	$205(t_I + t_F) + 177.889t_R$	40	$177.889S + 205S_p$	$205S_p$
	τ_{p_1}	$21(t_I + t_F) + 12.167t_R$	5	$12.167S + 21S_p$	$21S_p$
	τ_{p_1}	$83(t_I + t_F) + 41.917t_R$	30	$41.717S + 83S_p$	$83S_p$
	τ_{c_1}	$1561t_O$	360	0	$1561S$
	τ_{c_2}	$178t_O$	60	0	$178S$
	τ_{c_3}	$31t_O$	40	0	$31S$

6.3 Calculating status router GC cost using GCCM

Given a publisher-subscriber workload, a task set is modeled as Γ_N^{SR}, which is one of the inputs for the GC cost model. Other inputs include the status router application-dependent parameters defined in chapter 4, the status router attributes defined in the previous section 6.2, and the set of GC-implementation dependent cost coefficients defined in chapter 4. Two publisher-subscriber workloads are studied in this chapter, and their specifications are shown in figure D.1 in the Appendix D. Workload TS_1 has lighter loads than TS_2 that TS_1's overall incoming packet rate is 437.22p(acket)/s(econd) and outgoing packet rate is 389.59p/s while TS_2's overall incoming packet rate is 12091.67p/s and outgoing packet rate is 4175.00p/s. With the workloads specifications, their attributes defined in Γ_{N+1} are calculated using Equation (6.3) and (6.5) in table 6.1.

As in chapter 4, the experiments were conducted on a PC (IBM Thinkpad T60) with a Genuine Intel(R) CPU T2400 at 1.83GHz and 1.5GBytes physical memory running Linux 2.6.17-gentoo-r4 with real-time kernel patch-2.6.17-rt8. The real-time kernel provides a high resolution timer. SMP is disabled for better real-time kernel stability. Physical heap memory is locked using *mlock()* to prevent paging. For each experiment, we lock a specific amount of memory during Mono runtime start-up by setting the *GC_INITIAL_HEAP_SIZE* environment variable. Runlevel 1 (single user mode) was used to minimize kernel activity influences.

Table 6.2: Status Router Attributes (byte, μs)

	SR Application-specific Parameters (bytes, μs)						
	S	S_p	t_I	t_F	t_R	t_O	A_{init}
Status Router	512	32608	62	3	55	16	497792

Table 6.3: Status Router Application-dependent parameters in `GCCM`

Status Router	Application-dependent Parameters (number of references/byte, bytes, μs)						
	θ_t	θ_h	θ_{h_v}	θ_{dl}	L	G_l	G_s
TS_1(Workload 1)	0.011447	0.177405	0.174694	0.000443	91426	75667332	199408
TS_2(Workload 2)	0.032072	0.065848	0.041481	0.000374	80923	90210401	282672

Using the above system configuration, the GC cost model's inputs are obtained so that GC-related costs can be calculated for the status router with its workloads.

6.3.1 Obtaining GC cost model inputs

The inputs of (1) status router related parameters [2]; (2) status router application-dependent parameters defined in the GC cost model; and (3) the set of GC cost coefficients and the atomic GC phase costs are discussed in turn:

Status router attributes

The status router related parameters defined in section 6.2 are listed in table 6.2:

Status router application-dependent parameters in `GCCM`

The status router application-dependent parameters defined in the GC cost model are determined by both the status router implementation and its current workload. These parameters are reported in table 6.3 and 6.4 for both workloads.

Due to the implementation characteristics of the status router, the SR's A_{max} and $G_{max} = G_l + G_s$ can be calculated using Equations (6.6). The calculation has pessimism of approximately 1.35 for TS_1 and 1.59 for TS_2. Equation (6.6) is used to calculate the SR's memory usage on the basis of its period instead of measuring them.

[2]These parameters are not defined in the `GCCM`, since they are unique to the status router application.

Table 6.4: Status Router Application-dependent Parameters in GCCM (Cont.) (bytes)

Status Router	The measured and calculated A_{max} and G_{max} for SR (bytes)					
	$\widehat{A}_{max}$	A_{max}	ρ^A	$\widehat{G}_{max}$	G_{max}	ρ^G
TS_1(Workload 1)	76482968	103341140	**1.3511**	76866740	102473888	**1.334**
TS_2(Workload 2)	91133764	143444544	**1.5741**	91043741	143431520	**1.576**

GC cost coefficients and atomic GC phase costs in GCCM

Recall that in section 4.3 in chapter 4, the GC cost model calculates GC-related costs for real-time task sets Γ_{N+1} including independent tasks or dependent tasks. The GC cost coefficients measurements for the status router are reported in Tables 6.5, 6.6 and 6.7. There are the worst-case measurements wc, the adjusted worst-case awc_1 using 99.9% percentile and awc_1 using 99.8% percentile and the average measurements avg. We used the same measuring procedure as chapter 4.

Table 6.5: Status Router worst-case GC cost coefficients – processing rate

Application	S_4		S_5				S_7,S_8	
	δ_{R_r}		$\delta_{M_{sr}}$		$\delta_{M_{hr}}$		δ_{S_r}	
	CPU cycles/B	μs/B	CPU cycles/B	μs/B	CPU cycles/B	μs/B	CPU cycles/B	μs/B
Status Router	29.91	0.01636	19.38	0.0106	35.10	0.0192	0.0664	0.000036345

Table 6.6: Status Router worst-case and adjusted worst-case GC cost coefficients (a)

Application		S_1		S_2				S_4					
		δ_C		δ_{U_e}		δ_{U_p}		δ_{R_e}		$\delta_{R_{mp}}$		$\delta_{R_{bl}}$	
		CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs
Status Router	wc	33273	18.201	29513	16.137	6894	3.771	28996	15.855	26816	14.670	2432	1.330
	awc_1	990	0.541	693	0.379	5577	3.0450	990	0.541	885	0.484	2018	1.104
	awc_2	701	0.383	569	0.311	4516	2.470	790	0.432	671	0.367	1859	1.016
	avg	657	0.359	177	0.097	4346	2.377	779	0.426	148	0.081	510	0.279

The GC cost coefficient measurements again show that the GC cost model built in chapter 4 has a fine granularity that the part of the model pertaining to the GC implementation (the GC cost coefficients) is stable across applications and workloads.

Table 6.8 shows the measurements of the three atomic GC phases and two atomic finalization, sweep initialization time at the beginning of incremental GC phase P_6 and P_7 for the status router application.

Table 6.7: Status Router worst case and adjusted worst-case GC cost coefficients (b)

Application		S_5						S_6		S_7, S_8					
		δ_{M_e}		$\delta_{M_{mp}}$		$\delta_{M_{bl}}$		$\delta_{F_{dl}}$		δ_{S_l}		δ_{S_s}		$\delta_{S_{enq}}$	
		CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs
SR	wc	31732	17.359	30611	16.745	2271	1.242	1031	0.564	32338	17.690	33693	18.423	4716	2.580
	awc_1	990	0.541	495	0.271	2033	1.112	905	0.495	1826	0.998	9239	5.054	2310	1.263
	awc_2	516	0.282	320	0.175	1518	0.831	794	0.434	963	0.527	6078	3.325	1531	0.838
	avg	465	0.254	121	0.067	1328	0.726	593	0.324	846	0.463	1985	1.086	328	0.179

Table 6.8: GC atomic phase execution time for *SR* (byte, μs)

	Atomic GC phase execution time (μs)				
	T_{P_0}	T_{P_3}	T_{fin}	T_{sp}	T_{P_9}
Status Router	2.97	45.72	13.87	106.15	78.02

6.3.2 Calculating the worst-case GC execution time C_{GC} for status router

The worst-case GC execution time C_{GC} is calculated as a function of heapsize H using equation 5.9 in page (117) as:

$$C_{GC} = \sum_{i=0}^{i=9} T_{P_i} = \Upsilon + \delta_{S_r} \cdot H$$

Where $\delta_{S_r} = 3.6345\text{E}{-5}\mu s$/byte and Υ is calculated in table 6.9. Both the worst-case GC cost coefficients and adjusted GC cost coefficients (with 99.9% percentile) in table 6.6 and 6.7 are used, which are $\widehat{\Upsilon}_{wc}$, Υ_{wc}, $\widehat{\Upsilon}_{awc}$ and Υ_{awc}. The status router application-dependent parameters in table 6.3 and 6.4 are used as well.

6.3.3 Calculating GC write barrier cost for status router

In section 4.3.2, the write barrier cost coefficients are: $\delta_{WB_{C_1}} = 0.59\mu s$, $\delta_{WB_{C_2}} = 0.935\mu s$ and $\delta_{WB_P} = 1.311\mu s$. Complete write barrier calculations of the task set TS_1 and task set TS_2 are given in table D.1 and table D.2 in the Appendix D.

Table 6.9: Υ calculation for the Status Router (μs)

Status Router	Υ calculation where $C_{GC} = \Upsilon + \delta_{S_r} \cdot H$ (μs)					
	$\widehat{\Upsilon}_{wc}$	Υ_{wc}	$\rho^{\Upsilon_{wc}}$	$\widehat{\Upsilon}_{awc}$	Υ_{awc}	$\rho^{\Upsilon_{awc}}$
TS_1(Workload 1)	1926728.11	2266796.61	**1.1765**	58687.14	66132.37	**1.1270**
TS_2(Workload 2)	1690116.98	2354908.82	**1.3473**	52094.68	66878.94	**1.2481**

Table 6.10: Specifications of Status Router Workload 1 and 2 in Γ_{N+1} (ms, bytes)

	Tasks	C_i	σ_i	$T_i(D_i \leq T_i)$	$\widehat{A}_i$	A_i	$\widehat{G}_i$	G_i
TS_1	τ_{p_1}	0.136	2.553	20	22092	32758	21728	32096
	τ_{p_2}	0.860	1.729	50	157484	196950	146480	192576
	τ_{c_3}	0.368	1.099	120	0	0	9914	11776
	τ_{c_1}	0.192	1.151	200	0	0	5781	6144
	τ_{p_3}	4.618	1.796	300	936492	1334112	836792	1315936
	τ_{p_4}	5.745	1.875	360	936976	1533549	946728	1508512
	τ_{c_4}	0.432	1.100	480	0	0	11968	13824
	τ_{c_2}	0.784	1.088	600	0	0	23715	25088
TS_2	τ_{p_2}	2.034	1.069	5	516719	690999	578280	684768
	τ_{p_3}	7.701	1.060	30	2417215	2727926	2501329	2706464
	τ_{c_3}	0.496	1.131	40	0	0	12431	15872
	τ_{p_1}	23.109	1.081	40	5162221	6775720	5401208	6684640
	τ_{c_2}	2.848	1.138	60	0	0	78142	91136
	τ_{c_1}	24.98	1.125	360	0	0	698168	799232

With the status router specific parameters given in table 6.2, real-time task worst-case execution time C_i and memory consumption A_i, G_i calculated in table 6.1 using Equation (6.3) and (6.5), the complete status router workload specifications are shown in table 6.10.

The calculation of the real-time task maximum memory allocation A_i and deallocation G_i has pessimism ρ^{A_i} from 1.251 to 1.637 and pessimism ρ^{G_i} from 1.058 to 1.593 with the two workloads.

6.3.4 GC Scheduling Resolution ν_{GC} for status router

The scheduling and schedulability analysis computes the RTGC configuration parameters. The computation needs the knowledge of the GC scheduling resolution ν_{GC}, for instance, it is used in Algorithm 3 in page (119) in chapter 5. ν_{GC} is the maximum GC execution latency due to the GC incrementality that imposes on mutator execution (Refer to Equation 5.1 in page 111).

The GC scheduling resolution ν_{GC} is application-dependent, for instance, the atomic GC work a_5 in incremental heap mark phase is *Process half block of live heap in heap marking* P_5, the estimate of a_5's cost depends on the application (such as heap reference density θ_h *etc.*). In table 6.11, the GC scheduling resolution ν_{GC} is calculated for the status router with its workloads.

Table 6.11: Calculating the GC scheduling resolution ν_{GC} for status router (μs)

Status router	Atomic phase cost T_{P_j}	T_{P_0}	T_{P_3}	T_{P_9}	T_{sp}	T_{fin}			
		2.97	45.72	13.87	106.15	78.02			
SR Workload	Time cost t_i of atomic GC work a_i	t_1	t_2	t_4	t_5	t_6	t_7	t_8	ν_{GC}
TS_1		18.21	19.91	907	930	0.564	20.27	18.43	**930**μs
TS_2				1507	196				**1507**μs

6.4 Status router GC scheduling

In the previous section, the status router's GC costs are calculated based on its workloads. Both the GC costs and the specifications of the task set Γ_{N+1} are inputs of the `SCHED` component. Two GC scheduling modules are implemented according to the work-based ξ_W (with *MMU*) and time-based ξ_T (with fixed-priority system using *Cspace*) mechanisms presented in chapter 5. They calculate the feasible GC scheduling parameters that maintain the task set Γ_{N+1} schedulable.

The system setup is the same as previous experiments where the GC-related costs are estimated. The publishers and subscribers are running on a separate Desktop PC with a Genuine Intel(R) I686 Pentium(R) 4 CPU 2.53GHz and 512MBytes physical memory running Linux 2.6.19-ARCH #1 SMP PREEMPT. They are responsible for generating publication messages and receiving subscription messages.

6.4.1 ξ_W work-triggered GC scheduling parameters and performance

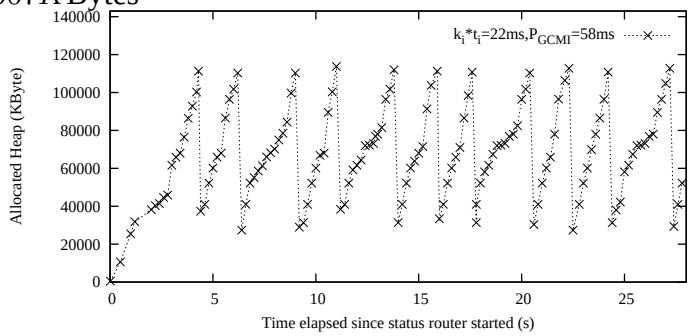
In table 6.12, the GC scheduling parameters, P_{GCMI}, R_{GC}, $\overline{kt}_{max}$ and $\overline{kt}_{min}$ are computed for timing requirements specified as a ($\mu_{MMU}, \delta t$) pair. The heapsize H is calculated for each set of the GC scheduling parameters. Using these GC scheduling constraints, we choose the GC execution characteristics – the size of $k_i \cdot t_i$ (for all $i = 1, 2, 4, 5, 6, 7, 8$) in our experiments as shown in table 6.12. The heapsize calculated using the adjusted worst-case GC cost coefficients are used when running the status router [3]. For instance, given a timing requirement of (0.6, 60)

[3]In table 6.12 and 6.13, *wc* shows the calculations using the worst-case GC cost estimations, and *awc* shows the calculations using the adjusted worst-case GC cost estimations (using the 99%-th pencentile). H_{used}^{max} is the observed maxmimum heap in use during execution. ρ_H is the pessimism of the calculated heapsize using the adjusted GC cost coefficients.

and $\overline{kt}_{min} = 9.3$ms, the choice of k_i needs to satisfy $9.3\text{ms} \leq k_i \cdot t_i \leq 22.34\text{ms}$ with a calculated heapsize of 144566KBytes as the second set of GC scheduling parameters in table 6.12[4]. The maximum heapsize in use, H_{used}^{max}, during execution is observed to be 113745KBytes.

The threshold of free memory F is chosen to be the the smallest amount of free memory during one periodic status router execution. This period is described and calculated in Equation 6.4 in section 6.2. The choice of F results in a heap consumption in figure 5.4 in page 113, where $M_e^A - M_e^G \leq 0$ so that threshold F's constraint expressed in Expression 5.4 is satisfied. The heap consumption with the calculated GC parameters in table 6.12 is visualized in figure 6.7, which shows a consistent heap consumption pattern as in figure 5.4.

Figure 6.7: TS_1 Memory consumption when GC is triggered at free memory threshold $F = 0.28H$, $H = 144567K$Bytes



In figure 6.7, GC work is scheduled using the parameters of $k_i \cdot t_i = 22$ms and $P_{GCMI} = 58$ms calculated in table 6.12, where H is set to $144567K$Bytes. F is chosen to be 28% of H, which is 40479KBytes. In other words, GC is triggered when the allocated heap reaches approximately 104088KBytes.

Let t_{arri}^{msg} be the arrival time of a message in the status router, t_{leav}^{msg} be the leaving time of the message (sent out by the status router), $R_{msg} = max(t_{leav}^{msg} - t_{arri}^{msg})$ is the maximum latency encountered by a message processed by the status router. R_{msg}^{STW} is the maximum latency using the

[4]Due to the large pessimism of the worst-case GC cost estimations, experiments are conducted using the adjusted worst-case GC cost estimation.

Table 6.12: ξ_W work-triggered *MMU* GC scheduling parameters for SR (ms, bytes)

	$(\mu_{MMU}, \delta t)$	P_{GCMI}	$\overline{kt}_{min}$	$\overline{kt}_{max}$		C_{GC}	R_{GC}	H	H_{used}^{max}	ρ^H	$k_i \cdot t_i$	P_{GCMI}-$k_i \cdot t_i$	R_{msg}^{STW}	R_{msg}^{INC}	A_{msg}^{STW}	A_{msg}^{INC}
TS_1	$(0.6, 60)$	29	9.3	11.97	*wc*	2274.18	7091.74	203181376			10	19				
					awc	805.48	2511.92	72347092	59790985	1.210			13.937	0.476	4.066	0.207
		58	9.3	22.34	*wc*	2281.59	14229.49	407088184			22	36				
					awc	808.11	5040.01	144566896	113744782	1.271			11.657	0.507	3.879	0.255
		58	18.6	22.28	*wc*	2274.23	7091.74	203181376			21	37				
					awc	805.50	2527.13	72781494	55516014	1.311			13.470	0.489	4.230	0.228
		60	20.00	23.27	*wc*	2273.83	6821.91	195473052			22	38				
					awc	805.38	2416.35	69616954	57725501	1.206			12.890	0.411	3.752	0.202
	$(0.4, 100)$	27	9.3	13.64	*wc*	2273.84	6601.05	193791488			12	15				
					awc	805.37	2338.51	67383706	54917446	1.227			10.723	0.329	3.816	0.196
		62	9.3	21.93	*wc*	2282.70	15217.66	435317603			21	41				
					awc	808.50	5390.15	154565205	109932578	1.406			12.329	0.421	4.117	0.237
		46	18.6	25.96	*wc*	2272.47	5620.74	161159034			20	26				
					awc	804.94	1990.92	57463600	49156202	1.169			11.813	0.397	3.562	0.209
		41	20.0	20.96	*wc*	2271.60	4657.09	133630016			20	21				
					awc	804.59	1649.60	47713360	41238859	1.157			13.672	0.402	4.251	0.248
TS_2	$(0.5, 50)$	47	15.07	21.06	*wc*	2377.86	7280.51	631367666			18	29				
					awc	67.49	207.04	28670884	23872510	1.201			9.892	0.326	2.870	0.231
		49	22.61	24.08	*wc*	2370.76	4603.60	232101764			21	28				
					awc	67.36	146.63	23122014	18968018	1.219			10.647	0.364	3.819	0.229
	$(0.75, 150)$	73	15.07	16.98	*wc*	2392.62	11636.90	1062893717			14	59				
					awc	68.00	330.81	40941786	31348994	1.306			12.391	0.395	4.121	0.238
		74	18.09	18.49	*wc*	2386.97	9866.47	882129872			16	58				
					awc	67.82	280.52	35801574	27796253	1.288			11.174	0.367	3.991	0.247

stop-the-world GC[5], and R_{msg}^{INC} is the maximum latency using the modified BDW real-time garbage collector. Each experiment run lasts for 15 minutes. Using the same heapsize, the observed R_{msg}^{INC} ranges from 0.326ms to 0.507ms which are less than R_{msg}^{STW} that ranges from 9.892ms to 13.937ms. The average message latency using the incremental GC, A_{msg}^{INC}, ranges from 0.196ms to 0.255ms are also less than A_{msg}^{STW} using the stop-the-world GC that ranges from 2.870ms to 4.251ms.

For task set TS_1, the observed mutator utilization ranges from 0.610 to 0.642 using the scheduling parameters derived with a targeted timing requirement of (0.6, 60); and ranges from 0.462 to 0.541 using the scheduling parameters derived with a targeted timing requirement of (0.4, 100). Also, for task set TS_2, the observed mutator utilization ranges from 0.561 to 0.617 using the scheduling parameters derived with a targeted timing requirement of (0.5, 50); and ranges from 0.772 to 0.791 using the scheduling parameters derived with a targeted timing requirement of (0.75, 150). The heapsize calculated using the adjust worst-case GC cost coefficients have a pessimism ranging from 1.157 to 1.406 compared to the observed maximum heapsize in use during execution.

6.4.2 ξ_T time-triggered GC scheduling parameters and performance

In the task model Γ_{N+1}, each producer or consumer task's deadline, D_i, is expressed as a function of the message latency, $R_{msg_k} = t_{leav}^{msg} - t_{arri}^{msg}$, encountered by a message processed by the status router.

$$D_i = C_i + max(\sum_{k}^{M_i} R_{msg_k}) = C_i + M_i \cdot R_{msg} \tag{6.7}$$

where M_i is the total number of messages incurred during one invocation of real-time task τ_i, R_{msg} is the maximum message latency, and R_{msg_k} is the message latency for the $k-$th message

[5]Here, the stop-the-world GC uses the modified BDW collector in a stop-the-world mode by disabling GC incrementality when GC cycle is triggered. The same heapsize H and free memory threhold F are set for both the incremental GC and the stop-the-world GC.

Table 6.13: ξ_T time-triggered *C-space* GC scheduling parameters for SR (ms, Bytes)

ξ_T scheduling parameters			P_{MI}	P_{GCI}	C_{GC}	$T_{GC}(D_{GC} = T_{GC})$	H	H_{used}^{max}	ρ^H	R_{msg}^{Target}	R_{msg}^{INC}	R_{msg}^{STM}
TS_1	τ_{GC}^a	*wc*	50	50	2272.49	5465.24	156718044			0.823		
		awc			805.50	2531.26	82901277	73902671	1.122	0.823	0.644	14.747
	τ_{GC}^b	*wc*	50	25	2274.86	7743.96	221815566			0.823		
		awc			806.34	3338.41	95959469	80099724	1.198	0.823	0.627	13.376
	τ_{GC}^c	*wc*	25	50	2270.06	3124.523	89849358			0.901		
		awc			804.60	1659.06	47984532	39953815	1.201	0.823	0.716	17.145
TS_2	τ_{GC}^f	*wc*	50	50	*Cannot find schedulable set*							
		awc			69.96	332.90	92701267	67566521	1.372	0.551	0.510	21.126
	τ_{GC}^e	*wc*	40	60	2357.96	5354.84	153553403			0.764		
		awc			68.81	216.77	62971011	46541767	1.353	0.622	0.542	19.311

among all these M_i messages, where $R_{msg_k} \leq R_{msg}$. Another constraint on the deadline D_i is that $D_i \leq T_i$, where T_i is the real-time task τ_i's period. Figure D.2 in the Appendix D lists the code to calculate a feasible GC period, T_{GC}, together with a targeted maximum message latency, R_{msg}, for a given workload in the status router.

The schedulability analysis is based on the *C-space* technique for fixed-priority systems. Rate-monotonic (RM) is used to assign real-time tasks' priorities. In the experiments, the initial target maximum message latency R_{msg} starts with $0.1ms$ and is incremented by $0.001ms$ until the task set Γ_{N+1} can pass the *C-space* schedulability analysis tests. This means the deadlines D_i calculated in Equation 6.7 can be met when GC is scheduled with the calculated heapsize, H and GC period, T_{GC}, using the chosen GC increment size, P_{GCI} and mutator increment size, P_{MI}. The calculated maximum message latency R_{msg} is the smallest latency that can be guaranteed by the schedulability analysis.

In table 6.13, the GC scheduling parameters, T_{GC} are computed for workloads and the chosen P_{GCI}(P_{MI}) so that the task set is schedulable. In other words, every task can meet its deadline so that a targeted maximum message latency can be achieved. The heapsize H is calculated for each set of the GC scheduling parameters. The heapsize calculated using the adjusted worst-case GC cost coefficients are used when running the status router.

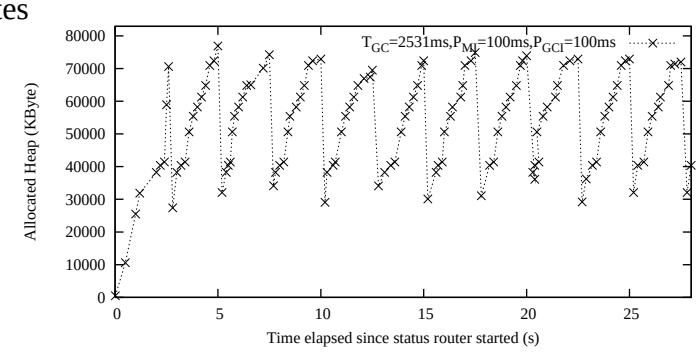
The calculated deadlines are reported in table D.3 in Appendix D with a targeted maximum

message latency, $R_{msg} = 0.823$ms for task set TS_1 and $R_{msg} = 0.551$ms for task set TS_2.

Each experiment run lasts for 15 minutes. Using the same heapsize, for task set TS_1, the observed R_{msg}^{INC} ranges from 0.627ms to 0.716ms for the GC scheduling parameters derived using a targeted $R_{msg} = 0.823$ms. These are less than R_{msg}^{STW} that ranges from 14.747ms to 17.145ms. For task set TS_2, the observed R_{msg}^{INC} is 0.542ms for the GC scheduling parameters derived using a targeted $R_{msg} = 0.612$ms, which is less than $R_{msg}^{STW} = 19.311$ms. Using the worst-case GC cost estimations with chosen $P_{GCI} = 100$ms and $P_{MI} = 100$ms, the schedulability analysis failed to find a schedulable GC scheduling parameters for task set TS_2 as a result of pessimism.

The heap consumption with the calculated GC parameters in table 6.13 is visualized in figure 6.8, which shows a GC invocation at the period of $T_{GC} = 2517$ms using GC task τ_{GC}^{b}'s parameters. The heapsize calculated using the adjust worst-case GC cost coefficients have pessimism ranging from 1.122 to 1.372 compared to the observed maximum heapsize in use during execution.

Figure 6.8: TS_1 Memory consumption when GC is triggered with period of $T_{GC} = 2531$ms, $H = 82901K$Bytes



6.5 Summary

As a test platform, a real-time application – *status router* was implemented to illustrate the use of our RTGC approach using the system model. The status router application (SR) is a multi-casting packet forwarder with real-time processing requirements, [50]. It needs be scheduled during operation as the offered load pattern changes.

The use of the systematic RTGC approach is demonstrated using the status router application with two publication-subscription workloads in this chapter. We presented the steps that need be taken to compute the GC scheduling parameters:

- The task model Γ_{N+1} expresses the status router implementation with its publication-subscription workloads. Calculations of the parameters that should be known *a priori* in the task model are performed.

- The GC cost model `GCCM` calculates GC-related cost for the status router with different workloads. The GC cost coefficients defined in the GC cost model in chapter 4 are measured using the status router application. It again shows that the part of the GC cost model pertaining to the GC implementation is stable across applications and workloads.

- The two RTGC configuration plans, work-based, ξ_W, and time-based, ξ_T, presented in chapter 5 are used to schedule GC in the status router application. The scheduling modules were implemented to calculate the feasible GC scheduling parameters.

The status router application runs using the feasible GC scheduling parameters derived by the systematic RTGC approach. For both workloads, there are no memory exhaustion during the status router execution while the real-time requirements are satisfied. For the work-based GC scheduling, ξ_W, the minimum mutator utilization over time duration pair, $(\mu_{MMU}, \Delta t)$, are observed to be met. For the time-based GC scheduling, ξ_T, the real-time tasks' deadlines are met. Since the deadlines in the status router are expressed as functions of the maximum message latency, R_{msg}. A maximum message latency is therefore guaranteed by meeting the publishers and subscribers deadlines. The maximum message latency, R_{msg}^{INC}, using the incremental GC is observed to be achieved.

Due to the incremental GC execution, the observed maximum message latency, R_{msg}^{INC}, in the status router is less than the maximum latency, R_{msg}^{STW}, when GC is carried out in a stop-the-world fashion for both of the scheduling mechanisms. The stop-the-world GC has approximately

11.46% overhead in CPU time during the status router's execution, while the incremental GC has approximately 16.34% overhead in CPU time during the status router's execution using the same heapsize and triggering policy. The smaller message latency achieved by using the incremental GC comes with a larger GC overhead in CPU time than the stop-the-world GC.

The next chapter summarizes the contributions of the thesis and discusses the future work.

CHAPTER 7

CONCLUSION AND THE FUTURE WORK

This dissertation develops a systematic approach to the application of RTGC in the real-time realm. The approach was motivated by the need for a uniform method to characterize GC-enabled real-time systems in order to assist real-time developers to utilize GC in a predictable way.

The original inspiration of the dissertation's approach is the Giotto system, [55], whose key contribution is that it supports "the automation of real-time control system design by strictly separating platform-independent functionality and timing concerns from platform-dependent scheduling and communication issues," [55]. Similarly, given a system with bounded workload, our systematic approach calibrates the control of RTGC by separating platform-independent functionality and timing concerns from the timing-related memory-management issues that depend on the details of a real-time system's implementation.

In summary, the key contributions of this dissertation are the following:

- **The system model** provides a consistent way to describe GC-enabled real-time systems. It covers both modeling of entities and entity interactions within the real-time systems. The system model shows how to characterize a given real-time system with workloads and illustrates necessary components to compute GC scheduling parameters in order to assist developers to use GC in a predictable way. Within the system model the **RTGC framework** models the GC task.

- **The GC cost model** factors the contributions of the GC implementation and the application-imposed workload. Separation of these aspects allows prediction of system behavior from knowledge of component behavior and environmental specifications. The complete GC cost model consists of a set of *GC cost coefficients*, a set of *application-dependent parameters* and a set of *rules* that allow calculation of GC-related costs.

- **The GC-integrated scheduling mechanisms** compute GC scheduling parameters, ξ_{GC}, so that the overall system satisfies its real-time requirements. Two RTGC configuration plans, work-based, ξ_W, and time-based, ξ_T, have been developed and implemented. The work-based RTGC configuration plan, ξ_W, is scheduled to guarantee a minimum mutator utilization, μ_{MMU}, over time duration, Δt. The time-based RTGC configuration plan, ξ_T, is scheduled to meet specific real-time tasks deadline requirements using fixed-priority assignments. *C-space* feasibility analysis, [15], is used in the time-based RTGC configuration, ξ_T.

Recall that the thesis of the dissertation is

> ***Thesis:*** Decomposing the model of (1) the performance of collector operations and (2) the garbage collection load offered by a real-time task set allows prediction of system behavior from knowledge of component behavior and environment specifications. This approach provides the models needed to engineer GC into real-time applications.

The thesis has been demonstrated by the design, implementation, and demonstration of a systematic RTGC approach. The demonstration consists of a status router application with two publication-subscription workloads. Experiments show that the GC-enabled status router application using the derived GC scheduling parameters meets its timing requirements with no memory exhaustion. For the work-based GC scheduling, ξ_W, the minimum mutator utilization requirements, ($\mu_{MMU}, \Delta t$), are observed to be met. For the time-based GC scheduling, ξ_T, the message latency requirement, R_{msg}, is met.

Moreover, due to the incremental GC execution, the observed maximum message latency, R_{msg}^{INC}, in the status router is less than the maximum latency, R_{msg}^{STW}, when GC is carried out in a stop-the-world fashion for both of the scheduling mechanisms. The average message latency, A_{msg}^{INC}, is less than the average message latency, A_{msg}^{STW}, using the stop-the-world GC as well. The stop-the-world GC has approximately 11.5% overhead in CPU time during the status router's execution, while the incremental GC has approximately 16.3% overhead in CPU time during the status router's execution using the same heapsize and triggering policy. The smaller message latency achieved by using the incremental GC comes with a larger GC overhead in CPU time.

7.1 Future work

The system model introduced in this dissertation provides a uniform way to describe a GC-enabled real-time system where GC is modeled using the RTGC framework. It also incorporates design and implementation of a systematic RTGC approach that computes GC scheduling parameters for a given real-time system's workload. The system model lays out the necessary components to assist real-time developers to use GC in a predictable way: a task model for the application real-time task set plus a GC task; a GC cost model estimating the worst-case GC-related costs; and a scheduling mechanism deriving GC execution parameters. Future work would enhance each of these three components:

- For the task model component, adding more real-time task models would allow GC to be applied in a wider range of real-time applications. For instance, a new task model called multi-frame real-time task model has been developed to support real-time audio video streaming applications, [23]. Integrating GC in the multi-frame model would allow the scheduling and schedulability analysis techniques developed for that task model to be applied to real-time systems with GC.

- For the GC cost model component, building cost models for other GC implementations–such as copying collectors and generational garbage collectors–would make our systematic RTGC approach applicable to other runtime systems that provide GC. A cost model should abstract the computation of GC-related cost in terms of (1) the performance of collector operations and (2) the garbage collection load offered by real-time tasks. For a particular GC implementation, the performance of collector operations characterized by the GC cost coefficients should be defined using sufficient granularity that these coefficients are stable across various applications. As previously discussed, the GC implementation may have to be modified to achieve stability of coefficients.

  Another future work for the GC cost model component is to model interactions among real-time application, GC and scheduler to determine both application and GC execution dependent parameters, maximum memory allocation, A_{max}, maximum garbage needing to be reclaimed, G, *etc.*. Eliminating outliers of GC cost coefficients will eventually reduce GC cost estimation pessimism instead of using the adjusted GC cost coefficients.

  This dissertation work only considers that a GC cost model needs be *conservative*, which is evaluated using the defined pessimism metric, ρ^M. Future work for assessing a GC cost model is to formalize cost model's safety modes with which the cost model can be assessed. In fault-tolerant systems, a set of failure mode assumptions are formalized using partially-ordered failure mode assertions, [82]. A notion of *assumption coverage* is introduced to quantitatively assess system dependability. It represents the degree (probability) to which a component failure mode assumption proves to be true in the real system. The methodology of failure assumptions used in fault-tolerant systems is one potential approach to assess GC cost model's safety (dependability). The implicit assumptions established in the GC cost model during experiments can be formalized so that GC cost model dependability can be assessed quantitatively.

- For the GC scheduling component, RTGC scheduling mechanisms can be extended to work in real-time systems that are modeled using advanced system design methods. Recently, approaches facilitating the design of complex systems, such as object-oriented methods and component-based methods, have been developed to model and design real-time systems with increasing complexity. At the same time, researchers in the real-time system community are actively working on the integration of these advanced design methodologies and analytical real-time scheduling theory. Such work includes the integration of real-time schedulability analysis with *UML-RT* object-oriented standard, [47, 98], and compositional schedulability analysis for component-based hierarchical scheduling frameworks, [34, 38]. When real-time systems are designed using these advanced system methods, extending RTGC scheduling mechanisms for these design methodologies would allow use of GC in these real-time applications.

  Another kind of future work is to make RTGC mechanisms adaptive to a changeable workload during runtime. The variation in the workload should be modeled and taken into account by GC-integrated scheduling. Feedback scheduling, [83], where the current scheduling choice is made based on feedback from previous scheduling knowledge is one potential approach.

With these extensions, the system model will cover a wider range of real-time systems (more comprehensive task model), provide more choices of runtime platforms (more GC cost models), and have an enhanced GC-integrated scheduling capabilities (more schedulability analysis techniques and adaptive scheduling). Such extensions to the system model will serve real-time developers with even better capability to use GC in a predictable way.

APPENDICS

APPENDIX A

DISSERTATION GLOSSARY

Table A.1: Nomenclature

Definition context	Symbol	Description
GC cost	P_i	GC phase i among the ten phases in one GC cycle
	T_{P_i}	GC phase execution time
	δ_x	GC cost coefficients for incremental GC phases
	T_{fin}	Fixed cost for atomic GC finalization initialization subphase
	T_{sp}	Fixed cost for atomic GC sweep initialization subphase
	ρ^m	Cost estimation pessimism of metric m
RTGC incrementality	I_i	GC increment in GC phase P_i
	a_i	The atomic GC work unit in defined in GC phase P_i
	k_i	The number of atomic GC work unit a_i that consists of one GC increment I_i in GC phase P_i
	t_i	The execution time for GC increment I_i
Write Barrier cost	δ_{wb}	Write barrier cost coefficients
	W_x^m	The maximum number of various pointer update statistics
	σ_{wb}	write barrier cost
Application parameters	θ_t	Upper bound on reference density in thread stack per byte
	θ_h	Upper bound on reference density in live heap per byte
	θ_{h_v}	Upper bound on valid reference density in live heap per byte
	L	Maximum live memory size in bytes
	U_{max}	Maximum uncollectable memory in bytes during application execution
	A_{max}	Maximum memory allocated in bytes during application execution
	G_l	Maximum garbage in bytes needing to be reclaimed during phase P_7
	G_s	Maximum garbage in bytes needing to be reclaimed during phase P_8
System-specific parameters	H	The total available heap size
	B	The block size in bytes
	M_t	The maximum thread stack size in bytes
	N_T	The maximum number of active threads in system
Task model	Γ_N, Γ_{N+1}	A real-time task set with N tasks or N task plus the GC task
	τ_i, tau_{GC}	Real-time task i, real-time garbage collection task
	C_i, C_{GC}	The worst-case execution time of real-time task τ_i
	$C_i \cdot \sigma_i$	The worst-case execution time of real-time task τ_i with write barrier overhead included
	T_i, T_{GC}	real-time task τ_i's period (or the minimum interarrival time for sporadic task)
	D_i, D_{GC}	Real-time task τ_i's relative deadline
	A_i	Real-time task τ_i's worst-case allocation during its one request
	B_i	Block time encountered by real-time task τ_i due to synchrnization
	U_i	Utilization factor of real-time task τ_i, calculated as $U_i = \frac{C_i}{T_i}$
	Dep	Real-time tasks dependency specification $Dep = (D, \overline{D})$, where D and $\overline{D}$ describe dependent real-time tasks and independent ones
GC Scheduling and schedulability analysis	ν_{GC}	GC scheduling resolution
	ξ_W, ξ_T	Work-based (or time-based) RTGC configuration (GC scheduling)
	P_{GCMI}	The summary of the time quantum for mutator and GC tasks, where $P_{GCMI} = P_{MI_i} + k_i \cdot t_i$ in work-based GC scheduling plan ξ_W
	P_{MI}, P_{GCI}	The time quanta assigned to mutator (or GC task)
	R_{GC}	The worst-case GC response time
	μ_{MMU}	The minimum mutator utilization
	E_{max}	The total amount of GC cycles during one complete application's execution
	F	The free memory threshold used to trigger GC cycle
Status Router	T_{pm}	the period of producer task τ_{pm} per incoming socket in status router
	T_{cn}	the period of consumer task τ_{cn} per outgoing socket in status router
	A_{pm}	the worst-case allocation during one invocation of producer task τ_{pm}
	$\overline{A}_{cn}$	the worst-case deallocation(garbage generated) during one invocation of consumer task τ_{cn}
	$pubInt_m^i$	the publishing interval of the $i-$th publication served by producer task τ_{pm}
	$pubInt_n^j$	the subscribing interval of the $j-$th subscription served by consumer task τ_{cn}

APPENDIX B

GC COST MODEL

Figure B.1 plots the calculated fractions for GC uncollectable cost coefficient δ_{U_e} observed over 150 GC uncollectable increments. The GC uncollectable increment is time-based with size of $T_{I_2} = 1000\mu$s. The maximum fraction is 0.20%, which means there is only at most 0.20% measurements larger than the 99.9th percentile for all these observed GC uncollectable increments. Figure B.2 and B.3 show similar distributions for $\delta_{M_{mp}}$ with heap mark increments and δ_{S_l} with sweep large increments.

Figure B.1: Cost coefficient δ_{U_e} Measurement Distribution in GC increments I_2: GC uncollectable increment instances I_2 vs. Calculated fraction of the 99.9th percentile in I_2 (%) ($T_{I_2} = 1000\mu$s)

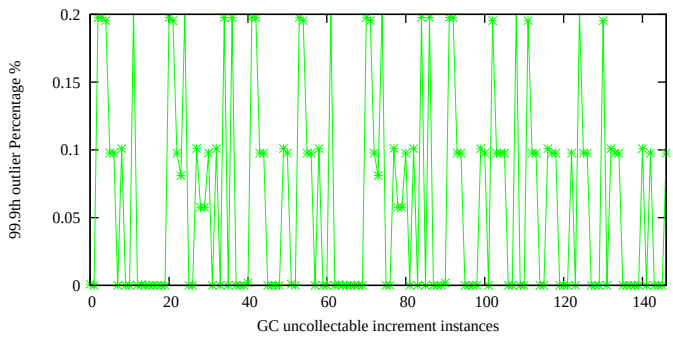


Figure B.4 plots value distributions of the measurements for the GC cost coefficient δ_C in GC clear increment I_1 with size k_1: $2500, 5000, 8000$[1]. The complete cost coefficients measurement set is gathered for the worst-case GC heap marking increment encountered by 100 application runs. The point (x, y) in each curve means that among the complete set of cost coefficient measurements,

[1]Recall that the atomic GC work a_1 is processing one block in GC clear phase P_1.

Figure B.2: Cost coefficient $\delta_{M_{mp}}$ Measurement Distribution in GC increments I_5: GC heap marking increment instances I_5 vs. Calculated fraction of the 99.9th percentile in I_5 (%) ($T_{I_5} = 2500\mu$s)

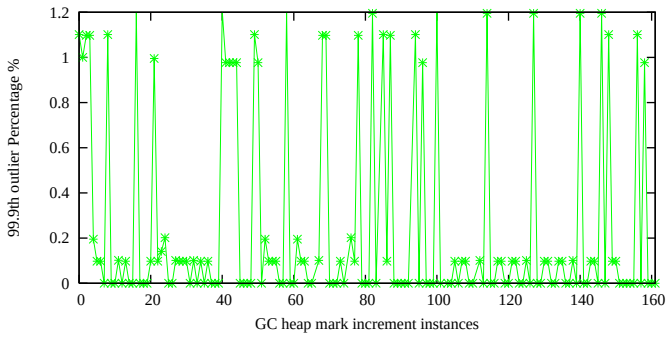
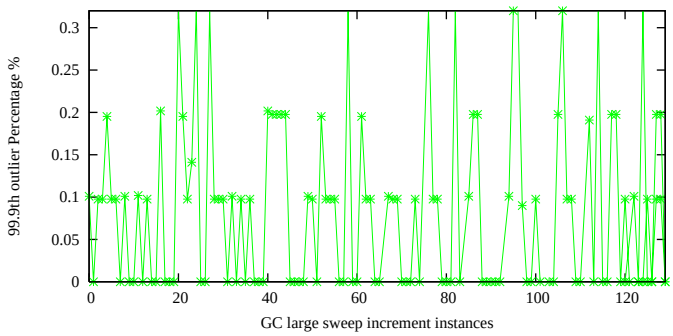


Figure B.3: Cost coefficient δ_{S_l} Measurement Distribution in GC increments I_7: GC sweep large increment instances I_7 vs. Calculated fraction of the 99.9th percentile in I_7 (%) ($T_{I_7} = 1500\mu$s)



there are y percent having values larger than x.

The data show that the adjust worst-case cost coefficients awc_1 are approximately the 99th percentile of the measurements in the worst-case GC heap marking increment. Other cost coefficients have observed to have similar distribution characteristic using the worst-case GC increments as shown in figure B.5 for δ_{S_l}.

Figure B.4: Cost coefficient δ_C Measurement using the worst-case GC increment I_1 (CPU cycle #) vs. Percentage (%)

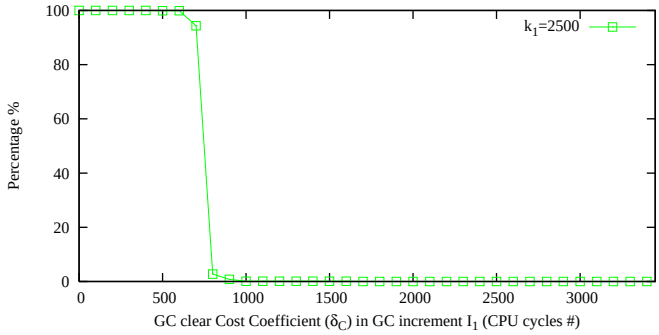


Figure B.5: Cost coefficient δ_{S_l} Measurement using the worst-case GC increment I_7 (CPU cycle #) vs. Percentage (%)

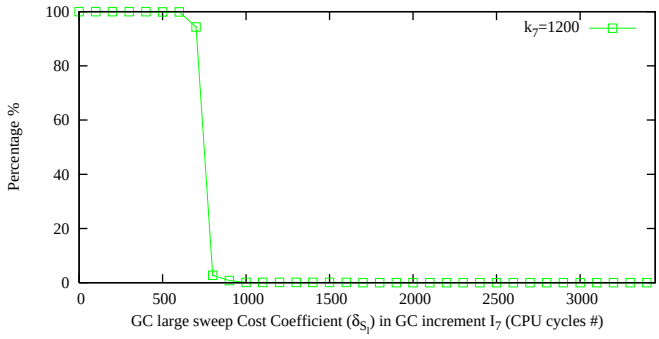


Table B.2 and B.3 report the adjusted worst-case cost coefficient measurements, where awc_1 and awc_2 are determined at 99.9th and 99.8th percentiles of the measurements respectively. These adjusted cost coefficients are stable across these applications.

Table B.1: The GC Cost Model Components for the mark-and-sweep collector

GC implementation aspects			Application implementation aspects	
GC phase	**Cost Coefficient**	**Definition**	**Parameters**	**Definition**
P_0 (GC_Start)	T_{P_0}	Fixed cost of atomic GC start phase (μs)	θ_t	Upper bound on reference density in thread stack per byte
P_1 (GC_Clear)	δ_C	Clear one block's mark bits (μs)	θ_h	Upper bound on reference density in live heap per byte
P_2 (GC_Uncollect)	δ_{U_e}	Get one allocated block and check whether it is uncollectable (μs)	θ_{h_v}	Upper bound on valid reference density in live heap per byte
	δ_{U_p}	Push pointers to all objects in one uncollectable block (μs)	θ_{dl}	Upper bound on disappearing link density in live heap per byte
P_3 (GC_nonthreadroot)	T_{P_3}	Fixed cost for atomic GC nonthread stack root scan phase (μs)	L	Maximum live memory size in bytes
P_4 (GC_threadscan)	δ_{R_r}	Scanning rate of thread stacks for pointers (μs/Byte)	U_{max}	Maximum uncollectable memory in bytes
	δ_{R_e}	Examine one pointer in thread stack (μs)	A_{max}	Maximum memory allocated in bytes
	$\delta_{R_{mp}}$	Mark and push a valid pointer to the main marking stack (μs)	G_l	Maximum garbage in bytes needing to be reclaimed during phase P_7
	$\delta_{R_{bl}}$	Record a value in blacklist (μs)	G_s	Maximum garbage in bytes needing to be reclaimed during phase P_8
P_5 (GC_heapmark)	$\delta_{M_{sr}}$	Scanning cost of an entry in main mark stack (μs/Byte)	**Write Barrier cost aspects**	
	δ_{M_e}	Examine one pointer in heap (μs)	$\delta_{WB_{C_1}}$	Check GC status in lines (1) and (4) in Write Barrier Algorithm 1
	$\delta_{M_{mp}}$	Mark and push a valid object in heap to the marking stack (μs)	$\delta_{WB_{C_2}}$	Check whether the pointer is marked or not in line (2);
	$\delta_{M_{bl}}$	Record a value in the blacklist during heap marking (μs)	δ_{WB_P}	Push a pointer to the mark stack in line (3) and (5)
P_6 (GC_Finalize)	T_{fin}	Fixed cost for atomic GC finalizing initialization (μs)	**Global system-specific parameters**	
	$\delta_{F_{dl}}$	Process one disappearing link in heap (μs)	B	The block size in bytes
P_7 (GC_Sweeplarge)	T_{sp}	Fixed cost for atomic GC sweep initialization (μs)	M_t	Maximum thread stack size in bytes
	δ_{S_r}	Scanning rate of heap to reclaim garbage for both P_7 and P_8 (μs/byte)	N_T	Maximum number of active threads in system
	δ_{S_l}	Reclaim one empty block (μs)	H	The total available heap size in system
	$\delta_{S_{enq}}$	Enqueue small object reclamation work into reclaim list to be processed in P_8 (μs)		
P_8 (GC_Sweepsmall)	δ_{S_s}	Reclaim unused small objects in one block (μs)		
P_9 (GC_Reset)	T_{P_9}	Fixed cost for atomic GC reset phase		

Table B.2: Adjusted worst-case RTGC cost coefficients (a)

Application		S_1		S_2				S_4					
		δ_C		δ_{U_e}		δ_{U_p}		δ_{R_e}		$\delta_{R_{mp}}$		$\delta_{R_{bl}}$	
		CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs
GC-stress	wc	35101	19.193	28699	15.692	7194	3.933	29205	15.969	27068	14.807	2552	1.395
	awc_1	957	0.523	594	0.325	6084	3.328	968	0.529	935	0.511	2398	1.311
	awc_2	693	0.379	528	0.289	4708	2.575	792	0.433	693	0.379	1903	1.041
	avg	649	0.355	151	0.083	5956	3.257	782	0.428	154	0.084	561	0.307
GCBench	wc	35981	19.675	28820	15.758	6864	3.753	28959	15.842	26540	14.519	2464	1.347
	awc_1	990	0.541	473	0.259	5902	3.329	979	0.535	924	0.505	2211	1.209
	awc_2	693	0.379	407	0.223	4516	2.470	770	0.421	638	0.349	1617	0.884
	avg	643	0.352	148	0.081	5635	3.083	974	0.533	148	0.081	717	0.392
BinaryTree	wc	36982	20.222	28996	15.834	6512	3.561	29880	16.346	28816	15.764	2453	1.341
	awc_1	990	0.541	535	0.293	5813	3.178	1001	0.548	495	0.271	2343	1.281
	awc_2	643	0.352	451	0.247	4518	2.471	792	0.433	693	0.379	1386	0.758
	avg	650	0.355	148	0.081	5908	3.230	979	0.535	146	0.080	942	0.515
Graph	wc	34507	18.869	28820	15.758	6622	3.621	29101	15.920	26470	14.480	2508	1.371
	awc_1	968	0.529	385	0.211	5681	3.108	979	0.535	770	0.421	2046	1.119
	awc_2	639	0.349	363	0.198	4316	2.361	792	0.433	550	0.301	1595	0.872
	avg	703	0.355	150	0.082	5157	2.821	773	0.423	146	0.080	905	0.495

Table B.3: Adjusted Worst case RTGC cost coefficients (b)

Appli-cation		S_5						S_6		S_7, S_8					
		δ_{M_e}		$\delta_{M_{mp}}$		$\delta_{M_{bl}}$		$\delta_{F_{dl}}$		δ_{S_l}		δ_{S_s}		$\delta_{S_{enq}}$	
		CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs	CPU_C	μs
GCstress	wc	32494	17.768	30668	16.769	2574	1.407	1139	0.623	35123	19.205	29513	16.138	5070	2.773
	awc_1	891	0.487	418	0.229	2233	1.221	998	0.546	1848	1.010	10505	5.744	2108	1.153
	awc_2	495	0.271	297	0.162	1597	0.874	890	0.487	979	0.535	6458	3.533	1331	0.728
	avg	441	0.241	122	0.067	1408	0.770	497	0.272	866	0.474	2034	1.112	300	0.164
GCBench	wc	32989	18.039	32340	17.683	2474	1.353	1254	0.686	35189	19.242	33429	18.279	5269	2.881
	awc_1	957	0.523	495	0.271	2113	1.156	1017	0.556	1914	1.047	10527	5.756	2013	1.101
	awc_2	495	0.271	297	0.162	1548	0.847	912	0.499	990	0.541	5896	3.224	1716	0.938
	avg	417	0.228	119	0.065	1496	0.818	503	0.275	938	0.513	2171	1.187	289	0.158
BinTree	wc	32186	17.598	32131	17.568	2274	1.244	1111	0.607	36477	19.955	33986	18.592	5071	2.772
	awc_1	979	0.535	495	0.271	2019	1.104	956	0.523	1991	1.089	10571	5.780	1991	1.089
	awc_2	495	0.271	297	0.162	1240	0.678	874	0.478	990	0.541	6435	3.519	1540	0.842
	avg	419	0.229	119	0.065	1435	0.785	481	0.263	944	0.516	2222	1.215	309	0.169
Graph	wc	34320	18.766	29700	16.240	2650	1.450	1098	0.601	37356	20.426	30877	16.884	4895	2.678
	awc_1	990	0.541	495	0.271	2254	1.233	945	0.517	1518	0.830	9746	5.329	2014	1.102
	awc_2	495	0.271	297	0.162	1682	0.920	901	0.493	957	0.523	6039	3.302	1237	0.677
	avg	384	0.210	121	0.066	1506	0.824	512	0.280	814	0.445	2197	1.201	285	0.156

APPENDIX C

GC SCHEDULING MECHANISMS

Figure C.1: Source code for schedulability analysis using *C-space*

```
private double calcWorkload(int i, double b){

        int f,c;
        double branch0, branch1;
        Task t_i;

        if (i<=0) return 0;
        if (b<=Global.lastA[i]) return Global.lastWorkload[i];

        t_i = (Task)tasks[i];
        f = (int)Math.Floor((double)b/(double)t_i.TI);
        c = (int)Math.Ceiling((double)b/(double)t_i.TI);

        branch0 = b-f*(t_i.TI-t_i.CI)+calcWD(i-1,f*t_i.TI);
        branch1 = c*t_i.CI + calcWD(i-1,b);
        Global.lastA[i] = b;
        if (branch0 < branch1)  Global.lastWorkload[i] = branch0;
        else Global.lastWorkload[i] = branch1;

        return Global.lastWorkload[i];
    }


public bool FPTest(){
        int i;
        Task t_i;

        for (i = 1; i<=tasks.Count; i++){
            t_i = (Task)tasks[i-1];

            if (t_i.CI + this. calcWorkload(i-1,t_i.DI) > t_i.DI)
                return false;

        }

            return true;
    }
```

APPENDIX D

EXPERIMENTAL DEMONSTRATION

Two publisher-subscriber workloads are studied in this chapter, and their specifications are shown in

Figure D.1 shows the specifications of the two publisher-subscriber workloads studied in chapter 6. Workload TS_1 has lighter loads than TS_2 that TS_1's overall incoming packet rate is 437.22p-(acket)/s(econd) and outgoing packet rate is 389.59p/s while TS_2's overall incoming packet rate is 12091.67p/s and outgoing packet rate is 4175.00p/s. T_{Gamma} is the status router execution's period calculated using Equation (6.4).

Table D.1 and table D.2 report write barrier calculations for the task set TS_1 and task set TS_2.

Figure D.2 shows the codes to determine GC period in accordance to the maximum message latency for the input workload. The function *Cspace_sched* takes the following inputs: the chosen GC increment size P_{GCI} and mutator increment size P_{MI}; application-dependent GC cost estimations Υ, δ_{S_r} (in Equation 5.9 in page 117) and maximum live memory L. It outputs the GC period

Table D.1: Workload 1: the calculation of write barrier cost C_{wb} (μs, $\#/\mu s$)

Workload TS_1	W_{total}	U_{GC}^{max}	$\frac{2T_{P_4}+T_{P_5}}{C_{GC}}$	C_m	$\widehat{C}_{wb}$	C_{wb}	$\widehat{\sigma}_{wb}$	σ_{wb}	r_M	$\rho^{\sigma_{wb}} = \frac{\sigma_{wb}}{\widehat{\sigma}_{wb}}$
τ_{p_1}	107	0.871	0.719	136	22.58	211.21	1.166	**2.553**	0.787	2.188
τ_{p_2}	310	0.875	0.731	860	30.96	626.94	1.036	**1.729**	0.360	1.667
τ_{p_3}	1797	0.878	0.738	4618	143.16	3675.93	1.031	**1.796**	0.389	1.742
τ_{p_4}	2292	0.879	0.737	5745	195.33	5026.88	1.034	**1.875**	0.399	1.813
τ_{c_1}	16	0.789	0.747	192	9.6	28.99	1.050	**1.151**	0.0833	1.096
τ_{c_2}	33	0.867	0.796	784	18.03	68.99	1.023	**1.088**	0.0421	1.064
τ_{c_3}	18	0.889	0.761	368	12.15	36.43	1.033	**1.099**	0.049	1.063
τ_{c_4}	21	0.886	0.751	432	13.39	43.2	1.031	**1.100**	0.048	1.067

Figure D.1: The Status Router Publisher and Subscriber Workload Configurations

Workload 1

Pub#1
Svar0 | Type0 |20ms

Pub#2
Svar1 | Type1 | 50ms
Svar2 | Type2 | 10ms

Pub#3
Svar3 | Type0 | 50ms
Svar4 | Type0 | 60ms
Svar5 | Type1 | 10ms

Pub#4
Svar6 | Type1 | 30ms
Svar7 | Type2 | 45ms
Svar8 | Type2 | 60ms
Svar9 | Type1 | 120ms
Svar10|Type0 | 20ms

Sub#1
Svar0 | 40ms
Svar1 | 200ms
Svar2 | 50ms
Svar6 | 100ms

Sub#2
Svar0 | 30ms
Svar2 | 120ms
Svar3 | 100ms
Svar9 | 200ms
Svar10 | 40ms

Sub#3
Svar1 | 40ms
Svar4 | 120ms
Svar5 | 10ms
Svar6 | 30ms
Svar7 | 120ms
Svar10 | 60ms

Sub#4
Svar0 | 160ms
Svar1 | 80ms
Svar8 | 120ms
Svar9 | 240ms
Svar10 |40ms

$\mathbf{T_r} = 7200$ms

Workload 2

Pub#1
Svar0 | Type0 |1ms
Svar1 | Type1 |1ms
Svar2 | Type2 |1ms
Svar3 | Type0 |10ms
Svar4 | Type0 |1ms
Svar5 | Type1 |2ms
Svar6 | Type1 |5ms
Svar7 | Type2 |4ms
Svar8 | Type2 |40ms
Svar9 | Type1 |20ms

Pub#2
Svar10 | Type0 | 1ms
Svar11 | Type1 | 1ms
Svar12 | Type2 | 1ms
Svar13 | Type0 | 1ms
Svar14 | Type1 | 5ms

Pub#3
Svar15 | Type2 | 2ms
Svar16 | Type2 | 1ms
Svar17 | Type0 | 5ms
Svar18 | Type1 | 1ms
Svar19 | Type0 | 15ms

Sub#1
Svar0 | 1ms
Svar1 | 1ms
Svar2 | 1ms
Svar6 | 5ms
Svar8 | 45ms
Svar9 | 40ms
Svar12 | 1ms
Svar14 | 15ms
Svar19 | 45ms

Sub#2
Svar1 | 4ms
Svar4 | 5ms
Svar5 | 10ms
Svar6 | 20ms
Svar7 | 5ms
Svar10 | 1ms
Svar11 | 10ms
Svar12 | 15ms
Svar18 | 1ms

Sub#3
Svar3 | 20ms
Svar9 | 20ms
Svar10 | 10ms
Svar12 | 10ms
Svar13 | 1ms
Svar15 | 5ms
Svar16 | 10ms
Svar17 | 20ms
Svar19 | 40ms

$\mathbf{T_r} = 360$ms

Table D.2: Workload 2: the calculation of write barrier cost C_{wb} (μs, $\#/\mu s$)

Workload TS_1	W_{total}	U_{GC}^{max}	$\frac{2T_{P_4}+T_{P_5}}{C_{GC}}$	C_m	$\widehat{C}_{wb}$	C_{wb}	$\widehat{\sigma}_{wb}$	σ_{wb}	r_M	$\rho^{\sigma_{wb}} = \frac{\sigma_{wb}}{\widehat{\sigma}_{wb}}$
τ_{p_1}	1020	0.953	0.675	23109	600.83	1849.12	1.026	**1.081**	0.044	1.054
τ_{p_2}	64	0.923	0.898	2034	22.38	138.61	1.011	**1.069**	0.032	1.057
τ_{p_3}	228	0.914	0.818	7701	123.22	458.58	1.016	**1.060**	0.029	1.043
τ_{c_1}	1940	0.957	0.558	24980	2073.34	3113.98	1.083	**1.125**	0.078	1.039
τ_{c_2}	270	0.910	0.495	2848	185.12	390.50	1.065	**1.138**	0.095	1.069
τ_{c_3}	44	0.939	0.491	496	20.83	64.52	1.042	**1.131**	0.089	1.086

Table D.3: Calculation of Status Router deadlines in Γ_{N+1} (ms, bytes) (1) Workload TS_1 with $R_{msg} = 0.823$ms and Workload TS_2 with $R_{msg} = 0.551$ms

	Tasks	$C_i \cdot \sigma_i$	D_i	T_i
TS_1	τ_{p_1}	0.347	1.17	20
	τ_{p_2}	1.487	6.425	50
	τ_{c_3}	0.405	19.34	120
	τ_{c_1}	0.221	10.097	200
	τ_{p_3}	8.294	42.037	300
	τ_{p_4}	10.772	49.453	360
	τ_{c_4}	0.475	22.70	480
	τ_{c_2}	0.853	41.180	600
TS_2	τ_{p_2}	2.174	5	5
	τ_{p_3}	8.163	28.324	30
	τ_{c_3}	0.561	11.442	40
	τ_{p_1}	24.981	40	40
	τ_{c_2}	3.241	50.231	60
	τ_{c_1}	28.103	360	360

T_{GC} that makes the task set Γ_{N+1} schedulable with a targeted maximum message latency.

The calculated deadlines are reported in

Table D.3 lists the calculated deadlines with a targeted maximum message latency, $R_{msg} = 0.823$ms for task set TS_1 and $R_{msg} = 0.551$ms for task set TS_2.

Figure D.2: Determine GC period T_{GC} and maximum message latency R_{msg}

```
public double Cspace_Sched(int PMI, int PGCI, int usingCGC, double A1, double A2, double B, long L){
        Task t_i;
        double tgc_init1, tgc_init2;
        long heap_init=0;
        double sumAITI=0;
        long Ainit=497792;
        int i=0, GC_insert_index=0;
        for (i=0;i<tasks.Count;i++){
                t_i = (Task)tasks[i];
                sumAITI+=(double)t_i.AI/(double)t_i.TI;
        }

        if(usingCGC==1) //using the worst-case GC cost estimations
        {
                tgc_init1 = (double)(A1 + B*L+(double)(PMI*PGCI)/(double)(PMI+PGCI))/((double)1/(double)(1+PMI/PGCI)-
                          2*B*sumAITI);
                tgc_init2 = (double)((A1 + B*L)/(1-2*B*sumAITI));
        }
        else{ //using the adjusted worst-case GC cost estimations
                tgc_init1 = (double)(A2 + B *L+(double)(PMI*PGCI)/(double)(PMI+PGCI))/((double)1/(double)(1+PMI/PGCI)-
                          2*B*sumAITI);
                tgc_init2 = (double)((A2 + B*L)/(1-2*B*sumAITI));
        }
        if (tgc_init1 > tgc_init2) gcTask.TGC = tgc_init1;
        else gcTask.TGC = tgc_init2;
        while (true)
        {
                heap_init = (long)(L + 2*gcTask.TGC*sumAITI) + Ainit;
                gcTask.calcCGC(heap_init, A1);
                gcTask.calcCAGC(heap_init, A2);
                gcTask.TI=gcTask.TGC;
                gcTask.DI=gcTask.TI;
                if (usingCGC= =1)          {gcTask.CI=gcTask.CGC;}
                else {     gcTask.CI=gcTask.CAGC;}

                //Order task set list with ascending priority order
                for (i=0;i<tasks.Count;i++){
                        t_i = (Task)tasks[i];
                        if (gcTask.TGC <=  t_i.TI){
                                tasks.Insert(i,(Task)gcTask);
                                GC_insert_index = i;
                                break; }
                }
                if(i= =tasks.Count){
                        tasks.Insert(i,(Task)gcTask);
                        GC_insert_index = i;
                }
                //Passing C-space schedulability analysis, output GC scheduling parameters
                if (this.FPTest())
                 {
                        this. CGC = gcTask.CGC;
                        this. CAGC = gcTask.CAGC;
                        this. HEAP = heap_init;
                        return gcTask.TGC;
                }
                else{
                        //First try to adjust GC  period
                        gcTask.TGC += CONST.TGC_INC;
                        tasks.RemoveAt(GC_insert_index);
                        //if Cannot find feasible GC scheduling parameters, try to increase targeted maximum message latency,
                         hence, the real-time tasks deadlines, and redo the whole schedulability analysis again
                        if (gcTask.TGC >=TS_period){
                                Global.latency=Global.latency + CONST.latency_inc;
                                //adjusting scheduling requirement in terms of latency
                                for (i=0;i<tasks.Count;i++){
                                t_i = (Task)tasks[i];
                                t_i.DI = t_i.CI+Global.latency*t_i.MI;
```



```
                                if (t_i.DI>t_i.TI) t_i.DI=t_i.TI;
                        }
                        if (tgc_init1 > tgc_init2) gcTask.TGC = tgc_init1;
                        else gcTask.TGC = tgc_init2;
                }
        }
```

BIBLIOGRAPHY

[1] Rdtsc. `http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1\\.HTM`.

[2] Improving real-time latency performance on COTS architectures. www.netacquire.com, July 2006.

[3] A.K.Mok and D.Chen. A multiframe model for real-time tasks. In *Proceedings of the 17th Real-time Systems Symposium, IEEE Computer Science Press*, 1996.

[4] K. Albers and F. Slomka. Efficient feasibility analysis for real-time systems with edf scheduling. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 492–497, Washington, DC, USA, 2005. IEEE Computer Society.

[5] M. V. Assche, J. Goossens, and R. Devillers. Joint garbage collection and hard real-time scheduling. In *Proceedings of 13th International Conference on Real-time Systems*, April 2005.

[6] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[7] D. F. Bacon, P. Cheng, and V. T. Rajan. Real-time garbage collector with low overhead and consistent utilization. In *ACM, POPL'03*, pages 15–17, January 2003.

[8] H. G. Baker, Jr. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[9] H. G. Baker, Jr. The treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, 1992.

[10] S. Balasubramanian, N. Natarajan, O. Franza, and C. Gianos. Deterministic low-latency data transfer across non-integral ratio clock domains. In *19th International Conference on VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design.*, page 5 pp., 3-7 Jan. 2006.

[11] C. S. Baruah. Scheduling real-time tasks: Algorithms and complexity.

[12] W. S. Beebee, Jr. and M. Rinard. An implementation of scoped memory for real-time Java. *Lecture Notes in Computer Science*, 2211:289–298, 2001.

[13] H. Ben-Abdallah, J.-Y. Choi, D. Clarke, Y.-S. Kim, I. Lee, and H. liang Xie. A process algebraic approach to the schedulability analysis of real-time systems. *Real-Time Systems*, 15(3):189–219, 1998.

[14] M. Bengtsson. *Real-Time Garbage Collection*. PhD thesis, Computer Science, Lund University, 1990.

[15] E. Bini and G. Buttazzo. Schedulability analysis of periodic fixed priority systems. In *IEEE Transactions on Computers*, volume 53, pages 1462 – 1473, November 2004.

[16] S. M. Blackburn, C. P., and M. K.S. A garbage collection design and bakeoff in jmtk: An efficient entensible java memory management toolkit. Technical Report TR-CS-03-02, Department of Computer Science, FEIT, Australian National University, September 2003.

[17] H. J. Boehm. Advantages and disadvantages of conservative garbage collection. `http://www.hpl.hp.com/personal/Hans\_Boehm/gc/issues.html`.

[18] H. J. Boehm and D. R. Chase. A proposal for garbage-collector-safe C compilation. *Journal of C Language Translation*, 4(2):126–141, December 1992.

[19] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 157–164, New York, NY, USA, 1991. ACM Press.

[20] H.-J. Boehm and M. Weiser. BDW collector library source code. `http://www.hpl.hp.com/personal/Hans\_Boehm/gc/\#where`.

[21] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. In *Software Practice and Experience*, number 9, pages 807–820, 1988.

[22] R. A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. *In proceedings of the 1984 ACM Symposium on Lisp and Functional programming*, pages 108–113, August 1984.

[23] S. Chakraborty and L. Thiele. A new task model for streaming applications and its schedulability analysis. In *Design, Automation and Test in Europe (DATE'05) Volume 1*, pages 486–491, 2005.

[24] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):667–678, November 1970.

[25] P. Cheng and G. Belloch. A parallel, real-time garbage collector. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 125–136, 2001.

[26] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 125–136, 2001.

[27] W.-N. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Predictable memory usage for object-oriented programs. Technical report, School of Computing, National University

of Singapore, `http://www.comp.nus.edu.sg/~chinwn/papers/main-mem.ps`, November 2004.

[28] D. Clarke, I. Lee, and H. Xie. Versa: A tool for the specification and analysis of resource-bound real-time systems, 1995.

[29] E. M. Clarke, J. O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[30] T. J. Consortium. Tuned dynamic real-time garbage collector. In *JOSES Technical Report 8208 (Task 4.2.2)*, May 2001.

[31] O. J. Dahl and K. Nygaard. *SIMULA - A language for programming and description of discrete event systems*. Norwegian Computing Center, Oslo, Norway, 5th Edition, September 1976.

[32] N. B. Dale, C. Weems, and M. R. Headington. *Introduction to Java and Software Design: Swing Update*. Jones & Bartlett Publishers, September 20 2002.

[33] K. Danne and M. Platzner. An edf schedulability test for periodic tasks on reconfigurable hardware devices. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers and tool support for embedded systems*, pages 93–102, New York, NY, USA, 2006. ACM Press.

[34] R. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10 pp.–, 5-8 December 2005.

[35] D. Detlefs. A hard look at hard real-time garbage collection. In *Object-Oriented Real-Time Distributed Computing, Proceedings. Seventh IEEE International Symposium*, pages 23–32, May 2004.

[36] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[37] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*. ACM Press, 2003.

[38] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental schedulability analysis of hierarchical real-time components. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 272–281, New York, NY, USA, 2006. ACM Press.

[39] ECMA. Ecma-335 common languange infrastructure. `http://msdn2.microsoft.com/en-us/netframework/aa569283.aspx`, 2002.

[40] T. V. Eicken, V. Avula, A. Basu, and V. Buch. Low-latency communication over ATM networks using active messages. Technical Report TR94-1456, 31, 1995.

[41] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems, 2001.

[42] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theor. Comput. Sci.*, 354(2):301–317, 2006.

[43] E. Fersman and W. Yi. A generic approach to schedulability analysis of real-time tasks. In *Nordic Journal of Computing archive*, volume 11, Summer, 2004.

[44] A. Fredette and R. Cleaveland. Rtsl: a language for real-time schedulability analysis. In *Real-Time Systems Symposium, 1993., Proceedings*, pages 274–283, December 1993.

[45] W. Fu and C. Hauser. A real-time garbage collection framework for embedded systems. In *Proceedings of the 9th International Workshop on Software and Compilers for Embedded Systems*, pages 20–26, October 2005.

[46] W. Fu and C. Hauser. Modeling real-time garbage collection cost. In *To be appearing in The 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, August 21-24 2007.

[47] Q. Gao, L. J. Brown, and L. F. Capretz. Integrating schedulability analysis with uml-rt. *Control Intell. Syst.*, 34(2):125–135, 2006.

[48] T. Gerdsmeier and R. Cardell-Oliver. Analysis of scheduling behaviour using generic timed automata. *Electronic Notes in Theoretical Computer Science*, 42, 2001.

[49] O. Gheorghioiu. Statically determining memory consumption of real-time Java threads. Master's thesis, Massachusetts Institute of Technology, 2002.

[50] K. H. Gjermundrød. *Flexible QoS-Managed Status Dissemination Middleware Framework For The Electric Power Grid*. PhD thesis, School of Electrical Engineering and Computer Science, Washington State University, August 2006.

[51] O. Goh, Y.-H. Lee, Z. Kaakani, and E. Rachlin. A schedulable garbage collection for embedded applications in CLI. In *RTCSA '05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pages 189–192, Washington, DC, USA, 2005. IEEE Computer Society.

[52] O. Goh, Y.-H. Lee, Z. Kaakani, and E. Rachlin. Integrated scheduling with garbage collection for real-time embedded applications in CLI. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC*, Washington, DC, USA, April 2006. IEEE Computer Society.

[53] J. Guy L. Steele. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[54] R. Henriksson. *Scheduling garbage collection in Embedded systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 1998.

[55] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded control systems development with Giotto. In *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 64–72, 2001.

[56] T. Higuera, V. Issarny, M. Banatre, G. Cabillic, and F. P. Jean-Philippe Lesot. Region-based memory management for real-time Java. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 387 – 394, 2001.

[57] P. Hudak, J. Fasel, and J. Peterson. A gentle introduction to haskell. Technical report, YALEU/DCS/RR-901, Yale University, May 1996.

[58] A. Ive, A. Blomdell, T. Ekman, R. Henrihsson, A. Nilsson, K. Nilsson, and S. G. Robertz. Garbage collector interface. In *Proceedings of NWPER'02, Copenhagen, Denmark*, August 2002.

[59] R. Iyer, H. jun Wang, and L. Bhuyan. Design and analysis of static memory management policies for CC-NUMA multiprocessors. Technical Report TR98-002, 18, 1998.

[60] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Storage Management*. John Wiley & Sons Ltd, 1996.

[61] T. Kim, N. Chang, N. Kim, and H. Shin. Scheduling garbage collector for embedded real-time systems. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 55–64, 1999.

[62] D. E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd Edition*. Addison-Wesley Professional, July 1997.

[63] T.-W. Kuo, Y.-H. Liu, and K.-J. Lin. Efficient on-line schedulability tests for priority driven real-time systems. In *RTAS '00: Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, page 4, Washington, DC, USA, 2000. IEEE Computer Society.

[64] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization andaverage case behavior. In *Real Time Systems Symposium, 1989., Proceedings*, pages 166–171, December, 5-7 1989.

[65] H. Lieberman and C. Hewitt. Garbage collector based on the lifetimes of objects. In *Communications of the ACM, , 26(6)*, pages 419–429, June 1983.

[66] M. Lindgren, H. Hansson, and H. Thane. Using measurements to derive the worst-case execution time. In *Proceedings of RTCSA 2000 Cheju Island, South Korea. IEEE Computer Society, 2000.*, 2000.

[67] G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*, pages 26–35, 2001.

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

[69] S. Liu and J. B. Goodenough.

[70] W.-C. Lu, J.-W. Hsieh, and W.-K. Shih. A precise schedulability test algorithm for scheduling periodic tasks in real-time systems. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1451–1455, New York, NY, USA, 2006. ACM Press.

[71] G. Madl and S. Abdelwahed. Model-based analysis of distributed real-time embedded system composition. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 371–374, New York, NY, USA, 2005. ACM Press.

[72] T. Mann, M. Deters, R. Legrand, and R. K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *Proceedings of the 2002 ACM Conferences on Languages, Compilers, and Tools for Embedded Systems(LCTES 2005)*, 2005.

[73] S. Manolache, P. Eles, and Z. Peng. Schedulability analysis of applications with stochastic task execution times. *Trans. on Embedded Computing Sys.*, 3(4):706–735, 2004.

[74] S. Martin. Real-time garbage collection for java. In *Object and Component-Oriented Real-Time Distributed Computing(ISORC),Ninth IEEE International Symposium*, April 2006.

[75] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. In *Communications of the ACM*, April 1960.

[76] S. J. Metsker. *Design Patterns C#*. Addison-Wesley Professional, 1st edition, April 15 2004.

[77] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[78] M. L. Minsky. A LISP garbage collector algorithm using serial secondary storage. Technical report, M.I.T. A.I lab., M.I.T., Cambridge, Mass., 1963.

[79] K. Nilsen. Issues in the design and implementation of real-time Java. *Java Developer's Journal*, 1(1):44, June 1996.

[80] P. Persson. Live memory analysis for garbage collection in embedded systems. In *Proceedings of the ACM Sigplan 1999 Workshop on Languages, compilers and Tools for embedded systems (LCTES'99)*, number 7, pages 45–54, June 1999.

[81] P. P. Pirinen. Barrier techniques for incremental tracing. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 20–25, New York, NY, USA, 1998. ACM Press.

[82] D. Powell. Failure mode assumptions and assumption coverage. In D. K. Pradhan, editor, *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, pages 386–395, Boston, MA, 1992. IEEE Computer Society Press.

[83] S. G. Robertz. *Flexible automatic memory management for real-time and embedded systems*. PhD thesis, Lund Institute of Technology, Lund University, 2003. ISSN 1404 - 1219.

[84] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. In *1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, Cyprus, October 2004.

[85] S.Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. In *Real-time Systems: The International Journal of Time-Critical Computing*, number 1, pages 99–128, 2003.

[86] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. In *IEEE Trans. Comput.*, number 9, pages 1175–1185, September 1990.

[87] A. Shaw. Reasoning about time in high-level language software. *IEEE Transactions on Software Engineering*, 15:875 – 889, July 1989.

[88] A. C. Shaw. *Real-time Systems and Software*. John Wiley & Sons Ltd, 2001.

[89] W. Shi. Implementation and performance of posix sporadic server scheduling in rtlinux.

[90] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[91] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate stack space and heap space analysis for high-level languages. Technical Report 538, Computer Science Department, Indiana University, `http://citeseer.ist.psu.edu/472671.html`, April 2000.

[92] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.

[93] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference*, pages 281–292, 2005.

[94] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management, Springer-Verlag Lecture Notes in Computer Science*, number Vol. 637, pages 1–42, 1992.

[95] Y. Xian and G. Xiong. Minimizing memory requirement of real-time systems with concurrent garbage collector. *ACM SIGPLAN Notices*, 40(3):40–48, March 2005.

[96] Ximian. Mono. `http://www.go-mono.org`.

[97] S. Yamane. Deductive schedulability verification methodology of real-time software using both refinement verification and hybrid automata. In *Computer Software and Applications*

Conference, COMPSAC, Proceedings. 27th Annual International, pages 527– 533, November 2003.

[98] S. Yau and X. Zhou. Schedulability in model-based software development for distributedreal-time systems. In *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, 2002. (WORDS 2002)*, pages 45–52, 2002.

[99] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, 1990.

[100] T. Yuasa, Y. Nakagawa, T. Komiya, and M. Yasugi. Return barrier - incremental stack scanning for snapshot real-time garbage collection. *International Lisp Conference*, October 26 2002.

[101] T. Yuasa, H. Saiki, Y. Konaka, T. Komiya, and M. Yasugi. Real-time gc in jerty^{Tm}vm using the return-barrier method. *ISORC*, pages 140–148, 2005.