A Foundation for the Development of Programming Languages for Real-Time Systems

Javad Ebrahimian Amiri

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

April 2021

Except where otherwise indicated, this thesis is my own original work.

Javad Ebrahimian Amiri
6 April 2021

to Mahboubeh, Hirad and my Parents

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Steve, for giving me the opportunity of doing a PhD, and making this hard and stressful journey enjoyable and more efficient by creating a friendly and positive atmosphere. For me, he is a role model of how a great supervisor should be.

I am also extremely grateful to my advisers Michael and Tony, for continuously supporting me throughout my PhD with their knowledge and patience. This work would hardly be possible without their insightful comments and suggestions.

Special thanks to Kunshan, Yi and John, for their high-quality works that provided the necessary foundations of this thesis.

I'd like to acknowledge the efforts of other µVM team members, specially Isaac for being very helpful even after his graduation.

Thanks also to all other members of the Computer Systems Research Group for their constructive comments and warm encouragement at the HDR monitoring sessions.

I would like to express my gratitude to those who have supported me financially: the Australian National University, the Australian Government and Data61 (formerly NICTA).

I am deeply indebted to my parents who supported me selflessly throughout my life and education.

And most of all, I owe my deepest gratitude to my spouse Mahboubeh. The completion of my dissertation would certainly not be possible without her patience, encouragement and personal support.

Abstract

Real-time systems have grown considerably in both diversity and popularity, and the demand for real-time software has never been higher. In contrast, the choice of programming languages used to develop these systems has mostly remained limited to decades-old languages, namely Ada and C/C++, and more recently real-time Java. We postulate that the main reason for this mono-culture is the difficulty of developing new programming languages for real-time systems, due to their strict correctness requirements.

Wang et al. [2015] argue that implementing even general-purpose languages is not easy, and is the source of many problems in today's languages. They propose the Micro Virtual Machine (µVM) as a minimal abstraction layer to relieve the challenges of implementing a managed language, and design a µVM specification named Mu. Compared to conventional language VMs, a µVM is minimal and low-level. We claim this makes a µVM an appealing platform for the development of programming languages for real-time systems, as it allows supporting a wide range of languages for diverse real-time systems. It also makes correct implementation and formal verification of the platform easier, which is vital for many real-time systems.

Prior to this thesis, there was only one concrete µVM specification [Mu, 2018]. However, Mu is not designed for real-time systems and lacks some essential features.

> My thesis is that a real-time-enabled micro virtual machine can provide an efficient and usable foundation for the development of programming languages suitable for building real-time software.

The first high-level contribution of this thesis is the design of RTMu, a µVM instance targeting programming languages for real-time systems. We build on the Mu specification and propose a set of modifications to its abstractions over concurrency and memory management to make it suitable for real-time systems.

The second contribution is the confirmation of the implementability of the RTMu's abstractions. For this purpose, we build a performant implementation of the RTMu specification, based on a performant implementation of Mu.

The third contribution is the design of a real-time extension to RPython, to make it a viable language for real-time systems, named RT-RPython. We implement RT-RPython on top of RTMu and evaluate its performance through the Collision Detection benchmark suite [Kalibera et al., 2009].

This thesis is a proof of concept, establishing the use of µVMs to build new high-quality programming languages for real-time systems. It also provides an empirical demonstration of performance and predictability for µVMs in the real-time domain. We believe that RTMu can help in tackling the current lack of diversity in programming languages for real-time systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis proposes a new foundation to address the current lack of diversity in programming languages for real-time systems. For this, we design, implement and evaluate the first micro virtual machine targeting real-time systems.

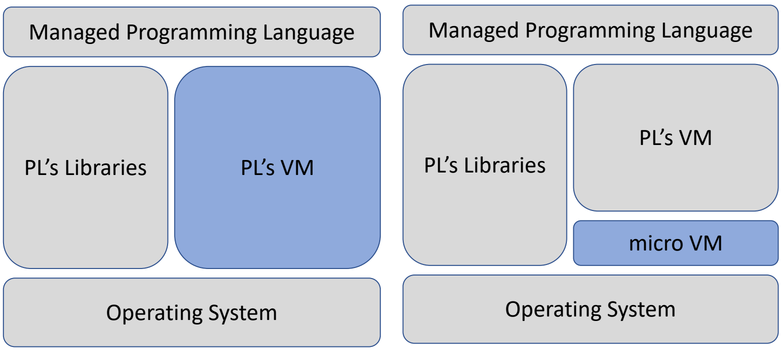
1.1 Motivation

A real-time system is a computer system in which the logically correct output must obey a timing constraint, often called a deadline. In addition to timeliness, real-time applications have other requirements, including throughput and reliability, at various levels of intensity. A soft real-time application like a video player demands high throughput, and missing deadlines only leads to reduced quality. In such systems, validation can often be done by running them a certain number of times with representative inputs and monitoring their performance and output. On the other hand, in hard real-time applications such as flight control systems, which are safety-critical, timing is paramount, and efficiency is secondary. The software subsystems in such applications are typically required to undergo rigorous testing and a level of formal verification.

Programming languages and their compilers and runtimes play a vital role in the compliance of real-time systems to their requirements and have been the subject of research since the 1960s. Many programming languages were developed or adapted for real-time systems. For instance, Stoyenko [1992] surveys around seventy such languages and goes so far as to estimate that *'the number of languages designed for or used in real-time programming is in the high hundreds or low thousands'* (as of 1992), and goes on to state that Ada was created by the U.S. Department of Defense due to concerns with maintaining *'over 1,000 languages'*. Clearly, only a few of those languages have survived. Currently, the choice of language for real-time systems is predominantly limited to Ada and C/C++, and more recently, real-time Java [Burns and Wellings, 2009]. This seems surprising given the diversity and popularity of real-time systems, and the flourishing ecosystem of general-purpose languages.

A.1.1-added this paragraph

One may argue that this low diversity means that the current choices are good enough. This is analogous to the argument that all programs can be written in machine code or the C language. This means although real-time systems are being



(a) Macro VM design (b) Micro VM design

Figure 1.1: A µVM is a minimal language-neutral abstraction layer over concurrency, compilation, and garbage collection. The bulk of the language-dependent features are implemented by the language-specific client VM, through Mu's abstractions. Contrarily, macro VMs such as variants of Java VM, provide many language-specific features. This means there will be less to do by the new client languages. However, this raises critical issues such as the semantics gap (e.g. different object layout and thread model) between the new client languages and the original one (e.g. Java) [Wang, 2018].

developed using the currently available languages, more choices can still have significant benefits. For instance, a high-quality managed languages such as Java is safer than C [Schoeberl et al., 2016] and can improve productivity and reusability [Pizlo et al., 2010a]. Hence, real-time software can be developed at a lower cost with higher reliability.

Part of the problem may be related to the level of reliability demanded of real-time language implementations. For example, it may be tolerable for a widely-used scripting language to crash or misbehave occasionally, but car brakes must always work. We believe that this explains why there is such a paucity of real-time languages, even while general-purpose languages have flourished.

Wang et al. [2015] argue that implementing even general-purpose languages is not easy. They claim that difficulty of implementation is the source of many problems (broken semantics, poor performance) in today's languages. Attacking this source problem, they propose the concept of the Micro Virtual Machine (µVM) as a thin abstraction layer over the three most challenging parts of implementing a managed language, namely: concurrency, compilation, and garbage collection. A schematic comparison of µVMs to conventional macro VMs is depicted in Figure 1.1. Wang [2018] proposes a particular µVM design instance called Mu, having a concrete specification, and Lin [2019] establishes its practicability through the implementation of real-world managed languages.

A.3.1-change1: added figure 1.1

1.2 Problem Statement

We argue that using a µVM instance, such as a real-time version of Mu (RTMu), to develop managed languages for real-time systems will help tackle the current monoculture of real-time languages for two reasons. First, it will relieve the difficulty and reduce the cost of developing new real-time languages. Second, it will bring the benefits of managed languages to the real-time domain [Bollella and Gosling, 2000].

This argument gives rise to a question, which we try to answer in this thesis: *'What is a suitable µVM design for real-time systems, and how can we verify its suitability?'*

In this thesis, we design, implement and evaluate RTMu. To ameliorate the complexity, we build on the Mu µVM specification and its high-performance implementation (Zebu). The determinative design property of RTMu, borrowed from Mu, compared to alternative VMs, is its minimality. This makes RTMu suitable for a wide range of languages, which can cover diverse real-time systems. In addition, the minimality of RTMu eases its implementation and will aid the task of formal verification. RTMu's amenity to formal verification helps in building a reliable platform, particularly for real-time systems which need rigorous testing and more formal validation such as safety-critical systems.

1.3 Scope and Contributions

In this section, we identify the scope of this thesis and our contributions.

RTMu Design The first goal of this thesis is to present *a design* for RTMu, which respects the design principles of a µVM, including minimality, and provides the necessary abstractions, as explained in Section 3.1. We do not claim to cover all real-time software systems.

Our contributions are:

1. Identification of core language features that distinguish real-time languages,
2. Design of the first µVM for real-time language implementation,
3. Specification of IR extensions to support real-time languages,
4. Specification of other runtime changes to support real-time languages,
5. Description of how to use the new primitives to conform to the requirements of a variety of real-time applications.

The ultimate goal of RTMu is to introduce a reliable platform that is flexible enough to serve as a target for a broad range of real-time systems.

RTZebu Implementation The second high-level goal of this thesis is to provide a performant and predictable implementation of the RTMu specification. It is not our goal to produce an implementation that can compete with highly developed, commercially used languages such as Ada, C or the Real-Time Specification for Java (RTSJ). Instead, we want to demonstrate its possibility by reaching a reasonable performance bar.

RTZebu is based on an existing performant implementation of Mu in Rust, named Zebu. Our main contribution by reaching this goal is to demonstrate that RTMu abstractions are feasible, mostly as light-weight wrappers around Real-Time Operating Systems' (RTOS) services.

RT-RPython and Evaluation This thesis proposes RTMu as an efficient and usable foundation for the development of programming languages suitable for building real-time software. We evaluate our claims as follows:

- *Usability*: We modify RPython and create RT-RPython as a language suitable for building real-time software. Then, we implement RT-RPython on top of RTMu to demonstrate its capability in implementing a real-world programming language for real-time systems.

- *Efficiency*: We implement an existing, standard real-time benchmark in RT-RPython and compare its average and worst-case performance to a commercially available implementation of RTSJ, and a recent version of Java.

We designed RTMu to cover a wide range of real-time applications and be verifiable. However, it is out of scope to evaluate these properties in this thesis. We only aim to provide an indicative, rather than comprehensive, evaluation of RTMu and RT-RPython.

1.4 Thesis Outline

Chapter 2 discusses background material on µVM and real-time programming languages which led to the current RTMu design. Chapter 3 presents the design of RTMu, a µVM on which managed programming languages for real-time systems can be developed. Chapter 4 discusses the important aspects of our high-performance implementation of RTMu, named RTZebu. Chapter 5 introduces RT-RPython, a language based on RPython for real-time systems, and explains how we implement RT-RPython on RTMu. Chapter 6 describes how we evaluate the performance of RT-RPython, presents the results of our evaluations, and argues its efficiency. Finally, Chapter 7 concludes the thesis and points out future work.

Chapter 2

Background and Related Work

Real-time systems are flourishing in number and diversity. From tiny in-body sensors to huge planes and industrial machines, real-time computer systems are increasingly ubiquitous. However, the programming languages for these systems suffer from a lack of diversity, while programming languages in the general-purpose domain are proliferating.

Addressing this issue requires recognizing its sources, which is what the first two sections of this chapter elaborate. We start with an introduction to real-time systems and the key concepts (e.g. protocols and algorithms) dedicated to their programming. This helps in realizing what makes real-time systems unique. Next, we briefly present some of the main challenges in implementing general-purpose programming languages. We mention what makes the situation even more challenging in real-time systems.

The last two sections of this chapter provide background information which informed the current design of RTMu. We discuss the most influential programming languages in the real-time domain. Indeed, these languages are the only survivors of the long history of real-time languages. Finally, we explain the µVM concept and introduce Mu, the µVM specification used as the basis for the RTMu design.

2.1 Real-Time Systems

A real-time system is any computer system where the correctness of its operation is dependent on both the logical and temporal correctness of the delivered response [Burns and Wellings, 2009]. Real-time systems consist of one or more real-time tasks. The temporal correctness constraint of a real-time task is often specified as a deadline, by when the task's outputs should be ready. Assuming a single-task system, the *worst-case execution time (WCET)* of the real-time task should be less than its deadline, to fulfil the correctness requirement. This is in contrast to non-real-time tasks, where *average-case execution time (ACET)* is the main concern.

Real-time tasks are classically divided into three categories according to how severe the consequences of missing a deadline are: *hard* real-time tasks which will cause total failure if a deadline is missed, with potentially catastrophic results; *firm* real-time tasks which are useless after they pass the deadline; and *soft* real-time tasks which will lose usefulness gradually after they pass the deadline. However, this categorization is too simplistic, and there are other works such as [Jensen et al., 1985] in the literature which try to overcome the limitations of this taxonomy.

A.2.1 and B.2.1-added citation and removed many

In the rest of this section, we briefly discuss some of the key differences between real-time and non-real-time (general-purpose) systems.

2.1.1 External Environment

Real-time systems are employed in diverse environments, from under the oceans (e.g. in submarines) to higher layers of the atmosphere (e.g. in satellites). A real-time system often has to respond to real-world events. It monitors its environment through sensor(s), processes the data, and produces outputs that may change its environment through actuators. A speed camera, for example, observes the speeds of the passing cars using a number of sensors, processes whether any car is going over the speed limit, and finally acts by taking a picture of that car. On the other hand, most general-purpose computers work in very similar environments which includes humans and/or other computers. Frequent use of various sensor and actuator devices in real-time systems, compared to the much less diverse I/O devices in general-purpose systems, highlights the importance of proper support for working with I/O devices in their software platforms.

2.1.2 Threads

Most of today's software applications consist of a number of tasks that are implemented as threads. While this is true for both real-time and general-purpose systems, their approaches to managing threads are significantly different. General-purpose systems often aim to provide fairness between threads and to improve the average throughput. In contrast, the most important goal in real-time systems is to maintain analyzability (predictability) and ameliorate the worst-case behaviour. The rest of this section discusses some of the key aspects of multi-threading in real-time systems.

Task Types

The most common type of task in real-time systems is *periodic*. These tasks are released at regular intervals called a *period* and have a deadline that is often equal to their period. Periodic tasks are often *time-critical* and must finish before their deadline for the correct operation of the whole system [Shin and Ramanathan, 1994]. Another common task type in real-time systems is *aperiodic*. These tasks are released as a result of an event, such as an error or a hardware interrupt, and may or may not have a deadline. In any case, these tasks should not jeopardize the deadlines of other tasks in the system.

Scheduling

Multi-threaded general-purpose applications are often developed without any assumption about how the threads are scheduled. Under the hood, threads are often scheduled using First-In-First-Out (FIFO) or Round-Robin (RR) algorithms (and their variants) by the underlying operating system. These applications work correctly as long as they are provided with a fair share of the system resources.

In contrast, designing a real-time application includes analysis of its timing behaviour, which is affected by thread scheduling. For instance, to design a hard real-time system, WCETs of tasks are estimated, and schedulability analysis is performed to prove meeting all deadlines. The scheduling is often performed by the underlying real-time operating system, but the real-time application developer needs to be able to control scheduling to achieve the desired timing.

Scheduling is one of the most investigated research topics in the real-time domain [Davis and Burns, 2011a; Bambagini et al., 2016]. Many scheduling algorithms have been designed, each targeting a range of real-time applications. In principle, most of these algorithms are based on or closely related to two algorithms: *earliest deadline first (EDF)* which dispatches the task with the closest deadline, and *rate monotonic (RM)* which dispatches the task with the smallest period. EDF is a dynamic-priority scheduling algorithm as the relative priorities of tasks may change at run time. On a single-processor computer, EDF is an optimal algorithm, meaning that if EDF cannot schedule a set of tasks, that set is not schedulable with any other algorithms. RM is an optimal fixed-priority algorithm. It has a lower schedulable utilization compared to EDF (69.3 % vs. 100 %) [Liu and Layland, 1973], but it also has less scheduler overhead.

A.2.3 and B.2.1-added two survey citations

Synchronization

The use of priority-based scheduling in real-time systems rises a new problem in synchronization of threads that happens when a high-priority thread waits to acquire a lock held by a low-priority thread. In this case, any medium-priority thread may preempt the running low-priority thread, and indirectly postpone the wakeup of the high-priority thread. This issue is called *unbounded priority inversion*, because medium-priority threads are delaying the dispatch of the high-priority thread despite its higher priority.

Sha et al. propose two solutions to eliminate the unbounded priority inversion problem [Sha et al., 1990]. The first is to temporarily raise the priority of the lock's owner to the high-priority task's priority when the high-priority task blocks. As a result, the low-priority task will not be preempted by medium-priority tasks. The priority of the lock owner is restored to its base value as soon as it releases the lock. This solution is called the *Priority Inheritance Protocol (PIP)*.

The second solution, called the *Priority Ceiling Protocol (PCP)*, defines a priority ceiling for each resource (lock), that is equal to the highest priority of any task which may lock that resource. When a task is blocked on a lock, the priority of the lock holder will be raised to the ceiling value of the lock, and this solves the unbounded priority-inversion problem. The advantage of PCP over PIP is that it avoids deadlocks that happen as a result of incorrect nesting of critical sections. However, it is harder to implement, and most platforms (e.g., Ada, RTSJ and RT-POSIX) employ simplified variants of PCP instead of the original protocol.

2.1.3 Memory

The choice of memory management in a real-time application affects its WCET significantly. On one end of the spectrum is *static* memory allocation that adds no run time overhead, but does not use memory efficiently. Alternatively, *manual* memory allocation/de-allocation, such as the `malloc` and `free` in C, has minimal average- and worst-case overhead, and can use the memory space very efficiently if exploited properly. The drawback of manual memory management is that it is error-prone, and this gets worse as the system scales up. Finally, there is *automatic* memory management, such as garbage collection (GC), which is easy to use and eliminates memory management errors, such as dangling references and memory leaks. GC has some potential shortcomings for being used in real-time applications: it may increase the WCET of applications, e.g. by introducing pauses or increasing the ACET; it is more complicated for the language implementer (not for the application developer) than other memory management alternatives, and hence harder to verify; and it has a memory space overhead. A number of works such as Bacon et al. [2003], Chang and Wellings [2010] and Pizlo et al. [2010b] considerably improve GC for real-time systems. However, some hard real-time systems will not use GC due to its potential delays [Burns and Wellings, 2009].

A.2.5 and B.2.1-metronome, schism and another RTGC

An alternative approach, named *scoped memory*, is used by the real-time specification for Java (RTSJ) to provide automatic memory reclamation. A scope is a memory pool with special lifetime charactristics where objects can be allocated. To allocate on a scope, a thread has to enter the scope. It is possible for multiple threads to enter a scope at the same time and use it as their allocation context. When all threads exit a scope, the memory associated with that scope is free to be reused. To prevent dangling references to objects in scopes, RTSJ enforces rules that forbid references from memory areas with longer lifetimes to objects in scopes with shorter lifetimes. Scoped memory does not impose the pause times of GC, and it has lower runtime overhead, at the cost of a more complicated programming model and not being able to use all Java libraries.

A.2.4-change-1

2.1.4 Safety Requirements

Many hard real-time computer systems are used in critical roles where failure can negatively affect human lives or lead to severe financial/environmental loss. Such systems are called *safety-critical*. For example, the computers that control brakes or airbags in a car can directly impact the passengers' safety, causing injury or even death.

The software in safety-critical systems is required to undergo strict validation and certification processes. For example, the software in airborne systems and equipment must conform to the standards specified in RTCA [2011] or EUROCAE [2011], to be approved for commercial use. To be eligible for the highest levels of assurance in such software safety standards, the application may only use a very restricted set of features available in languages such as Java and their libraries [Henties et al., 2009]. For instance, all safety-critical tasks in a system may only run as periodic handlers with private time-slices in one underlying thread. They may also be required to only use static memory allocation. More instances of such limitations are presented in Sections 2.3.1 and 2.3.3.

2.1.5 Operating Systems

Real-time systems are diverse and so are their operating systems. For example, there exist real-time systems for industrial robotics [Lippiello et al., 2007] and automation [Doukas and Thramboulidis, 2011] that use real-time variants of Linux. Other real-time applications use a small set of services from highly-configurable executives such as RTEMS, or even operate with no operating system.

Despite the huge diversity, there are requirements that are common to many *real-time operating systems (RTOS)*. Among them is that RTOS service delays must be bounded, regardless of the current system state (e.g., the number and states of tasks). This not only requires the correct choice of algorithms, but also an efficient implementation that has minimal non-preemptive critical sections (e.g., negligible interrupt disable time). Another common requirement of RTOSs is that they should be highly configurable, so that real-time applications are not burdened with unused features. Finally, like other software used in real-time systems, RTOSs are subject to various levels of validation and certification, which may enforce limitations on RTOS features. For example, in safety-critical systems, multitasking is only possible if the RTOS guarantees space and time partitioning. This allows each task to be verified separately, making verification of the whole system much easier.

2.2 Programming Language Implementation

Many of today's programming languages suffer critical issues like very poor performance and hard to reason about semantics. For instance, as shown in Figure 2.1, even some popular programming languages like Python and PHP may be up to 100 times slower than C. Wang [2018] argue that many of these issues can be traced back to the difficulty of properly implementing these languages. They identify three major concerns, namely execution (compilation), concurrency, and garbage collection, that contribute to the complexity.

Implementing an appropriate language gets even more difficult and costly when considering the requirements of real-time systems. For instance, the safety requirements of these systems, may add strict validation procedures or safety-certification to the language's development, which makes the whole process even more time-consuming and expensive. This is probably the most important reason why the

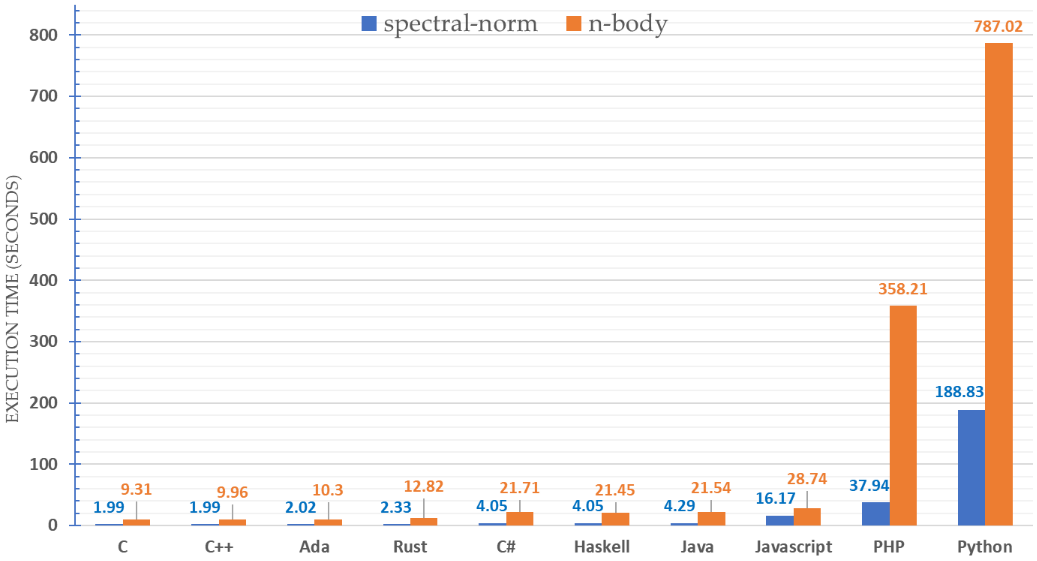


Figure 2.1: Execution times of two benchmarks on ten different programming languages according to the measurements of [LangBenchGame].

real-time community has been so reluctant to accept new languages.

2.2.1 Execution

Programming languages may be interpreted, ahead-of-time (AOT) compiled, just-in-time (JIT) compiled, or a mix of these. While an interpreter reads, decodes and executes the program source-code or byte-code at run time, a compiler produces machine code that is directly executed on the processor. Compared to compiled languages, an interpreted language has the advantages of ease of implementation and program portability, yet it lacks efficiency as it has to perform extra processing before accomplishing any operation.

Implementing an efficient compiler requires detailed knowledge of the target architecture. In AOT compilers, machine code is fully generated before run time, and a platform-dependant executable is produced. JIT compiled languages defer machine code generation to run time, when they compile the program on the user's machine. This allows distribution of the program in a machine-independent way.

JIT compilation introduces the new challenge of considering the run-time overhead of compilation. To minimize this overhead, JIT compilers start by quickly generating sub-optimal code, and then detect and optimize frequently executed code. For this purpose, the language runtime is required to utilize advanced language implementation techniques such as a proper profiling mechanism, and on-stack replacement (OSR) to quickly transition to the optimized code. In addition, compilation gets even more complicated when it is combined with concurrency and GC.

2.2.2 Concurrency

Concurrent programming models such as multi-threading provide programmers with tools to write correct concurrent applications. However, when it comes to running these applications on modern multi-core processors, certain properties, such as sequential consistency[1], cannot be assumed to be true anymore.

While absence of sequential consistency may lead to hard to reason about results on parallel hardware, enforcing it unconditionally is inefficient. The first attempt to enable solving this problem was the Java 1.5 memory model [Manson et al., 2005], followed by the C++11 memory model [Boehm and Adve, 2008]. A memory model describes the relation between threads and memory, to clarify which instruction reorderings can legally be done by the compiler optimizer or inside the processor. The language compiler is responsible for generating the instruction sequences that comply with the language memory model on each supported platform.

2.2.3 Garbage Collection

Garbage collection has been used in many modern programming languages since its first appearance in LISP [McCarthy, 1960]. The difficulty of implementing a high-performance GC has persuaded many language implementers to start language development with naive GCs [Wang, 2018], which not only degrades the overall performance, but is also very hard to upgrade to high-performance collectors [Jibaja et al., 2011].

A high-performance GC needs support from the compiler. The compiler should provide features such as *stack map*s to identify object references in stacks, and *write barriers* to monitor writes to reference fields. A high-performance GC may also use parallel hardware to run parallel and concurrent GC algorithms. In this case, it interacts with the memory model. It also relies on the compiler to insert yieldpoints to handle handshaking between application and GC threads.

2.3 Programming Languages for Real-Time Systems

Numerous programming languages have been built or adapted for real-time systems. However, only Ada, real-time Java, and C are widely used today. In this section, we review Ada SPARK, the Real-Time Specification for Java (RTSJ), Safety-Critical Java (SCJ), and C with Real-Time POSIX (RT-POSIX), as a representative set of RTPLs that cover a diverse range of real-time systems. The differences between these languages and their general-purpose counterparts inspire the design of RTMu.

[1] An execution of concurrent threads is called *sequentially consistent* if its result is the same as an execution where all operations from all threads are executed in a specific sequential order.

2.3.1 SPARK

SPARK is a subset of Ada for the development of high-integrity software [Barnes, 1997]. To satisfy the stringent reliability demands of such applications, it provides the tools to support various verification methods, from unit testing to formal proof of application properties. To enable this, SPARK restricts the hard-to-analyze features of Ada, including concurrency, memory management and synchronization [McCormick and Chapin, 2015]. SPARK supports multitasking through the Ravenscar tasking profile (a subset of tasking features) for high-integrity real-time programs, introduced by Burns et al. [1998]. The most important properties of this profile from the point of view of a real-time language developer are:

- No allocation or deallocation of objects and tasks at run time is allowed.
- Tasks are non-terminating with no user-defined attributes or dynamic priority.
- The dispatching algorithm within a priority level is FIFO.
- Non-preemptive scheduling is also allowed.
- Locking supports the Priority Ceiling Protocol (PCP).

A.2.7- added also

If RTMu is to be able to emulate SPARK's facilities, it will need to satisfy the following requirements: First, RTMu should provide an immortal memory area which allows allocation of objects, but is only deleted when the program finishes. Such an area could efficiently implement SPARK's static memory allocation. Second, RTMu should provide static priority scheduling. To support SPARK's scheduling, the RTMu scheduler does not need to support changing thread priority at run time. Third, the synchronization primitive in RTMu should support PCP.

2.3.2 RTSJ

RTSJ [Bollella and Gosling, 2000; RTSJ, 2018] is a specification for a type-safe Java-based managed programming language for large-scale real-time embedded systems that may co-locate hard-, soft- and non-real-time code. Language VMs such as FijiVM [Pizlo et al., 2009] and JamaicaVM [Siebert, 2010] are implementations of the RTSJ specification. RTSJ has been used in and proven adequate for many serious real-world applications such as Avionics, Automotive, industrial IoT and other critical embedded real-time applications [Sharp et al., 2003; Armbruster et al., 2007; AicasWebPage].

A.2.6 and B.2.1- added a paper citation

A2.2.2 and B.2.1-added one sentence mentioning fiji and jamaica

RTSJ introduces several enhancements over Java, mostly in the area of concurrency and memory management:

- The scheduler supports 28 levels of priority, compared to only 10 levels in Java. Within each level, First-In-First-Out (FIFO) and Round-Robbin (RR) are possible.
- Additional schedulers may be added besides FIFO and RR.

- Real-Time (RT) and No-Heap Real-Time (NHRT) threads are added. RT threads have higher priority than normal threads. NHRT threads are RT threads which do not use the heap.
- Heap, scoped and immortal memory areas are available. No specific Garbage Collection (GC) algorithm is specified, but its effects on preempting real-time tasks should be characterized by the RTSJ implementations.
- Threads may choose to only use the scoped memory, or both the heap and scoped memory at the same time. In the former case, GC should not impose any interference.
- Scopes may be allocated on specific areas of the physical memory.
- Raw access to physical memory is possible.
- Synchronization mechanisms utilize PIP, or optionally PCP to tackle priority inversion.

The set of features required to implement RTSJ is a superset of what is needed by other languages in this study, because RTSJ tries to cover a diverse range of real-time systems, while others such as SPARK and SCJ are more specialized.

If RTMu is to be able to emulate RTSJ's facilities, it will need to satisfy the following requirements:

- The static priority scheduler of RTMu should support changing thread attributes at run time.
- Synchronization primitives of RTMu should support PIP and PCP.
- RTMu should provide a garbage collected heap with analyzable preemption effects.
- RTMu should provide memory areas that can be created and deleted at run time.
- RTMu should support raw access to and allocation on physical memory.

2.3.3 SCJ

SCJ is a certifiable RTSJ-based language for safety-critical real-time systems. It is designed to comprise the minimal set of features for safety-critical systems [TheOpenGroup, 2017]. This minimality causes its programming model to be considerably different from the normal Java. Compared to other real-time programming languages, SCJ is fairly new and still under research and development.

There are three compliance levels in SCJ. Level 0 is the most restricted one, called the cyclic executive model. It consists of a sequence of missions, running on one processor. A mission is a set of Periodic Event Handlers (PEH). Each PEH has a dedicated private memory area, which is entered and exited at each release, and may create new ones. Private memory areas can be entered and exited but not shared with other PEHs. All PEHs share a mission memory and the immortal memory, but are not allowed to use synchronized methods.

Level 1 is also a sequence of missions. Each mission is a set of PEH and Aperiodic Event Handler (APEH) objects that may run concurrently. Event handlers are managed by a fixed-priority preemptive scheduler with at least 28 levels of priority. Each event handler may have a static processor affinity set. Level 1 event handlers can access the same types of memory areas as level 0 PEHs. They may also use synchronized methods.

Level 2 starts with a single mission but may create additional concurrent missions. In addition to PEHs and APEHs, each mission may contain No-Heap Real-Time Threads (NHRT). The private scoped memory of a NHRT is entered when it runs and exited when the run method returns. In addition to synchronized methods, level 2 applications may use `Object.wait()` and `Object.notify()`.

Schoeberl et al. [2017] summarize recent efforts on SCJ implementations, analysis tools and sample real-world applications. They state that it is difficult to learn programming in SCJ, due to its new programming model. In addition, its object oriented nature imposes a performance overhead which can make handlers with small periods infeasible.

From an application programmer's point of view, SCJ's abstractions are significantly different from those of RTSJ. However, from the RTMu's point of view, SCJ's abstractions can be implemented using a subset of the low-level tools required to implement RTSJ. A potential exception is the multi-level scheduler in SCJ, which looks different from the conventional priority-based schedulers, as in RTSJ. However, SCJ's scheduler can still be implemented on top of a scheduler that can implement the RTSJ's scheduler.

Although SCJ does not directly drive RTMu's abstractions, it has a critical implication to its design: RTMu has to be configurable, meaning that RTMu implementations should allow their client languages to include/exclude features based-on their target systems. Most importantly, a client like SCJ should be able to assume that it is not burdened by the complexities of GC.

2.3.4 RT-POSIX

Programming real-time systems in C is often done through RTOS interfaces. Many common RTOS implementations, such as RT-Linux, conform to the base POSIX standard and its real-time extensions (POSIX.1b), which are now merged in a single standard document [IEEE and TheOpenGroup, 2018]. Hence, RT-POSIX is commonly used for writing real-time applications. To summarize its implications for RTMu's design, RTMu's abstractions over concurrency and time are largely inspired by RT-POSIX, as it helps minimizing the implementation complexity, by keeping the abstractions close to the RTOS level.

2.4 The Mu Micro Virtual Machine

Mu is a µVM specification that facilitates the development of high-performance managed programming languages for a wide range of applications. It was first introduced by Wang et al. [2015], and the concrete specification is available online [Mu, 2018]. Mu abstracts over three basic language implementation challenges, namely concurrency, the compiler backend and memory management, and lets language implementers focus on higher level issues. Mu has a number of design principles:

- Mu observes minimalism, meaning that many features and optimizations are deferred to higher layers, as long as doing so does not jeopardize viability or efficiency of the three main functionalities.
- Mu assumes that language implementations (µVM clients) are trusted. Unnecessary overhead is thus avoided by excluding extra protection layers.
- Mu IR is modeled on LLVM IR, treating it as a baseline from which Mu diverges only when essential.
- Mu is a specification with well defined behavior, admitting multiple compliant implementations.

Architecture The high-level architecture of a managed language based on Mu is depicted in Figure 2.2. Although not drawn to scale in the figure, Mu is only a thin abstraction layer. The client does the bulk of the job of implementing a managed language on top of Mu's abstractions, which deal with the most difficult concerns.

A Mu client translates the source code or bytecode of an application to Mu IR, and uses the Mu client interface (API) to build and load IR bundles into a Mu instance. An IR bundle is the unit of code the client sends to Mu. It contains many Mu top-level entities, including a: type, function signature, constant, global cell, function, or exposed function. These bundles are then compiled to machine code. For example, the MuPy project presented by Zhang [2015], implements the RPython language [RPyDoc] as a client of Mu. In MuPy, application source code in RPython is first translated to an intermediate control flow graph (CFG) representation, with low-level types and instructions. Then, the types and instructions in the CFG representation are translated to Mu IR. Following that, the Mu API is used to build and load a Mu IR bundle from the translated CFG. Finally, Mu compiles the bundle to machine code and produces an executable.

Mu instances are expected to support Just-In-Time (JIT) compilation, which allows efficient IR submission and compilation at run time, but ahead-of-time (AOT) compilation and interpretation are also possible. For instance, the RPython client discussed in the previous paragraph performs AOT compilation. Mu clients can also use the Mu API to inspect and modify the state of the Mu instance, including contents of memory and threads/stacks, at run time. In addition, a Mu instance may trap to the client for handling of events that it cannot directly manage.

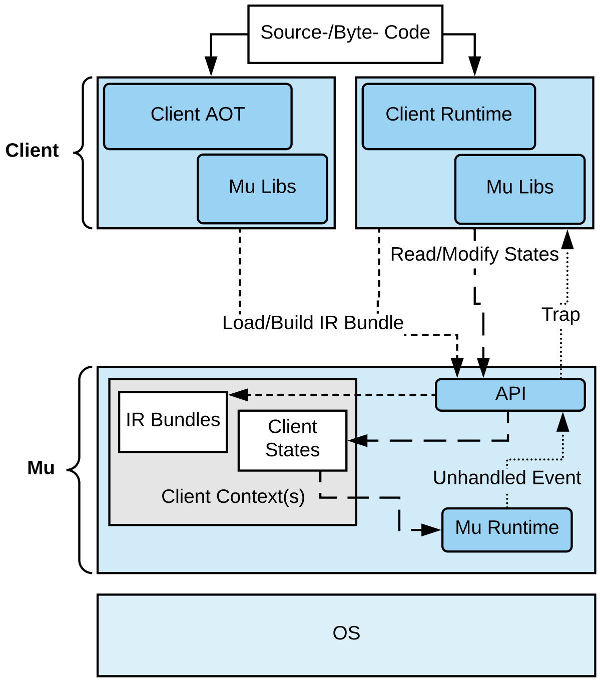


Figure 2.2: A Mu client, e.g. a managed language, builds Mu IR bundles and loads them to Mu to be executed. This can either happen ahead of time or at run time. It can also read and modify the internal state of the VM. In case of any non-trivial events, the Mu runtime traps to the client to handle it. All of the communication between Mu and the client is done through the Mu API.

Table 2.1: The Mu type system is simple and low-level, and consists of primitive-numerical, aggregate, reference and miscellaneous data types. These low-level types combine to support the implementation of a managed language type system. For instance, the `int` type in Python can be implemented using Mu's `int<n>`, `struct<T1 T2 ...>` and `ref<T>` types.

Type	Description
`int<n>`	n-bit fixed-size integer
`float`	IEEE754 32 bit floating point
`double`	IEEE754 64 bit floating point
`uptr<T>`	Untraced pointer to a memory location
`ufuncptr<sig>`	Untraced pointer to a native function
`struct<T1 T2 ...>`	Structure with fields `T1 T2 ...`
`hybrid<F1 F2 ... V>`	A hybrid with fixed and variable parts
`array< T n>`	Fixed-size array of same type elements
`vector<T n>`	Vector of same type elements
`ref<T>`	Reference to a heap object
`iref<T>`	Internal reference to a memory location
`weakref<T>`	Weak reference to a heap object
`funcref<sig>`	Reference to a Mu function
`stackref`	Opaque reference to a Mu stack
`threadref`	Opaque reference to a Mu thread
`framecursorref`	Opaque reference to a Mu frame cursor
`irbuilderref`	Opaque reference to a Mu IR builder
`tagref64`	64 bit tagged reference
`void`	Void type

A.2.8- added more on types to table 2.1

Type System Mu provides a simple low-level type system with reference types to support precise garbage collection. The types supported by Mu appear in Table 2.1, are categorized into four groups: (1) Primitive numerical types including `int`, `float`, `double`, `uptr` and `ufuncptr`, which are not traced by the GC, (2) Composite types including `struct`, `hybrid`, `array` and `vector`, which consist of smaller components, and are used to define new data types, (3) Reference types including `ref`, `iref`, `weakref`, `threadref`, `stackref`, `framecursorref` and `irbuilderref`, which are traced by the GC and can only be created as a result of specific Mu instructions, and (4) Miscellaneous types including `tagref64` and `void`.

Although the current Mu type system is capable of implementing a managed language type system, it lacks some of the basic data types required by the additional features of RTPLs. These data types are explained in Section 3.3.2.

Concurrency A thread is the unit of concurrency in Mu. To run, each thread must be bound to a stack as its execution context. It is also possible to unbind a thread from its stack and rebind it to another stack. This is the `SwapStack` operation [Dolan et al., 2013]. `SwapStack` enables implementation of language features such as co-routines and language-defined user-level thread scheduling. Mu threads are typically implemented as native threads scheduled by the operating system, and run concurrently. However, for the sake of flexibility, the Mu specification does not force how Mu threads are bound to the operating system threads. It is up to the Mu implementation to choose a proper thread mapping.

A.2.9- added Mu's thread mapping

To implement diverse synchronization primitives, Mu provides a basic tool that is similar to the Linux futex and can easily be mapped to it. The client uses the Mu futex and Mu atomic operations to implement higher-level primitives such as mutexes, condition variables, and message queues. In addition, Mu has a well-defined C11-like memory model. It is the client's responsibility to use the provided memory orderings to synchronize multi-threaded programs.

A.3.7- added message queue

Most real-time systems require more control over the behaviour of multi-threaded programs, beyond what Mu provides. We discuss the necessary additional concurrency abstractions in Section 3.3.5.

Memory Management Mu's memory consists of a garbage-collected heap, a global memory area, and the stack area. The garbage collector automatically handles allocation and reclamation of fixed or variable sized objects on the heap. Global memory is allocated statically, and lives throughout the program's lifetime. The stack area accommodates stacks which may be created or deleted manually.

Garbage collection is an integral part of Mu's memory management. However, it is too complex for some real-time systems, and imposes intolerable pauses or overhead for many of them. In Section 3.3.4, we discuss an alternative approach that is suitable for a wide range of real-time applications.

Compiler Backend The compiler backend of Mu translates Mu IR to machine code. Following the minimality principle, and because of the importance of language-specific optimizations [Castaños et al., 2012], Mu expects most optimizations to be performed by the client. While some optimizations (e.g., register allocation) must be done close to the machine, language-neutral optimizations are often much less important than language-specific ones, and those must be performed within the higher layers of the client, where enough knowledge about client language semantics is available.

For a real-time system, a compiler backend aiming at optimizing worst-case execution time (WCET) rather than average-case performance may seem desirable. However, not all real-time systems can afford the level of complexity in such a backend. Also, µVM design principles argue against adding new features, in favor of preserving minimality. We discuss our recap of these challenges and the current RTMu backend design in Section 3.3.3.

Unsafe Native Interface The Mu Unsafe Native Interface (UNI) provides support for direct interaction between Mu IR and native programs such as OS system calls. This is usually necessary for managed languages. For instance, C# provides support for directly calling C functions. The JVM on the other hand, prohibits direct interaction. Thus, Java applications have to use JNI and endure a heavyweight C-Java boundary overhead.

Real-time systems often require a high level of control over their outside world, including the other pieces of software on the system and the physical environment. Hence, the UNI is necessary in developing RTPLs, mainly because it helps to implement a means of interaction with system software, such as the RTOS kernel and device drivers. Using this interface, the RTPL primitives are able to control other software elements and I/O devices.

Implementation and Evaluation Currently, there are two open-source implementations of the Mu specification: a reference implementation which acts as a proof of implementation [Wang, 2018], and a high-performance implementation [Lin, 2019], which is still under development. To demonstrate the capability of Mu in supporting a real-world language as a client, some experiments are reported by Wang [2018]. A portion of the PyPy project's RPython framework has been retargeted from C to Mu, which was able to execute the RPySOM interpreter and the core of the PyPy interpreter on a Mu implementation. Some early work on supporting GHC has also been done.

2.5 Summary

RTMu is designed based on the Mu specification and inspired by real-time programming languages (RTPL) and real-time operating systems (RTOS). In this chapter, we presented the background material for this thesis. We introduced our target domain, real-time systems, and discussed some of their key differences to non-real-time systems. Next, we briefly explained the main challenges in implementing programming languages, including the ones for real-time systems. Then, to review the language features specific to real-time systems, we surveyed a representative set of RTPLs that cover a diverse range of real-time systems. We also introduced Mu and explained the key aspects which require further attention from the viewpoint of a RTPL developer. The next chapter presents the design of RTMu and discusses how the information in this chapter influenced the design.

Chapter 3

RTMu: A Micro Virtual Machine for Real-Time Systems

A.3.2-change-1:emphesize specification

A.3.1-change 2: added footnote

In this chapter, we present the design of RTMu, *a concrete specification* for the first micro virtual machine (µVM) on which managed programming languages for real-time systems can be developed[1]. The goal of our design is to make the advantages of correct managed languages far more accessible in the real-time domain. We build on a previously published µVM specification, named Mu, and propose a set of modifications to its abstractions over concurrency and memory management to make it suitable for real-time systems.

The first two sections of this chapter (3.1 and 3.2) address the high level aspects of the design. Sections 3.3.1 and 3.3.3 present the parts which mostly reuse the Mu abstractions, with slight changes where necessary. Sections 3.3.4, 3.3.5 and 3.3.6 explain our new abstractions over memory management, concurrency and time. Finally, we summarize the design in Section 3.4.

This chapter is based-on work originally published at VMIL-2019 [Amiri et al., 2019], and subsequently refined as a result of the experience implementing the specification. The contributions of this chapter are: (1) identification of core language features that distinguish real-time languages, (2) design of the first µVM for real–time language implementation, (3) specification of IR extensions to support real-time languages, (4) specification of runtime changes to support real-time languages, (5) description of how to use the new primitives to conform to the requirements of a variety of real-time applications.

3.1 Key RTPL Features

Considering the Real-Time Programming Languages (RTPL) in Section 2.3 and many more, RTPLs can be divided into three main categories, based on the range of real-time systems they target: (1) high-integrity systems, (2) resource-constrained systems, and (3) other systems, often as part of a large-scale real-time system.

[1] Existing real-time Java VMs such as FijiVM [Pizlo et al., 2009] and JamaicaVM [Siebert, 2010] are all macro VMs.

In *high-integrity* systems, such as the safety-critical software systems in a car or an airplane, reliability is the most important requirement. From a language design perspective, reliability should be investigated at two levels. First, it should be possible to verify various aspects of an application written in the language. To achieve this, languages such as SPARK relinquish hard-to-analyze language features such as pointer types and dynamic object allocation, in favor of analyzability. Second, it should be possible to verify the language and its toolchain, which again means that simplicity is essential for the language design and implementation.

Languages for high-integrity systems may trade ease of programming, performance and efficiency for higher levels of verifiability. For instance, Schoeberl et al. [2017] report that the three level programming model of SCJ is not easy to learn. They also mention that the performance overhead of SCJ may make tasks with small periods infeasible. Additionally, writing an application as a set of PEHs with only static memory allocation is not an efficient way of using processing and memory resources.

In real-time systems with *resource constraints*—such as limited processing power, small memory, or a limited energy source—the language footprint is the most critical issue. So, languages may overlook ease of use and verifiability to consume resources more efficiently. In such systems, the C programming language is often used, since its runtime has a minimal computational and spatial footprint. In less constrained cases, managed languages such as SCJ may also be used.

Large-scale systems often consist of a number of real-time tasks with various requirements. Languages such as RTSJ, which target these systems, should supply a wealth of features including:

a. Static-priority scheduling algorithms, including the priority-based scheduling available in most RTOSs, and the Rate-Monotonic (RM) algorithm, that are the most popular scheduling choices in real-time systems, due to their availability and simplicity.

b. Dynamic-priority scheduling algorithms, such as the Earliest Deadline First (EDF), that are known to have better schedulable utilizations in some real-time systems.

c. Avoiding the unbounded priority inversion problem by supporting the Priority Inheritance Protocol (PIP), or avoiding both the unbounded priority inversion and the deadlock problems by supporting variants of the Priority Ceiling Protocol (PCP).

d. Automatic memory management through real-time GC, often accompanied by other easier-to-analyze and more predictable memory management techniques, such as scoped memory.

e. Access to physical memory addresses, to work with I/O devices, or on platforms with multiple types of memory.

f. Tools to monitor and manage time.

These features reduce the cost and difficulty of implementing and maintaining large-scale real-time applications, but they increase the language size and hamper verification. As a result, such languages are not often used in high-integrity or resource-constrained systems, despite being easier to program and less error-prone.

3.2 Scope

RTMu is a concrete µVM specification designed to facilitate development of correct programming languages for real-time systems, including implementing new real-time languages or reimplementing existing ones. RTMu is language-neutral, and addressing language-level concerns such as adherence to specific safety standards is outside its scope. In this section, we specify the range of real-time programming languages covered by this design.

A.3.8- emphasize SCJ levels out of scope

Requirements and Constraints To satisfy the requirements of real-time systems, RTPLs provide various features which are often not available in general-purpose languages. RTMu should supply the necessary low-level abstractions to implement these features.

In summary, RTMu must provide the following additional features:

- Concurrency:
  - Threads whose attributes indicate their timing and resource access constraints.
  - Control by the client over thread execution contexts and their mutual effects by setting thread attributes.
  - A scheduler that accommodates the client in building static and dynamic priority schedulers common in real time programming languages, including SCJ, RTSJ, RT-POSIX and SPARK Ada.
  - Inter-thread communication primitives that support prevention of unlimited priority inversion and deadlock.
- Memory management:
  - A choice of automatically managed (garbage-collected), semi-automatic, and manual memory management on the same real-time system.
  - A garbage collector that does not affect threads that do not access the heap.
  - Control over allocation and access for specific physical memory addresses.
- Other:
  - Basic tools for time measurement and time-triggered events.

Considering the already-demonstrated capability of Mu in implementing real-world languages, we will argue that the above features will cover the additional requirements of real-time systems and make RTMu capable of supporting the implementation of real-world RTPLs such as RTSJ.

Configurability Some of the features required by RTSJ, such as dynamic memory allocation and dynamic-priority scheduling, are not necessary when implementing languages such as SCJ and SPARK. Thus, RTMu is designed so that unused features will not burden such RTPLs. For instance, it is possible for a client language to completely ignore GC, or even to only use static memory allocation, and to do so without penalty.

Assumptions Following Mu, we assume the high-level language implementation (the client) is trusted, and that it will emit well-defined code for the RTMu runtime to execute. RTMu implementations are then permitted to omit dynamic safety checks, such as those for array bounds violations or null pointer dereferences, avoiding the related overheads or avoiding duplicating them in cases where the high-level language performs the check.

Non-Goals In this chapter, we are presenting the *design* of a reliable foundation to facilitate the emergence of new high-quality real-time managed languages to increase the breadth of the RTPL ecosystem. To narrow the scope of this design, we declare some non-goals. First, preserving minimality, this design drops support for popular general-purpose language features like JIT compilation, interpretation and dynamic class loading, which can introduce huge delays or make timing and correctness analysis of real-time applications very hard. Second, we do not address WCET analysis in this work: we do not require or prescribe a timing model for the underlying hardware, nor the IR instructions themselves. Third, we do not address concerns such as disabling unbounded loops and recursion, and calculating application memory requirements, because these can effectively be handled at language or application level, and it is against the *minimality* design principle of µVM, which is mentioned in Section 2.4, to put such features inside RTMu. These are interesting objectives but lie beyond the scope of this work.

A.3.2 and A.3.4 more clear non-goals

3.3 Design

Like Mu, RTMu provides low-level abstractions over concurrency, the compiler backend and memory management, and is designed to be minimal and formally verifiable. RTMu aims to support a wide range of real-time systems, including high-integrity systems. In the rest of this chapter, we explain RTMu's design, and when applicable, we will argue that our design decisions are minimal and sufficient for RTMu's purpose.

3.3.1 Architecture

RTMu-based managed language implementations follow the same high-level architecture as Mu, as depicted in Figure 2.2. The major difference is that RTMu only supports AOT compilation which means there will be no IR load/build through

Table 3.1: RTMu adds new data types, required for its new real-time features.

Type	Description
`timerref`	References to timers
`regionref`	References to explicitly managed memory regions
`attrref`	References to thread attributes
`futexref`	References to futexes
`condvarref`	References to condition variables

the API at run time. Also, the RTMu runtime manages two new memory areas in addition to the heap, immortal memory, and stacks: regions and explicitly managed memory (EMM).

3.3.2 Type System

The RTMu type system reuses the whole Mu type system and adds five types to support newly added features for real-time systems. The added types are shown in Table 3.1.

To support the basic time management primitives of RTMu, we add the `timerref` type to identify timers. Time values in RTMu are 64 bit integers, representing nanoseconds. These two types and their usage are explained in Section 3.3.6.

RTMu provides memory regions which can be created and destroyed dynamically. The client may build an arbitrary number of memory regions. To identify these regions, we add the `regionref` type.

To create a RTMu thread, the client needs to initialize its attributes. Inspired by the `struct pthread_attr_t` type from POSIX threads, and to simplify the relevant instructions' arguments, we add the `attrref` type to refer to an object that encapsulates all of these attributes. An instance of this type can only be interpreted and modified using the provided RTMu instructions.

RTMu adds a new `futexref` type, instead of reusing the 32 bit futex word from Mu, which was designed to be easily mapped to the Linux futex. The new type makes it easier to implement the RTMu futex on top of the wide range of platforms used in real-time systems. Finally, we add the `condvarref` type to identify RTMu condition variables.

Rationale

RTMu's new types cover all of its new operations. Given the already demonstrated capability of Mu's type-system in supporting modern managed languages, and the fact that RTMu does not exclude any of the Mu types, we argue that RTMu's type-system is *sufficient* to implement new real-time managed languages or reimplement existing ones, as long as RTMu's abstractions over concurrency, memory and compiler are also sufficient.

RTMu reuses existing Mu types in its new operations as much as possible, and introduces new types only where essential. For instance, RTMu reuses the Mu `int<n>` type to represent time value, priority, deadline and processor id. The new RTMu types in Table 3.1 (except `futexref`) are used to identify RTMu-specific entities which do not fit any existing type. Also, the addition of `futexref` is crucial for RTMu's portability. Hence, we argue that RTMu's type-system is *minimal* because excluding any type will lead to loss of expressiveness.

3.3.3 Compiler Backend

The RTMu system is responsible for executing the IR code given to it by the client. Since we do not support the Mu system's API for dynamic addition of code to an already running system, an RTMu implementation is permitted to support only ahead-of-time compilation of such IR to machine code. (Indeed, our RTMu design does not in principle prevent an implementation from *interpreting* IR directly.)

As mentioned earlier, supporting or performing any form of WCET-analysis is an explicit non-goal of this design, so we expect the execution engines of existing Mu implementations could be used in RTMu implementations. As the following sections explain, our changes to the RTMu IR are at the level of adding new entry-points for controlling the runtime system, rather than changing "computational" facilities.

3.3.4 Memory Management

Memory management in RTMu provides the following basic types of memory areas which can be used by the client to construct more sophisticated memory managers:

- stacks,
- garbage collected heap,
- explicitly managed memory area,
- regions, and
- immortal memory area.

A summary of these memory areas and the operations a client can perform on them is shown in Figure 3.1. Each of these areas is explained separately in the following paragraphs. Among these memory areas, the explicitly managed memory (EMM) and regions are not already available in Mu. We borrow the other areas from Mu and modify them to suit the requirements of RTMu.

A detailed list of new memory operations of RTMu is depicted in Table 3.2. All of these instruction act on the new EMM and regions.

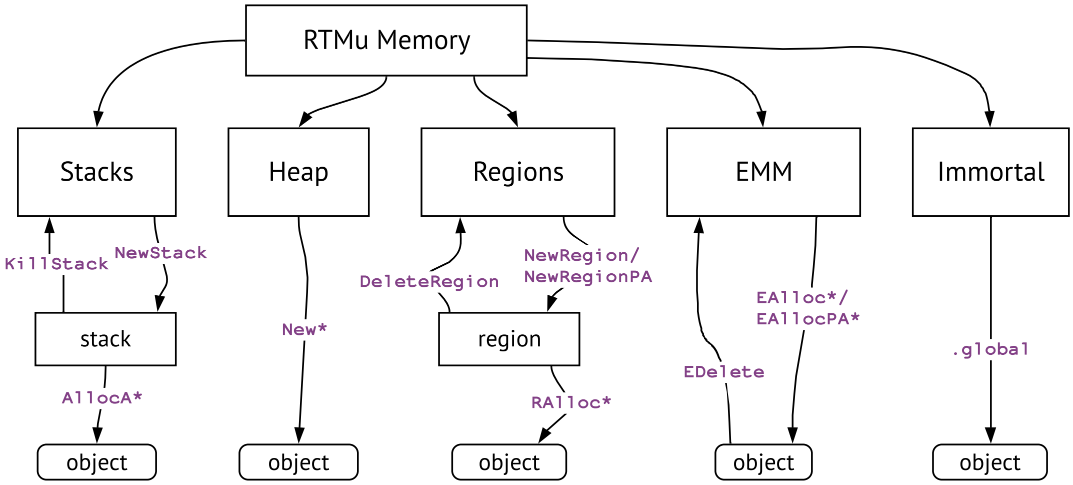


Figure 3.1: RTMu memory is divided into five areas, each serving a range of higher level memory managers. Among them, stacks, the garbage collected heap and immortal (static or global) areas are common in non-real-time managed languages. For real-time languages, we add EMM and Regions which are highly flexible and may be used to implement a range of manual and semi-automatic memory managers. In the figure, an object is a typed fixed-size entity, while a region is a fixed-size container for objects. (Instructions marked by a star have hybrid versions which allocate objects with variable-length (hybrid) types.)

Table 3.2: Instructions added by RTMu. RTMu adds several instructions to support implementation of the common memory managers in RTPLs and more. The first seven instructions serve region-based memory, like RTSJ scoped memory. The next five instructions mainly target manual dynamic memory, like `malloc` and `free` in C. The last instruction checks whether a reference is located in a region and returns a `regionref` or NULL.

Operation	**Description**
`regionref NewRegion (int<64> size)`	Allocate a new region with `size` number of bytes.
`regionref NewRegionPA (int<64> size, uptr<void> addr)`	Allocate a new region with `size` number of bytes at the specified physical memory address.
`void CollectRegion (regionref regref)`	Delete all objects allocated in `regref`.
`void DeleteRegion (regionref regref)`	Delete the region pointed by `regref` and all objects it contains.
`void BindRegion (regionref regref)`	Disable swap-out to disk for the region pointed by `regref`.
`void UnBindRegion (regionref regref)`	Re-enable swap-out to disk for the region pointed by `regref`.
`ref<T> RAlloc/RAllocHybrid (regionref regref, T)`	Allocate a fixed/variable-size object of type `T` on the region `regref`.
`ref<T> EAlloc/EAllocHybrid (T)`	Allocate a fixed/variable-size object of type `T` on the EMM space.
`ref<T> EAllocPA/EAllocHybridPA (T, uptr<void> addr)`	Allocate a fixed/variable-size object of type `T` at the specified physical memory address.
`void EDelete (ref<T> obj)`	Delete the object pointed by `obj` from the EMM space.
`void BindObject (ref<T> obj)`	Disable swap-out to disk for the EMM object pointed by `obj`.
`void UnBindObject (ref<T> ptr)`	Re-enable swap-out to disk for the EMM object pointed by `ref`.
`regionref RefToReg (ref<T> obj)`	Return the region where `obj` is located, or `NULL` if not in Regions area.

B.3.1-changed client to RTMu implementation

Garbage-Collected Heap The RTMu's garbage collected heap is an automatically managed memory area like the garbage collected heap in RTSJ. The RTMu specification does not stipulate any particular GC algorithm. Instead, RTMu implementations should choose the right GC for their purpose and account for its effects on tasks that do not use the heap. Although this is not straightforward to achieve, it has already been shown to be possible in a wide range of real-time systems [Sharp et al., 2003]. Additionally, RTMu enables the client to build a language with no GC support. In this case, the language is not affected by any of the complexities of GC.

Stacks A stack in RTMu is the context of activations of functions. Allocating on a stack is done using `AllocA T` or `AllocAHybrid T` instructions, which return a reference of type `iref<T>`. The returned reference is traced by the GC because the object it points to may contain direct or indirect references to heap objects. In RTMu, the client may create tasks which do not use the heap to avoid complexities and delays. Such tasks should not experience any interference from the GC (e.g. through scanning the task's stack).

Explicitly Managed Memory The EMM area of RTMu provides low-level primitives to implement memory managers, which support both allocation and deletion, at the granularity of an object. For instance, the `malloc` and `free` functions in C can be built using this memory area.

The EMM area allows allocation of typed fixed-sized objects. The client may use the two variants of the `EAlloc` instruction to directly allocate objects in the EMM. These objects may be deleted using the `EDelete` instruction.

By default, the `EAlloc` instruction allocates in RTMu's virtual address space, and the underlying OS handles the virtual to physical address mapping. By default, nothing prevents the OS memory manager from swapping the memory page(s) holding the object out to secondary storage such as the hard-disk. Some real-time applications need to avoid the unpredictability and overhead of the page swapping in RTOSs like RT-Linux. So, RTMu adds the `BindObject` instruction to bind an object to the main memory. The client may later use the `UnBindObject` instruction to allow swapping to happen again.

Real-time applications may also require direct access to specific addresses in physical memory, for instance to do memory-mapped I/O, or to handle platforms with more than one type of memory. To enable this, RTMu adds the `EAllocPA` instruction to allocate an object at the client-specified address.

Availability and behavior of memory management instructions such as `BindObject` is tightly dependent on the underlying RTOS memory manager. If an RTOS does not provide virtual memory management or swapping, the `BindObject` instruction will have no effect. Also, if an RTOS restricts the access to physical memory or specific physical addresses, the affected RTMu instruction will not be available.

```
class SomeList {
    SomeList next;
}

outer_scope.enter();
SomeList head = new SomeList();
...
ScopedMemory inner_scope = new ScopedMemory(const_size);
inner_scope.enter();
SomeList tail = new SomeList();
head.next = tail;   // exception
```

Listing 3.1: Simplified RTSJ code that tries to create a reference from an object (`head`) allocated in an outer (older) scope to an object (`tail`) allocated in an inner (younger) scope. RTSJ disallows such references, requiring the RTSJ VM to detect and trigger an exception at the point of the assignment.

Regions The *regions* area is a part of the RTMu memory space in which the client may allocate and delete fixed-size contiguous pieces of memory, each called a region. Creating a new region in the *regions* area is done by calling one of the two variants of the `NewRegion` instruction. The created region may be deleted by calling the `DeleteRegion` instruction. It is also possible to delete all objects contained in a region using the `CollectRegion` intrinsic. In this case, the memory used by the region is not reclaimed. To allocate an object inside a region, RTMu provides the two variants of the `RAlloc` instruction. The allocated objects are deleted only when the client deletes or collects the containing region. The RTMu regions can be used to implement memory managers such as the variants of scoped memory in RTSJ, mission and private memories in SCJ and the unbounded containers in SPARK.

Deleting a region may lead to dangling references to objects inside the region. RTMu provides the `RefToReg` intrinsic, which takes a reference to an object and returns a reference to its containing region, or `NULL` if the conversion is not valid (e.g. the input is a heap object). The client can use this instrinsic to check for dangling references, allowing the implementation of semantics like those for RTSJ's scoped memory. Listing 3.1 shows a scenario in RTSJ where an exception is thrown to indicate a potential dangling reference. A translation of this scenario to RTMu IR is depicted in Listing 3.2.

As explained for EMM objects, the client may need to bind and unbind regions to main memory. For this, RTMu adds the `BindRegion` and `UnBindRegion` instructions. It is also possible to allocate a region at a specific physical memory address through the `NewRegionPA` instruction.

Similar to the stack, the client may allocate objects with or without references to the heap, in a region. An RTMu implementation is responsible for guaranteeing that a task using regions with no references to the heap is not affected by the GC. Additionally, the GC must have access to all the required data to collect heap garbage correctly and leave no dangling references.

```
regionref _inner_scope = NewRegion (const_size)
// client-written function to keep scopes and update current_scope
Call push_scope (_inner_scope)
iref<ref<SomeList_t>> _head_next = GetFieldIRef <_headref, 0>
ref<SomeList_t> _tail = RAlloc (current_scope, SomeList_t)
// check lifetimes
regionref _dest = RefToReg (_head_next)
regionref _src = RefToReg (_tail)
// client-written function to quantify scope lifetimes
age_t _dest_age = Call scope_age (_dest)
age_t _src_age = Call scope_age (_src)
// allowed if src will live equal or longer
int<1> is_allowed = cmpOp::UGE _src_age _dest_age
Branch2 is_allowed store_block exc_block
exc_block:
    Throw some_exception
store_block:
    Store <<ref<SomeList>> _head_next, _tail
```

Listing 3.2: A translation of the last four lines of the RTSJ code in Listing 3.1 into (compact) RTMu IR code. This shows how a client can use the low-level memory management instructions of RTMu to provide a higher-level memory management feature required by a real-time language, namely preventing dangling references in scoped regions.

Immortal Memory The immortal memory area is the preferred tool to implement memory managers which do not need to delete objects. It is more efficient than other memory areas, as it keeps less information and is simpler. The immortal memory area can be used to implement the global memory in C, the immortal memory in RTSJ and SCJ, and the bounded containers in SPARK.

Configurability

Client languages such as SCJ and SPARK should be able to assume that they are not burdened by the complexities of GC. Hence, RTMu implementations must guarantee that a client without `New/NewHybrid` instructions is not affected by the GC (e.g., GC-related read/write barrier code must not be emitted). As RTMu only supports AOT compilation, this can be done by simple static analysis at compile-time.

A client language may want to support safety-critical and non-safety-critical real-time tasks at the same time. In such a language, there may be three types of tasks: (1) low-priority tasks that use `New/NewHybrid` instructions, (2) medium-priority tasks that allocate on alternative areas (e.g. scopes) but refer to heap objects, and (3) safety–critical tasks that do not refer to heap objects, and assume that they are not burdened by the complexities of GC. To support such a language, RTMu implementations are responsible for performing the required analysis based-on their choice of GC.

Rationale

RTMu's memory instructions are designed to be *sufficient* for implementing the memory managers in the real-time languages we surveyed. In this section, we mentioned some of the popular ones including the garbage-collected heap and scoped memory in RTSJ, `malloc` and `free` in C, and static memory allocation in SPARK and SCJ. We demonstrate this capability by designing and implementing a comprehensive memory manager for RT-RPython, our new real-time language, explained in Section 5.2.1.

B.3.2-changed language to less bold

The memory instructions of RTMu are *minimal* for two reasons: First, if we remove any of these instructions, RTMu will lose its ability to implement at least one of the memory areas in our list of surveyed real-time programming languages. Second, all of these instructions represent low-level operations with no overlapping functionalities. The only exception is the `CollectRegion` instruction which might be replaced with deleting the target region and creating a new one. However, this can be inefficient compared to a single `CollectRegion` instruction.

3.3.5 Concurrency

RTMu makes several necessary amendments to the concurrency primitives of Mu for real-time systems. It grants more control over thread attributes and scheduling parameters, so that the client can implement various scheduling algorithms. Also, some new features, essential in real-time applications, are added to Mu's basic synchronization primitives.

RTMu provides the client with a basic two-step scheduler which enables the implementation of a wide range of static and dynamic-priority scheduling algorithms. The RTMu scheduler consists of a certain number of priority levels. Threads may be added to each priority level or removed from it at run time. RTMu implementations are allowed to put an upper-bound on the number of threads at each priority level. In the common case of running on top of a RTOS such as RT-Linux or RTEMS, both the maximum number of priority levels and the number of tasks at each level depend on the limitations of the underlying RTOS scheduler. [2] In the first step, the scheduler finds the highest priority level with at least one ready-to-run task. If there is more than one thread at that priority level, the second scheduler step selects a thread to run based on the chosen scheduling policy. The choice of scheduling policies includes Round Robbin (RR), First In First Out (FIFO) and Earliest Deadline First (EDF). The first two policies can be used to implement fixed priority scheduling, the most popular scheduling approach in real-time systems [Burns and Wellings, 2009]. The last one is a foundation for the development of dynamic priority scheduling algorithms, including EDF itself. The schematic structure of the scheduler and how it imposes scheduling policies is depicted in Figure 3.2.

A.3.6-change-1: added dynamic and restricted

A.3.6-change-2: added footnote

To enable the scheduler, RTMu adds three attributes to threads. (1) The `priority` attribute is a number between zero (the highest priority) and the number of sched-

[2] It is possible for an RTMu implementation to implement its own scheduler without these limitations. That will introduce *implementation* challenges which are outside the scope of this chapter which is on the RTMu *specification*.

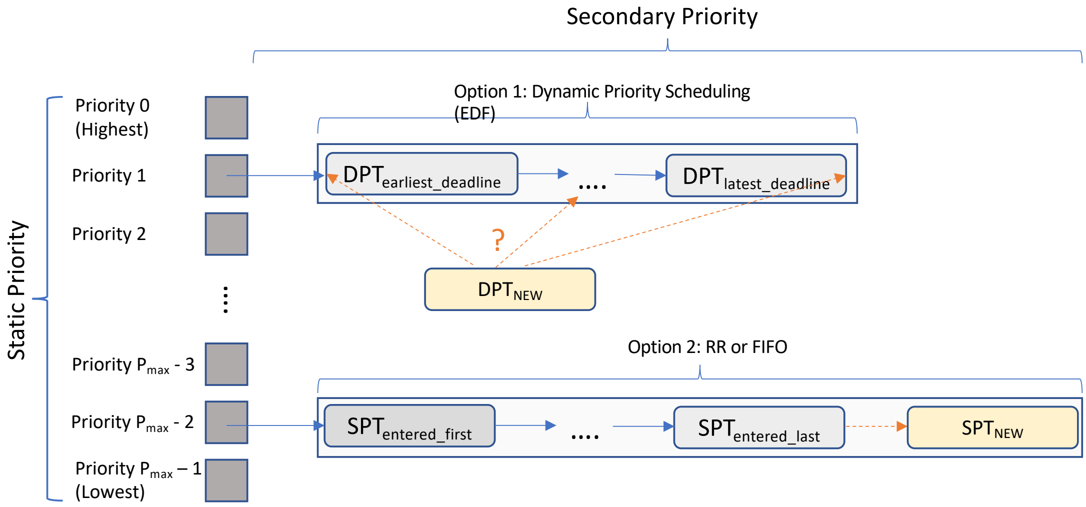

Secondary Priority
Option 1: Dynamic Priority Scheduling (EDF)
Static Priority
Priority 0 (Highest)
Priority 1
Priority 2
Priority P_{max} - 3
Priority P_{max} - 2
Priority P_{max} – 1 (Lowest)
$\text{DPT}_{\text{earliest\_deadline}}$
....
$\text{DPT}_{\text{latest\_deadline}}$
?
DPT_{NEW}
Option 2: RR or FIFO
$\text{SPT}_{\text{entered\_first}}$
....
$\text{SPT}_{\text{entered\_last}}$
SPT_{NEW}

Figure 3.2: The RTMu Scheduler consists of a number of (static) priority levels. For each level, the scheduler keeps a queue of ready tasks. The position of a new ready task in the queue depends on the scheduling policy at that priority level. For RR and FIFO, the new task is always the last in the queue, and for EDF, tasks with smaller deadlines are inserted closer to the queue head.

uler's priority levels minus one (the lowest priority). Tasks at higher priority levels are dispatched earlier. (2) The `affinity` attribute is an opaque data type indicating the processing nodes on which the current threads may run. The structure of the data is platform-dependant and can only be modified using the relevant RTMu instructions. (3) The `deadline` is an integer (time value), used to decide the dispatch sequence of threads at a priority level with EDF scheduling policy. The thread with the smallest `deadline` is dispatched first.

RTMu thread attributes are encapsulated in a new type named `attrref`. When a client creates a new real-time thread, they must pass the initial attributes of the thread as an object of this type. The internal structure of the `attrref` type is platform-dependent. Modifying objects of this type is only possible through the provided RTMu instructions.

To implement synchronization primitives, RTMu provides three basic tools including futex, condition variables and atomic operations. The RTMu futex is a fast mutual exclusion lock which supports the PIP or PCP. Similar to the RT-Linux futex, the contention-free case in RTMu futex can be handled at the user-level. The RTMu condition variable is a priority-aware synchronization primitive that allows multiple threads to wait for a condition. RTMu's atomic operations are borrowed from Mu without any change. As with Mu, RTMu clients are responsible for implementing higher-level language-specific synchronization mechanisms, such as protected objects, message queues or monitors using futex, condition variable and atomic operations. All of RTMu's concurrency related instructions are shown in Tables 3.3, 3.4 and 3.5.

A.3.7- added message queue

RTMu is designed to conform to RTOSs' concurrency primitives, to allow lightweight implementation of the above-mentioned properties. If the RTOS does not

Table 3.3: RTMu provides instructions to create, initialize and manage real-time threads and their attributes.

Instruction	Description
`attrref NewAttr`	Create a new thread attributes object
`void DeleteAttr (attrref a)`	Delete a thread attributes object and reclaims its memory

`void AttrSetPriority`
`(attrref a, int<64> p)`
	Set the priority field of a thread attributes object

`int<64> AttrGetPriority`
`(attrref a)`
	Get the priority field of a thread attributes object

`void AttrSetDeadline`
`(attrref a, int<64> dl)`
	Set the deadline field of a thread attributes object

`int<64> AttrGetDeadline`
`(attrref a)`
	Get the deadline field of a thread attributes object

`void AttrSetCPU`
`(attrref a, int<64> n)`
	Add processor node *n* to the active processors of a thread attributes object

`void AttrClearCPU`
`(attrref a, int<64> n)`
	Remove processor node *n* from the active processors of a thread attributes object

`void AttrZeroCPU`
`(attrref a)`
	Remove all processor nodes from the active processors of a thread attributes object

`bool AttrIsSetCPU`
`(attrref a, int<64> n)`
	Check whether processor node *n* is in the active processors of a thread attributes object

`threadref NewRTThread`
`(attrref a, stackref s,`
`threadLocalClause,`
`newStackClause, excClause)`
	Create a new thread with the specified arguments

`void ThreadSetAttr`
`(threadref t, attrref a)`
	Update the current attributes of a thread to a

`attrref ThreadGetAttr(threadref t)`	Return the current attributes of a thread

`void ThreadSetPriority`
`(threadref t, int<64> p)`
	Set the priority of a thread

`int<64> ThreadGetPriority`
`(threadref t)`
	Get the current priority of a thread

`void ThreadSetDeadline`
`(threadref t, int<64> dl)`
	Set the deadline of a thread

`int<64> ThreadGetDeadline`
`(threadref t)`
	Get the current deadline of a thread

`void ThreadSetCPU`
`(threadref t, int<64> n)`
	Add processor node *n* to the active processors of a thread

`void ThreadClearCPU`
`(threadref t, int<64> n)`
	Remove processor node *n* from the active processors of a thread

`bool ThreadIsSetCPU`
`(threadref t, int<64> n)`
	Check whether processor node *n* is in the active processors of a thread

Table 3.4: RTMu provides futex as a mutual exclusion lock that supports PIP or PCP. PIP is the default protocol. To switch to PCP, the priority ceiling must be set to a value other than the lowest RTMu priority (platform-dependant). Also, resetting to the lowest RTMu priority will switch back to PIP.

Instruction	Description
`futexref NewFutex`	Create a new futex
`void DeleteFutex (futexref ftx)`	Delete a futex
`void LockFutex (futexref ftx)`	Lock a futex, blocks if the futex is already locked
`void TimedLockFutex (futexref ftx, int<64> dur)`	Lock a futex, blocks for at most `dur` nano-seconds
`void UnlockFutex (futexref ftx)`	Unlock a futex
`void SetPCFutex (futexref ftx, int<64> p)`	Set the priority ceiling of a futex
`int<64> GetPCFutex (futexref ftx)`	Get the current priority ceiling of a futex

Table 3.5: An RTMu condition variable is a synchronization primitive that allows multiple threads to wait for a condition. Similar to POSIX condition variables, each RTMu condition variable is associated with an RTMu futex which must be locked before waiting on the condition variable, and unlocked after returning from the wait operation. A signal operation unblocks the highest priority thread waiting on the condition variable.

Instruction	Description
`condvarref CondVarNew`	Create a new condition variable
`void CondVarDelete (condvarref cv)`	Delete a condition variable
`void CondVarWait (condvarref cv, futexref ftx)`	Wait on a condition variable
`void CondVarTimedWait (condvarref cv, futexref ftx, int<64> dur)`	Wait on a condition variable for at most `dur` nano-seconds
`void CondVarSignal (condvarref cv)`	Signal one of the threads waiting on a condition variable
`void CondVarBroadcast (condvarref cv)`	Signal all threads waiting on a condition variable

supply the needed foundation, RTMu may not be able to provide some of its features. For instance, RT-Linux and RTEMS support EDF only for threads at the highest level of priority. Thus, an RTMu implementation on RT-Linux may choose not to provide EDF at all priority levels if the complexity or overhead is too high.

Rationale

Scheduling multi-processor real-time systems consists of two problems [Davis and Burns, 2011b]: (1) *allocation* of tasks to processors, and (2) assigning *priority* to order the execution of jobs (e.g. one release) of tasks in the system. For the allocation problem, RTMu provides an API to control tasks' affinities throughout the system run time. For the priority assignment problem, RTMu provides APIs to assign static and dynamic priority to tasks, both of which may be changed throughout the system run time. Hence, RTMu provides *sufficient* abstractions to support the key scheduling algorithms in the multi-processor real-time systems' domain.

RTMu's abstractions over task scheduling do not dictate any higher-level scheduling algorithms such as RM and EDF (the notion of deadline in RTMu is only a number to represent dynamic priority in general, and is not limited to deadline as a time). Rather, they provide a *minimal* set of tools to implement these algorithms, and a wide range of alternatives.

3.3.6 Clock and Timers

Timeliness is a vital part of a real-time application's mission. Thus, any language for such systems should provide a toolset for managing time. There are two types of tools for this purpose [Burns and Wellings, 2009]: (1) tools to measure the passage of time (e.g., the `Calender` package in Ada and the `Clock` class in RTSJ), and (2) tools to schedule events at specific times (e.g., `delay` in Ada and `sleep` in RTSJ). To support the implementation of these two types of tools, RTMu provides low-level abstractions over clock and timers.

All time-related RTMu instructions are mentioned in Table 3.6. There is one instruction to read the current system *clock* value as a 64 bit integer representing the number of nano-seconds. The other four instructions create, monitor, modify and delete *timer*s which can be used to manage Time-Triggered (TT) events. Additionally, RTMu adds the `timerref` type to identify timers. As an example of implementing high-level language features on RTMu, the pseudo-code for creating a periodic task using a RTMu timer is shown in Listing 3.3.

Rationale

The time-related instructions of RTMu are designed to include a *minimal* set of the time-management facilities of RTOSs such as RT-Linux and RTEMS. We argue that these low-level instructions are *sufficient* for implementing higher-level language services, because they cover both types of tools mentioned in Section 3.3.6.

Table 3.6: The new clock and timer methods in RTMu include one basic operation to read the clock, and four basic operations to manage timers.

Method	Description
`int<64> GetTime ()`	Return the current system time
`timerref NewTimer ()`	Create a new timer and returns a handle

`void SetTimer`
`(timerref tmr, int<64> tm,`
`funcref fn, ref<void> arg)`
	Activate the timer `tmr` to call `fn(arg)` after an interval `tm`

`void CancelTimer (timerref tmr)`	Deactivate the timer `tmr`
`void DeleteTimer (timerref tmr)`	Delete the timer `tmr`

```
Fn parent():
    // the period is 1 millie-second
    int<64> _period = 1_000_000;  // nano-seconds
    stackref _stack = NewStack (entry);
    Call init_thread (_stack, _period);

Fn entry(ref<void> arg)
    BEGIN:
    Call wait (self.cond);
        // THREAD BODY GOES HERE
    Branch BEGIN

// initialize the periodic thread
Fn init_thread(stackref _stack, int<64> _period)
    attrref _attr = NewAttr ();
    threadref _thread = NewRTThread (_stack, _attr);
    timerref _timer = NewTimer (); // not started yet
    ref<void> arg = RefCast (_thread)
    // call wake_thread periodically
    SetTimer (_timer, _period, wake_thread, arg);

Fn wake_thread(ref<void> arg)
    threadref _thread = RefCast (arg)
    Call wake(_thread.cond);
```

Listing 3.3: Simplified RTMu IR code that creates a periodic thread. A `parent` function creates a new stack for the periodic thread's `entry` function. Then, the `init_thread` function initializes the thread and binds it to the stack. Next, the parent thread creates a timer to periodically call `wake_thread`. At each period, this function sends a signal to wake the periodic thread, which will run the thread body.

3.3.7 Unsafe Native Interface

A.3.5- added new subsection

As mentioned in Section 2.1.1, working with I/O devices is a critical part of many real-time systems, and any language targeting these systems needs to provide a proper support for it. To enable this, RTMu borrows the Unsigned Native Interface (UNI) of Mu, which allows direct interaction between RTMu IR and native programs such as OS system calls. Using UNI and RTMu's new instructions for working with physical memory addresses, mentioned in Section 3.3.4, RTPLs are able to implement features such as writing new Interrupt Service Rountines (ISR) and interacting with device drivers.

3.3.8 Client Interface

Compared to the client interface in a non-real-time µVM such as Mu, the RTMu API is more restricted. It mainly allows the client to build and load IR code bundles. However, this cannot happen at run time because RTMu only supports AOT compilation. The API may also provide tools such as `KeepAlive` clauses for debugging purposes, provided that it does not affect the run-time behaviour when not debugging.

Due to the removed support for JIT compilation, the RTMu API does not provide features such as accessing and manipulation of the states of µVM memory, threads and stacks. It also does not support run-time optimizations.

3.4 Summary

In this chapter, we presented the design of RTMu, a µVM on which managed programming languages for real-time systems can be developed. RTMu is designed based on an existing non-real-time µVM specification named Mu. To design RTMu, we first extracted a set of key features that distinguish real-time and non-real-time programming languages. Then, we proposed a minimal set of changes to Mu's abstractions over concurrency and memory management, plus a new abstraction over time management, to make it suitable for implementing real-time languages. We also justified the minimality of our proposed changes, and their sufficiency for implementing RTPLs.

In the next chapter, we present our implementation of the RTMu specification as the first step towards demonstrating its viability.

Chapter 4

RTMu Implementation

Our first step in demonstrating the viability of the RTMu specification is to provide an implementation of RTMu. We build on a high-performance implementation of the Mu specification in Rust [RustTeam], named *Zebu* [Lin, 2019], and extend it in compliance with RTMu. We call the new implementation *RTZebu*. The source code for RTZebu is available on ANU Gitlab [RTZebuGit, 2021].

B.7.1- cited gitlab link

A.4.1, A.4.4, A.4.6, A.4.7, A.6.4 and B.4.1- added one paragraph on other RTOSs

RTZebu is an implementation of the RTMu specification on RT-Linux. Although RT-Linux is not the RTOS of choice for all real-time systems, it is sufficient for the goal of this thesis which is to demonstrate the possibility of an efficient implementation of the RTMu specification. We acknowledge that there are other more popular RTOSs such as seL4 [Klein et al., 2009], RTEMS and FreeRTOS, which outperform RT-Linux in certain aspects such as lower worst-case delays, less memory footprint and higher reliability. Implementing RTMu on them means it will be usable on a wider range of real-time systems, specially the hard real-time ones. However, these RTOSs provide considerably less OS level services than RT-Linux and implementing RTMu on them demands a very substantial engineering effort unrelated to the core objectives of this thesis. Hence, it did not fit in the timeline of this thesis and we do not discuss the implementation of RTMu on these RTOSs in this chapter.

We start this chapter with an introduction to Rust, as our language of choice for high-performance VM implementation. Then, we briefly present *Zebu* as the basis for the *RTZebu* implementation. Finally, we mention the key aspects and challenges in the *RTZebu* implementation.

4.1 Rust

Rust is a systems programming language that aims to bring together the low-level control and efficiency of C and C++, and the safety features of higher-level languages. Rust's design has three major principles [Turon, 2015b,a]:

1. *memory-safety* without garbage collection,
2. concurrency without data races (*thread-safety*), and
3. abstraction without overhead.

Rust achieves all of these through its rules of object ownership, lifetimes and reference borrowing, which are verified by a powerful compile-time safety checker. This shifts as much as possible of the safety burden to compile time and eliminates the run-time overhead. All these benefits may sometimes come at the cost of Rust's expressiveness [Lin, 2019] and its longer than usual learning curve.

In addition to its safe world with no data races and no memory faults, Rust provides an unsafe world, where programmers may use unsafe code, such as raw pointers and external functions at their own risk, provided that they wrap it in an unsafe block or function. If used properly, unsafe Rust can be a powerful tool to fine-tune performance and compensate for Rust's restrictions.

4.2 Zebu

Zebu VM [Lin, 2019] is an efficient implementation of the Mu µVM specification in Rust. It consists of three tightly-coupled major components, namely a garbage collector, threads, and compiler, corresponding to the the key Mu abstractions.

The GC in Zebu is an implementation of the Immix garbage collector [Blackburn and McKinley, 2008] that supports parallel (thread-local) allocation and parallel collection. It was the first Zebu component to be written in Rust, and served as a proof-of-concept for the usability of Rust as a language for high-performance virtual machine implementation. Lin et al. [2016] report that their GC implementation in Rust, with only 4 % unsafe code, is less than 1 % slower than the same GC implemented in C, and it significantly outperforms the popular Boehm-Demers-Weiser (BDW) collector implemented in C [Boehm et al., 1991].

Mu threads in Zebu are implemented using the `std::thread::spawn()` function from Rust's standard library. Each Rust standard thread maps to a native OS thread. For instance in Linux, Rust threads are implemented using `pthreads`. After creating a new thread, *Zebu* does a `SwapStack` from the default thread stack to the Mu stack bound to the new Mu thread.

The Zebu's compiler is designed in alignment with the Mu requirements of minimality, efficiency, and flexibility. The only major optimizations included in the compiler are instruction selection and register allocation, which cannot be done by the client. Most of the IR-level optimizations are deferred to clients or optimizer libraries, to preserve minimality and verifiability. The compiler also supports two simple code patching mechanisms to allow efficient implementation of dynamic languages. For introspection of execution state, Zebu builds a stack map and a call-site table to support zero-cost exception handling and stack introspection with `KeepAlives`. Performance evaluation of the Zebu compiler backend using micro-benchmarks shows a slowdown of 2 % to 25 % compared to LLVM, which is reasonable considering Zebu's minimalism and comparative immaturity [Lin, 2019].

4.3 RTZebu

RTZebu is our implementation of the RTMu specification in Rust, demonstrating the feasibility of RTMu's abstractions. In the rest of this chapter, we briefly discuss two interesting aspects of RTZebu: (a) how most abstractions are implemented as light-weight wrappers around RTOS services, and (b) when and why we had to violate Rust's safety.

4.3.1 Compiler Backend

To implement the compiler backend of *RTZebu*, we reuse the AOT compiler of *Zebu* and add two categories of changes.

First, we add the support for the new RTMu types mentioned in Table 3.1. All of these new types are opaque reference types, and they share most of their properties, including size, alignment, and register group, with existing opaque reference types, such as `threadref` and `stackref`. The *RTZebu* runtime abstracts all these types as `Address`, as proposed by Lin et al. [2016].

Second, we modify the compiler to translate the new RTMu IR operations. These new operations work on the RTZebu runtime. To emit code for them, the compiler needs to know the runtime entry point for each of these operations. It also needs to know the input argument types for these operations. Thus, we add new runtime entry points and their argument types to RTZebu, following the same structure previously implemented for Zebu.

4.3.2 Threads and Scheduling

Rust provides two interfaces for creating new threads. The first one is by calling the `std::thread::spawn()` function, and specifying the thread body as a Rust closure or an object of a type that implements the `std::ops::FnOnce` and `std::marker::Send` traits. In this case, the thread is created with the default attributes. The second one is through the `std::thread::Builder` type which allows configuring the `name` and `stack_size` parameters of a new thread, before it is spawned. At the time of writing this thesis, Rust does not provide any other safe interfaces for creating and configuring threads.

Zebu uses the `std::thread::spawn()` function to create its threads. All new Zebu threads (created using `NewThread`) swap to their bound Mu stack (created using `NewStack`) as soon as they start, so Zebu does not need to change the stack sizes. Additionally, the Mu specification does not require configuration of any other thread attributes, and Rust's safe interfaces are sufficient for Zebu. On the other hand, the RTMu specification provides the instructions mentioned in Table 3.3 to control thread attributes and scheduling. However, most of these instructions cannot be implemented in safe Rust.

One solution is to use Rust's `libc` crate [TheRustProjectDevelopers, 2020] which provides raw foreign function interface (FFI) bindings to system libraries. For example, to implement the RTMu ThreadSetPriority instruction, libc for Linux provides the following unsafe function and its input argument types:

```
pub unsafe extern "C" fn pthread_setschedprio(
    native: pthread_t,
    priority: c_int
) -> c_int
```

The first argument of this function is the target thread's id, assigned by the pthread_create() function when the thread is created. However, this id is not known to the application when the thread is created using the safe Rust interface.

A workaround to this problem is that each RTZebu thread calls the pthread_self() function from libc to save its pthread id as soon as it starts. Because each Rust thread is mapped to one pthread, any future operations to change a RTZebu thread's attributes can be translated to a relevant function from libc, with the saved pthread id as one of its arguments.

While the previous workaround enables assigning a RTZebu thread's attributes after it starts, it does not allow creating threads with custom attributes as required by the NewRTThread instruction of RTMu. To support this, we need to switch from the safe Rust thread interfaces to the unsafe libc::pthread_create() function that takes the initial thread attributes as an argument.

In theory, using libc bindings should be sufficient for implementing RTMu threading instructions. However, at the time of writing this thesis, the Rust libc crate for Linux does not provide bindings for all of the pthread related functions required by our implementation. For instance, to modify the priority and scheduling policy of a thread attribute object, pthread_attr_setschedparam() and pthread_attr_setschedpolicy() are respectively required, but libc lacks bindings to these functions. Hence, we need to declare them as extern "C" functions.

Complexity

All runtime entry points for the thread management instructions in Table 3.3, except the NewRTThread instruction, are implemented in a similar way to the example in Listing 4.1. The entry point function, e.g. muentry_thread_set_priority() in Figure 4.1a, is platform-independent and consists of three steps: (1) initial processing of the input arguments which often includes unsafe access to input data through pointers or Addresses, (2) calling a platform-dependant function, e.g. sys_thread_set_priority() in Figure 4.1b, to perform the main operation, and (3) returning the operation's result or checking its outcome. The platform-dependant function includes calling a libc or external function, accompanied by platform-dependant processing of input arguments and/or a return value.

A.4.2- added a sentence to caption

Dynamic Priority Scheduling

As explained in Section 3.3.5, RTMu implementations may choose not to support the EDF scheduling algorithm, if the underlying platform (e.g. RTOS) does not provide

```
pub type MuPriority = SysPriority;

#[no_mangle]
pub extern "C" fn muentry_thread_set_priority(
    muthread: *mut MuThread,
    priority: u64
) {
    let tid = {
        // null means the calling thread
        if muthread.is_null() {
            let mut cur_thread = MuThread::current_mut();
            cur_thread.sys_thread_id
        } else {
            unsafe { (*muthread).sys_thread_id }
        }
    };
    assert_neq!(tid, 0);
    // call the platform-dependant function
    let res = sys_thread_set_priority(tid, priority as MuPriority);
    assert_eq!(res, MU_SUCCESS);
}
```

(a) The runtime entry function for the `ThreadSetPriority` instruction processes the input arguments, calls the platform-dependant function to set the target thread's priority, and checks its outcome.

```
pub type SysThreadID = libc::pthread_t;
pub type SysPriority = libc::c_int;
pub type SysResult = libc::c_int;

pub fn sys_thread_set_priority(
    native: SysThreadID,
    priority: SysPriority
) -> SysResult {
    let sch_param = libc::sched_param {
        sched_priority: priority
    };
    unsafe {
        libc::pthread_setschedparam(
            native,
            MU_DEFAULT_SCHED_POLICY,
            &sch_param as *const libc::sched_param
        )
    }
}
```

(b) The POSIX-specific function to update a thread's priority.

Listing 4.1: An implementation of a sample thread-related instruction of RTMu. All thread management instructions of RTMu, except `NewRTThread`, are implemented similarly. They are translated to a call to a platform-independent entry function at run time. The entry function calls a platform-dependant function to perform the main operation. Note that this is only a sample implementation and does not consider any specific requirements such as the guarantee for no exception in some hard real-time systems.

EDF or any other dynamic priority scheduling algorithms, and a correct and efficient implementation of EDF on the platform is not possible.

RTZebu is currently only implemented on top of RT-Linux which provides EDF only at the highest priority level. As shown in Figure 3.2, supporting EDF at a static priority level requires controlling the order of threads in the queue of ready threads at that priority level, but RT-Linux does not provide any interfaces to efficiently manage this order (the order is always FIFO). Besides, other RTOS services may inherit this limitation. For instance, the implementation of PIP in RT-Linux's PI-futex, prioritizes threads with deadlines over all other threads. Consequently, RTZebu does not support EDF at all priority levels.

4.3.3 Synchronization

In the initial RTMu design [Amiri et al., 2019], the only synchronization primitive was the priority inheritance (PI-) futex which exposed all PI-aware operations of the RT-Linux futex. During the implementation of RTZebu, we changed it to the current design which includes PI-futex as a fast user-level PI-aware mutex lock, plus a PI-aware condition variable (explained in Section 3.3.5, and Tables 3.4 and 3.5). The main reason behind this decision is that the PI-aware operations of the RT-Linux futex are specifically designed for implementing POSIX's PI-mutex and PI-condition-variable. In contrast, the normal Linux futex is more flexible and does not target any particular higher-level primitives. Besides, the new PI-futex and PI-condition-variable are portable and easier to use.

Complexity

We implement the RTMu PI-futex using the PI-aware operations of the RT-Linux futex. For each 32 bit futex word, we keep an instance of the `AtomicU32` type from the `std::sync::atomic` module of the Rust's standard library. The uncontended fast-path is mapped to a user-level call to the `AtomicU32::compare_exchange()` function. The slow-path that includes blocking or unblocking threads are implemented using a futex system-call.

We implement the RTMu PI-condition-variable using pthread's PI-aware condition variable. For each RTMu condition variable, we maintain two objects: an object of type `libc::pthread_cond_t` which is the internal mutex of the condition variable, and an object of type `libc::pthread_mutex_t`. Each condition variable instruction in Table 3.5 maps to one relevant pthread mutex or condition variable function.

4.3.4 Memory

RTZebu implements all five RTMu memory areas, mentioned in Section 3.3.4. The only major limitation with RTZebu's memory management that we are aware of is that real-time garbage collection (RTGC) is not implemented for RTZebu's heap, as it is an explicit non-goal of this thesis. Thus, it does not address RTGC-related concerns such as avoiding fragmentation.

A.4.3-change-1: remind scope

Allocating Memory

Implementing RTMu's `NewRegion` and `EAlloc` family of instructions requires dynamic allocation of memory blocks with various sizes. Rust allows this through the `unsafe` interfaces of the `std::alloc` module of the standard library. At the time of writing this thesis, the `std::alloc` module relies on the default operating system allocator (e.g. `malloc()` on Unix) to implement its functionalities.

On RT-Linux, allocating memory expands the address space without immediately allocating physical memory pages. Each time a memory page is touched for the first time, a page-fault occurs and the kernel maps physical pages then. This may lead to more efficient usage of the physical memory space. However, on a real-time system, this may cause additional delay throughout the system's runtime.

Rust's `std::alloc` module as well as the RT-Linux `malloc()` function do not provide any means for the application to avoid this lazy allocation. To solve this issue for the `NewRegion` instruction, we had to switch to the lower-level `mmap()` function [1] which allows populating page tables for a mapping, so that later accesses will not be delayed by page-faults. All RTZebu threads run in a single address space. The client RTPL or their application developers are responsible for correctly using the shared addresses.

A.4.5- mmap for newregion

A.4.6-change2: footnote on mmap WCET

A.4.3-change-2: added for concurrent allocs and sharing

Regions

We implement the `RAlloc` family of instructions as bump pointer allocation inside a region. This includes checking the region limit, updating the free space pointer, and returning the newly allocated object's address. To collect the memory region object, we simply reset the free space pointer to the region start.

To implement the `RefToReg` instruction, we keep a vector of tuples protected by a read-write lock. Each region has an entry in the vector which includes its starting address and size. To find the region, we do a linear search over the vector entries. This works efficiently as long as the number of active regions in the system is small, as is the case for our evaluations in Chapter 6. For more complicated cases, this needs to be further optimized as it directly affects the performance of some write barriers (e.g. the one explained in Section 5.3.2).

Reference Types

In the initial RTMu design in [Amiri et al., 2019], we used untraced pointers (e.g. `uptr <T>`) as the result type for `RAlloc` and `EAlloc` family of instructions. The rationale for this design was to maintain an untraced world where GC did not scan memory areas that were not affected by traced allocation, in addition to the traced world where the GC is allowed to scan any memory areas.

[1] As one of the examiners stated, bounding the WCET of the `mmap()` function is almost impossible. While this is a true concern in hard real-time systems, it is not an issue in the context of this thesis where all implementations and testings are performed on RT-Linux.

In practice, using two separate reference types introduces complexities specially on the language implementer's side. For example, the client language may need to create up to 2^n copies of a function, where n is the number of the function's input and output arguments with reference types. Thus, in the current RTMu design, we use references (e.g. `ref<T>`) as the result type for all object allocation instructions.

4.3.5 Time

Rust's standard library provides the `std::time` module which allows safely measuring time in the nano-second scale. In Unix, this module relies on the `clock_gettime()` system call and reads the `CLOCK_MONOTOIC` clock. We implement RTMu's `GetTime` and `SetTime` instructions using this module in safe Rust.

Implementing a timer as required by the RTMu specification is possible in Rust using the `timer` crate. For each new timer, this crate creates two threads. A *scheduler* thread that maintains and runs call-backs, and a *communication* thread that handles the communication between the scheduler thread and the caller thread.

The way the `timer` crate implements a timer is not suitable for real-time systems, mainly because the two timer handler threads are created regardless of the attributes of the caller thread. For instance, if a high-priority real-time thread creates a timer, the timer handler threads run at the RTOS's default thread priority, and may be delayed by lower-priority threads or their timer handler threads. This may introduce unpredictability to the timing behaviour of a high-priority task, and is often not desirable.

To avoid the above problem, we implement the timer instructions of RTMu using the POSIX per-process timer. We did this in unsafe Rust through the system-call wrappers of the `libc` crate.

4.3.6 Technical Challenges

RTZebu is written in Rust, a young and quickly evolving programming language. New features are frequently being added to Rust, and new opportunities are created to improve our implementation. For instance, the `#[global_allocator]` attribute which became stable in Rust-1.28.0, enabled a considerably cleaner implementation of the Rust object dumper and loader (*rodal*) used by RTZebu. Existing features are also being constantly enhanced. For example, the implementation of the `std::collections::HashMap` type was changed since Rust-1.36.0 to improve its performance. For RTZebu, this meant *rodal* would need to be updated to work on Rust-1.36.0 and any newer releases. In summary, Rust's rapid changes improved the language and allowed us, and looks likely to allow us to achieve higher-quality performant implementations, but they also significantly lengthened the development cycle of RTZebu, as more maintenance was required to keep-up with the changes over time.

4.4 Summary

In this chapter we presented our implementation of the RTMu specification, named *RTZebu*. We enumerated the key features of Rust, the programming language used for the implementation of *RTZebu*. We introduced *Zebu*, the high-performance implementation of the *Mu* µVM specification, as the starting point for *RTZebu*. Finally, we described the *RTZebu* implementation in Rust, including how most RTMu operations are implemented as lightweight wrappers around RTOS services, and why we had to frequently use unsafe Rust, specially to implement the concurrency related operations of RTMu. In the next chapter, we explain a new real-time language we designed and implemented on RTZebu to demonstrate the expressiveness of the RTMu's abstractions.

Chapter 5

RT-RPython: A Python-like Language for Real-Time Systems

To demonstrate the expressiveness of RTMu's abstractions, we propose a new managed language named RT-RPython and implement it on top of RTMu. RT-RPython is based on the RPython language [RPyDoc], modified to suit the requirements of a range of real-time systems targeted by the real-time specification for Java [RTSJ, 2018].

In this chapter, we first briefly introduce RPython, focusing on the aspects that need to be changed to become suitable for real-time systems. Then we elaborate our proposed changes and illustrate how they are implemented on top of RTMu.

The main contributions of this chapter are the design of RT-RPython as a high-level managed language for real-time systems, and the demonstration of its implementability on RTMu.

5.1 RPython

A.5.9- added AOT compiled

RPython is an ahead-of-time compiled restricted subset of Python-2, which mainly acts as an efficient framework to generate JIT compilers from interpreters of dynamic programming languages. The restrictions enable static typing and compile-time optimization of RPython programs. RPython is a part of the PyPy project [PyPyDoc] and has also been used for other languages such as SOM, Racket and Erlang.

RPython is designed to target multiple backends, but officially it only provides a C backend. The work of building a Mu backend for RPython (MuPy) was started by Zhang [2015], with the goal of supporting the wide range of RPython's client languages on top of Mu. At the time writing this thesis, the core PyPy interpreter which is written in RPython could be compiled on MuPy.

5.1.1 Application Programming in RPython

RPython was designed as a restricted language specifically for implementing PyPy, however it can be used for application programming with some limitations. The most restrictive properties of RPython from an application programmer's point of view are:

```python
def to_int(input):
    return int(input)

f = to_int(1.0)     # input argument is inferred as float
s = to_int(’1’)     # error: cannot unify float and string
```

(a) Types of all input arguments of all user-defined functions must be mappable to exactly one type at compile time. The input arguments’ types are inferred on the first call to the function, seen by the compiler. Any subsequent calls that violate these types will raise a compile error.

```python
def func(i):
    if i == 0:
        return 0    # return variable is inferred as integer
    else:
        return ’1’  # error: cannot unify integer and string
```

(b) Types of all variables in a function, including the return variable, must be mappable to exactly one type at compile time. Each variable’s type is inferred the first time it is seen by the compiler.

Listing 5.1: Examples of compile errors caused by RPython variable restrictions. Both examples are valid programs in Python.

- RPython variables can only contain values of at most one type in each control flow point. This means that the valid Python codes in Listing 5.1 will not compile in RPython, because the type for their input arguments or return value cannot be inferred at compile time.
- Global variables and static class members are constants in RPython. The only way to have mutable global or static data is to wrap it in a class and create a global/static instance of the class.
- Many of the special class methods of Python-2, including `__hash__()`, comparison operators (e.g. `__eq__()`), binary operators (e.g. `__add__()`) and `__str__()` are ignored.

RPython provides libraries to compensate for some of these limitations. For instance, the `r_dict` library allows creating dictionaries with custom hashing and equality functions, while this is done by overloading `__hash__()` and `__eq__()` methods in Python.

5.1.2 Real-Time Programming in RPython

RPython is a garbage collected language. Garbage collectors, especially the ones not designed for real-time systems, can negatively affect the timing behaviour of applications. For instance, a garbage collector may pause the whole application or preempt high priority tasks, which is not desirable. Although there exist garbage collectors for real-time languages, such as the real-time specification for Java (RTSJ), even these do not provide the required level of guarantees for some hard real-time systems. In Section 5.2.1, we propose a set of changes to RPython’s memory manager to make it suitable for a wide range of real-time applications.

RPython supports multi-threading, but threads must hold the Global Interpreter Lock (GIL) before they are executed. This means there is no true concurrency in RPython. Consequently, there is no need for inter-thread synchronisation primitives such as locks or condition variables. This level of support for multi-threading is insufficient for many real-time systems. We address this issue in Section 5.2.2.

5.2 Real-Time Extensions

RT-RPython is a high-level managed language for real-time systems, aiming at the same range of applications targeted by the RTSJ. Hence, we design RT-RPython abstractions over memory and concurrency, analogous to the same abstractions in RTSJ.

A.2.4- change-2

RT-RPython is designed to demonstrate that RTMu's abstractions are capable of implementing a comprehensive set of real-time language features, including the RTSJ scoped memory. Thus, it inherits some of the limitations of RTSJ, such as not being compatible with many of the existing Python and RPython libraries. However, this does not mean that all real-time languages implemented on RTMu will have the same issue, because the language designer can choose not to expose specific RTMu features, including the regions.

5.2.1 Memory Management

The user manageable memory in RT-RPython consists of four types of areas: the heap, global memory, explicitly managed memory (EMM), and scoped memory. Among them, the heap and global memory are directly borrowed from RPython, which already implements both. The heap is the default allocation area in RT-RPython, and is garbage collected. The global memory includes the global application data and static class members. As in RPython, all data in RT-RPython's global memory is immutable.

RT-RPython's EMM is a new memory area which allows explicit allocation and reclamation of memory at the granularity of an object. It is similar to `malloc()` and `free()` in C.

The next new area is RT-RPython's scoped memory. It includes a number of fixed-sized scopes, which are created at the client's request. The client may allocate objects in each scope, but can only delete the whole scope. Each scope has a reference counter that is increased when a thread enters the scope and decreased when a thread exits. Every time the reference counter becomes zero, all objects in the scope are collected. The scoped memory in RT-RPython is inspired by the scoped memory in RTSJ.

Design Choices There are various ways to co-locate multiple memory area types in a language. For example, RTSJ, explained in Section 2.3.2, maintains a stack of allocation contexts [1] for each schedulable object. The area on top of the stack is called

A.5.4- added a footnote

[1] The internal representation of an allocation context is implementation-defined. In our current implementation of RT-RPython, each memory area is represented by an internal handle.

Table 5.1: To support the proposed memory management scheme in RT-RPython, the operations in this table are added. The first three operations change the current allocation context to a new one. The fourth operation reverts the allocation context to the previous one. The last two operations create a new scope or delete an already created scope.

Operation	Description
`enter_heap()`	Push the heap onto the caller thread's area stack
`enter_emm()`	Push the EMM to the caller thread's area stack
`enter_scope(scp: Ref<Scope>)`	Push the scope referenced by `scp` onto the caller thread's area stack
`exit_area()`	Pop the last memory area from the caller thread's area stack

`new_scope(size: int)`
`  -> Ref<Scope>`
	Create a new scope of `size` number of bytes

`delete_scope(scp: Ref<Scope>)`	Delete the scope and reclaims its memory

the current allocation context. Using the `new` operator allocates objects in the current allocation context. It is also possible to allocate objects in other allocation contexts in the area stack by calling the `newInstance` and `newArray` methods of the target area.

For the sake of simplicity, we choose to minimize the syntactical extensions of RT-RPython. Every thread in RT-RPython has a thread-local area stack, with the heap as its first element. A thread may then enter or exit various memory areas including the heap, the EMM and any available scope. Memory allocation is performed in the area on top of the area stack. New memory operations in RT-RPython are listed in Table 5.1. Also, sample code for allocating/deallocating an object on any of these areas is shows in Listing 5.2.

To prevent dangling references, we adapt the single parent rule and the reference restrictions of RTSJ [RTSJ, 2018] for RT-RPython as follows. Each scope with a non-zero reference count has a non-negative *rank* which reflects the number of unique scopes on the area stack, before the scope was first entered. Scopes may be entered multiple times by the same thread, but their rank is assigned only on the first entry, or on any entry that changes its reference count from zero to one. The parent of a scope X is the scope Y on the current area stack, if rank of X equals rank of Y plus one. The ***single parent rule*** of RT-RPython requires all referenced scopes on all area stacks to have exactly one parent at a time. This implies that each scope can only have one rank at a time, even if it is entered by multiple threads. The referencing rules of RT-RPython are summarized in Table 5.2. These rules preserve the integrity of references as long as the client uses the EMM correctly.

```python
class T:
    pass

def test():
    obj = T()
```

(a) The heap is the default allocation context. An object allocated on the heap is automatically reclaimed by the GC.

```python
class T:
    pass

def test():
    enter_emm()
    obj = T()
    obj.delete()
```

(b) EMM: The object is freed when its delete method is explicitly called.

```python
class T:
    pass

def test():
    scp = new_scope(16)
    enter_scope(scp)
    obj = T()
    exit_area()
```

(c) Scopes: The object is collected when the scope's reference count reaches zero (e.g. after exiting the current area).

Listing 5.2: Sample code for allocating an object on each of the three available allocation contexts in RT-RPython.

Table 5.2: RT-RPython prevents the storing of references to objects in scoped memory in the global or heap areas. It also prevents the storing of references to objects in scopes with shorter lifetimes. RT-RPython does not enforce any restrictions on EMM, as it expects the application developer to use the EMM correctly.

	Reference to Object in		
Stored in	**Global/Heap**	**EMM**	**Scopes**
Global/Heap	Permit	Permit	Forbid
EMM	Permit	Permit	Permit
Scopes	Permit	Permit	Permit from higher or same rank

Table 5.3: RT-RPython provides various functions to manage thread attributes. Normally, it starts by creating a new attribute object using `Attr()` or reusing a previously created object (e.g. by calling `get_thread_attr()`). Then, the object is monitored and updated using the functions in the last two rows of this table. At the end, `set_thread_attr()` is called to apply the updated attribute on the destination thread.

Operation	Description
`Attr()-> Ref<Attr>`	Create a new thread attributes object and returns a reference
`delete_attr(Ref<Attr>)`	Delete the specified thread attribute object
`set_thread_attr (Ref<Thread>, Ref<Attr>)`	Update the attributes of the specified thread
`get_thread_attr(Ref<Thread>) -> Ref<Attr>`	Return the current attributes of the specified thread
`set_attr_priority(Ref<Attr>, Int)`	Set the priority field of the attribute object
`get_attr_priority(Ref<Attr>) -> Int`	Return the current priority field of the attribute object
`set_attr_cpu(Ref<Attr>, Int)`	Add the specified cpu id to the current attribute
`isset_attr_cpu(Ref<Attr>, Int) -> Bool`	Check whether the specified cpu id is in the current attribute
`clear_attr_cpu(Ref<Attr>, Int)`	Remove the specified cpu id from the current attribute
`zero_attr_cpu(Ref<Attr>)`	Remove all cpu ids from the current attribute

5.2.2 Concurrency

Threads and Parrallel Execution

RT-RPython supports parallel execution of threads. To create a new thread, the client should call the following function:

```
start_new_rt_thread(
    entry_function: Func, thread_attributes: Ref<Attr>, *arguments
) -> Ref<Thread>
```

The first argument is the thread entry function. The new thread calls `entry_function (*arguments)` when it starts execution. The `*arguments` enables giving an arbitrary number of inputs to the thread. The second argument is used to specify attributes of the new thread, including its priority and affinity. Table 5.3 shows the RT-RPython functions provided for this purpose.

Inter Thread Communication

In a language that supports truly concurrent threads, thread synchronization primitives are essential. In RT-RPython, we add three basic forms of synchronization. First, we add `Mutex` which serves as a fast user-level mutual exclusion lock supporting

Table 5.4: RT-RPython `Mutex` is a mutual exclusion primitive that supports the PIP and PCP. It provides three variants of `lock()`, and one `unlock()` method. It also provides the `setpc()` method to activate PCP or update the ceiling priority, and the `unsetpc()` method to switch back to PIP. Before using a new mutex, the `initialize()` method must be called.

Operation	Description
`mtx = Mutex()`	Create a new PI mutex
`mtx.initialize()`	Initialize a PI mutex
`mtx.lock()`	Lock the mutex
`mtx.trylock()`	Lock the mutex if it is currently unlocked, or immediately throw an exception
`mtx.timedlock(t: Int)`	Lock the mutex before `t` nano-seconds or throws an exception
`mtx.unlock()`	Unlock the mutex
`mtx.setpc(pc: Int)`	Set a priority ceiling for the mutex; Activates PCP if required
`mtx.unsetpc()`	Deactivate PCP for the mutex; Switch to PIP

the Priority Inheritance Protocol (PIP). When there is no contention over a Mutex, locking and unlocking is done by user-level atomic operations. In case of contention, the RTOS kernel is involved to block and unblock threads. For the PIP to work correctly, RT-RPython only allows the locker thread to unlock it. Second, we add `ConditionVariable` as a mechanism that allows multiple threads to wait for occurrence of a condition. Priority is inherited from the waiting threads to the thread which signals the condition. Third, we add `AtomicInt` which represents a 64 bit integer with basic atomic operations. Details of these primitives are shown in Tables 5.4, 5.5 and 5.6.

5.2.3 Time

To manage time, RT-RPython provide three basic tools. The first one is a high-resolution clock which measures time in nano-seconds. The second is the sleep operation that puts the calling thread to sleep for a certain number of nano-seconds. The third is the timer that calls a handler function at a designated time or periodically. Details of the time management operation are shown in Table 5.7. The accuracy of these operations depends on the underlying platform, including the RTOS kernel and hardware.

5.3 Implementation

We use RPython with the Mu backend (RPyMu) as the starting point for the implementation of RT-RPython. Details of the RPyMu implementation are explained by Zhang [2015]. Here, we introduce the RPyMu translation process, and briefly explain how we change it to support the RT-RPython extensions.

Table 5.5: RT-RPython `ConditionVariable` is a synchronization construct that allows multiple threads to wait for a condition. Each condition variable is associated with a mutex lock that must be acquired before any wait, signal, or broadcast operation, and released after their completion.

Operation	Description
`cv = ConditionVariable()`	Create a new PI condition variable
`cv.initialize()`	Initialize a new PI condition variable
`cv.wait(mtx: Ref<Mutex>)`	Wait for a signal on the condition variable
`cv.timedwait (mtx: Ref<Mutex>, t: Int)`	Wait for a signal on the condition variable; Throw an exception if not signalled within `t` nano-seconds
`cv.signal()`	Signal the highest priority thread waiting on the condition variable
`cv.broadcast()`	Signal all threads waiting on the condition variable
`cv.delete()`	Delete the condition variable and reclaims its memory

Table 5.6: The RT-RPython `AtomicInt` type represents an atomic integer. It provides a basic set of atomic operations.

Operation	Description
`ai = AtomicInt()`	Create a new atomic integer
`ai.load()-> Int`	Load the current value of the atomic integer
`ai.store(new_val: Int)`	Store a new value into the atomic integer
`ai.compare_exchange (current_val: Int, new_val: Int)`	Store the `new_val` into the atomic integer if its current value equals `current_val`

Table 5.7: RT-RPython provides a basic set of functions to work with the clock and timers. The client can get/set the current time, suspend (sleep) the current thread for a certain amount of time, and create/delete and set/unset timers that schedule calls to handler functions.

Operation	Description
`get_time_ns()-> Int`	Get the current time value in nano-seconds
`set_time_ns(new_time: Int)`	Set (update) the current time value
`sleep_ns(duration: Int)`	Suspend the calling thread for the specified number of nano-seconds
`new_timer()-> Ref<Timer>`	Create a new timer
`delete_timer(tmr: Ref<Timer>)`	Delete a timer and free its memory
`arm_timer(tmr: Ref<Timer>, handler_func: FUNC, duration: Int, period: Int, *args)`	Set ‘`tmr`‘ to call ‘`handler_func(*arg)`’ after ‘`dur`’ nano-seconds, and every ‘`prd`’ nano-seconds after that
`disarm_timer(tmr: Ref<Timer>)`	Cancel a previously set timer ‘`tmr`’

5.3.1 The RPyMu Translation Process

The process of translating RPython code in RPyMu consists of seven tasks, of which the first three are common between the Mu and C backends. First, the ***annotator*** task gets the control flow graphs of the program and infers the general type information of variables. It also finds and annotates the functions called in the program code. Second, the ***rtyper*** specializes the general type and operation information from the annotator, producing close-to-C types and their methods. Third, the ***backendopt*** task performs some optional optimizations such as inlining and no-op removal, and transforms exceptions and garbage collection to C-compatible implementations. Fourth, the ***mutyper*** task maps the types, values and operations to their Mu equivalents. Fifth, the ***optimize_mu*** task removes some useless operations. Sixth, the ***database_mu*** task builds a database of all global information, including types, constants, external functions, global cells, graphs, and function references. Seventh, the ***compile_mu*** task uses the Mu IR Builder to generate and load a Mu bundle (explained in Section 2.4). The *compile_mu* task also allows making a boot image of the loaded bundle.

5.3.2 RT-RPython Extensions

Our first step to implement the real-time extensions, proposed in Section 5.2, is to add the new RTMu constructs (e.g. types, instructions and intrinsics) from Section 3.3, to RPyMu. Therefore, for each new RTMu type, we add a new RPython type (e.g. `class Thread` for `threadref`). Also, for each new instruction or intrinsic, we add a dummy RPython function. The names of these RPython types and functions are reserved words in RT-RPython.

mutyper The new types and functions go through the first three translation tasks as before. To this point, we only make some modifications to avoid unwanted optimizations (e.g. removal of arguments and inlining) of RT-RPython constructs. The *mutyper* task is where the mapping of RT-RPython types and dummy functions to their real RTMu counterparts is done.

Mapping to RTMu types is always a simple one-to-one translation. For example, the `Ptr<GCStruct Thread>` and `Ptr<GCStruct Region>` types from the RT-RPython *rtyper* are respectively mapped to `threadref` and `regionref` in RTMu. This is also true for most of the dummy functions. For instance, `new_timer()` and `get_time_ns()` are directly mapped to a`NewTimer` and `GetTime` instructions in RTMu. Other dummy functions map to more than one instruction or intrinsic. For example, `start_new_rt_thread()` maps to a `NewStack` intrinsic and a `NewRTThread` instruction in the simplest case.

Code Generation We reuse the ***optimize_mu*** and ***database_mu*** translation tasks of RPyMu without any change. For the ***compile_mu*** task, we add the logic to generate code for the newly added instructions and intrinsics. We also modify the RPyMu IR

```
Fn rtmu_allocate_<T>() -> ref<T>:
    int ca = rtmu_get_current_area()
    ref<T> res
    if ca == HEAP:
        res = New T
    elif ca == EMM:
        res = EAlloc T
    else:
        regionref reg = rtmu_get_current_scope()
        res = RAlloc reg, T
    Ret res
```

Listing 5.3: Simplified RTMu IR code for allocator functions. For each type T, we generate a new allocator function. This function executes the correct allocation instruction depending on the current allocation area. Calls to these functions may be inlined by the RTMu AOT compiler to reduce the overhead. The `rtmu_get_current_area()` and `rtmu_get_current_scope()` helper functions are written in RT-RPython and provide a bridge between user code (in RT-RPython) and RTMu IR code at run time.

generator for `New`, `NewHybrid` and `Store` instructions to support the requirements of RT-RPython.

For each '`New T`' or '`NewHybrid T`' instruction, we replace it with a call to an allocator function for type `T`. The function performs a run-time check for the current allocation context. Depending on the result, one of these instructions: `New`, `RAlloc`, `EAlloc`, or their hybrid counterparts is executed. The simplified RTMu IR code for allocator functions is shown in Listing 5.3.

For the `Store` instruction, we consider a special case when the new value to be stored is of RTMu `ref` type. In this case, we emit a write barrier to check whether this store instruction adheres to RT-RPython reference integrity rules. The barrier finds the memory areas containing the source and destination objects. If the destination area has a longer lifetime compared to the source area, an exception is thrown to indicate an illegal assignment. The simplified RTMu IR code for these barriers and a helper function is shown in Listing 5.4.

5.4 Summary

We developed RT-RPython, a high-level managed language for real-time systems. It is based on RPython and inherits its limitations [RPyDoc]. To keep the implementation complexity reasonable given the constraints of a PhD thesis, we choose to only address the restrictions, such as GIL-based multi-threading, which are essential in programming real-time systems, and leave the rest untouched. In the next chapter, we demonstrate RT-RPython's expressiveness, performance and predictability through implementing and evaluating a real-time benchmark.

```
Fn rtmu_ltchk_store_<T1>_<T2>(
    dst: T1,
    src: T2
):
    iref<void>  dst_vir = RefCast dst
    ref<void>   src_vr  = RefCast src
    Call rtmu_check_lifetime(dst_vir, src_vr)
    Store dst, src
    Ret
```

(a) This function performs the lifetime checked store operation. It casts the operands and calls a type-neutral function to check the lifetimes. If the checker function returns successfully, the Store instruction is executed.

```
Fn rtmu_check_lifetime(
    dst: iref<void>,
    src: ref<void>
):
    rs = RefToReg src
    // if src is in heap, emm, or immortal, its lifetime is infinite
    if rs != Null:
        rd = RefToReg dst
        // if dst’s lifetime is infinite
        if rd == null:
            Throw LifeTimeException
        // if both src and dst are in scopes
        ds = Call get_area_depth(rs)
        dd = Call get_area_depth(rd)
        if dd < ds:
            Throw LifeTimeException
```

(b) This function compares the lifetimes of the source and destination arguments and throws an exception if the source has a shorter lifetime.

Listing 5.4: Simplified RTMu IR code for RT-RPython write barriers and the reference lifetime checker function. When the source operand in a `Store` instruction is a reference, we replace the Store instruction with a call to a function. The name of the function to call is `rtmu_ltchk_store_<T1>_<T2>()`, which depends on types of the source and destination operands.

Chapter 6

Evaluation

Performing a comprehensive evaluation of a real-time programming language would require implementing a wide range of real-time applications, followed by defining and measuring various criteria. However, most real-world real-time applications are not publicly available. Besides, there is a lack of open-source real-time benchmarks, and the available ones are often micro-benchmarks that mostly evaluate the primitives of the underlying RTOS without giving much useful information about the language. Hence, for our evaluation of RT-RPython we chose to implement a real-time application benchmark suite proposed by Kalibera et al. [2009].

A.6.1- added a sentence

In this chapter, we first introduce the Collision Detection (CD$_{\mathrm{x}}$) benchmark. Then, we explain CD$_{\mathrm{rr}}$, our implementation of CD$_{\mathrm{x}}$ in RT-RPython, and identify its differences to the CD$_{\mathrm{x}}$ implementation in RTSJ (CD$_{\mathrm{j}}$). Finally, we compare the performance results of CD$_{\mathrm{rr}}$ to CD$_{\mathrm{j}}$, and argue that RT-RPython establishes RTMu as an efficient foundation for the development of programming languages suitable for building real-time software. We acknowledge that the CD$_{\mathrm{x}}$ benchmark suite does not evaluate every aspect of RT-RPython such as the explicitly managed memory, and that using more real-time benchmarks could provide a more comprehensive evaluation if they were available to us.

A.6.3- added one sentence

6.1 The Collision-Detection Benchmark

The Collision Detection (CD$_{\mathrm{x}}$) benchmark suite, published by Kalibera et al. [2009], is a configurable open-source real-time application benchmark that targets various RTSJ and Java VMs. The main components in the CD$_{\mathrm{x}}$ benchmark are an air traffic simulator (ATS), that generates radar frames, and a collision detector (CD), which is a periodic task that implements an aircraft collision detection algorithm. The algorithm includes two steps, reduction and collision checking, which both mostly consist of floating-point calculations. The ATS task runs in advance and generates the radar frames for all CD periods. At each period, the CD task processes the designated frames. The sequential execution of these tasks means the choice of scheduling algorithm does not affect the benchmark results [1].

A.6.2- added some sentences

[1] CD$_{\mathrm{x}}$ supports running ATS and CD in parallel, only for debugging.

Details of the ATS and CD tasks can be found in Kalibera et al.'s paper. Here, we only highlight the language features required to implement these tasks. Some of these language features are only common in real-time applications:

- concurrent ***threads*** (to run ATS and CD),
- ability to control thread ***priority***,
- ***periodic*** threads,
- ***PI condition variable***, ***PI mutex***, ***atomics*** and ***join*** to synchronize threads,
- high-resolution and accurate ***clock*** to measure performance metrics,
- high-resolution and accurate ***sleep*** function to implement waiting for the next period which happens every 4 to 500 milliseconds,
- ***scoped*** and ***immortal*** memory areas.

Some other features such as file I/O, strings, exception handling, classes and inheritance, and data structures (e.g. list, dictionary and set) are commonly used in non-real-time applications as well.

6.1.1 Implementation in RT-RPython

To implement the CD benchmark in RT-RPython ($\mathrm{CD}_{\mathrm{rr}}$), we tried to transliterate the RTSJ implementation (CD_{j}) to RT-RPython syntax. However, as the two languages are not identical, slight structural modifications were inevitable. In this section, we enumerate these changes and reason why the benchmark implementation is still valid with the changes.

Loops CD_{j} uses C-style for-loops, as in Listing 6.1a, to implement running code a certain number of iterations. The visually closest translation of this for-loop in RT-RPython is a Python-style for-loop as in Listing 6.1b. However, the semantically closest translation of a C-style for-loop in RT-RPython is a while-loop with a counter, as in Listing 6.1c. In $\mathrm{CD}_{\mathrm{rr}}$, we choose to use the semantically closer alternative, as we found that using a Python-style for-loop leads to considerable run-time overhead, because it iterates over the output of a `range()` function, and that is much more complicated than the simple counter in the C-style for-loop.

Globals and Statics As mentioned in Section 5.1.1, global and static variables are immutable in RPython, and RT-RPython inherits the same restriction. In contrast, CD_{j} uses mutable static class members frequently. Our solution to this conflict is to change static class members to non-static members wherever they don't have to be static, otherwise, we put them in a wrapper class. Therefore, while the wrapper object is still immutable, the wrapped object may be mutated. This adds an additional level of indirection compared to RTSJ, and may cost some overhead. In $\mathrm{CD}_{\mathrm{rr}}$, access to these variables does not occur in the performance-critical path.

```java
for(int i = 0; i < MAX; i++){
    // code
}
```

(a) C-style for-loop in RTSJ

```python
for i in range(MAX):
    # code
```

(b) Python-style for-loop

```python
i = 0
while i < MAX:
    # code
    i += 1
```

(c) Python while-loop with a counter

Listing 6.1: Loops with a certain number of repetitions are written as C-style for-loops in RTSJ. The visually closest equivalent of C-style for-loop in RT-RPython is the Python-style for-loop, but the semantically closest equivalent is the while-loop with a counter.

Threads and Synchronization As shown in Listing 6.2, RT-RPython and RTSJ use syntactically different approaches for configuring and creating a thread. However, the outcome and features are the same. In addition, CD$_{\text{j}}$ uses Java monitors for synchronization, whereas CD$_{\text{rr}}$ uses condition variables.

Array of Bytes In CD$_{\text{j}}$, allocating and processing byte-arrays are done frequently. In RTSJ, an array of `size` bytes can simply be created by writing `new byte[size]`. But in RT-RPython, a byte-array can only be created using the `bytearray(s)` function, and the input to this function can only be a string. So, creating the same array in RT-RPython can be written as `bytearray(b'\x00' * num_of_bytes)`, which is a slower operation.

Special Class Methods As mentioned in Section 5.1.1, many of the special class methods of Python are not honoured in RT-RPython, which means the default special methods are used even if we overload them. As a result, translating the overloaded operators of CD$_{\text{j}}$ to their equivalent in RT-RPython is not sufficient for them to work in CD$_{\text{rr}}$. We also need to explicitly call the overloaded operators wherever they are used. For example, if we overload the `__eq__` method of a class, we should write the statement `obj1 == obj2` as `obj1.__eq__(obj2)` for objects of that class. In other words, these special methods are treated as normal object methods.

6.1.2 Non-Goal

The current implementation of RT-RPython does not come with a real-time GC, as it depends on RTZebu's memory management which does not currently include a real-time GC implementation. It has not been a goal of this thesis to design and implement a real-time GC for RTZebu.

```java
// definition of a new realtime thread
public class MyThread extends RealTimeThread {
    public MyThread(SchedulingParameters sp, ...) {
        super(sp, ...)
        // constructor code
    }
    ...
    public void run() {
        // thread body
    }
}
...
PriorityParameters pp = new PriorityParameters(PRIORITY);
// running the thread
final MyThread new_thread = new MyThread(pp, ...);
new_thread.start()
```

(a) In RTSJ, any real-time thread must extend the RealTimeThread class or one of its children (e.g. NoHeapRealTimeThread). The run method of the new class is used by the start method as the thread body.

```python
def run():
    # thread body
...
a = new_attr()
set_attr_priority(a, PRIORITY)
# running the thread
start_new_rt_thread(run, a)
```

(b) In RT-RPython, creating a thread needs an entry point, and a thread attribute object. The entry point can be a function or a static class method, and does not need to be named `run`.

Listing 6.2: Creating a thread with a specific priority in RTSJ and RT-RPython.

6.2 Test Setup

To evaluate the performance of RT-RPython as a real-time language, we compare CD_{rr} (explained in Section 6.1.1) to CD_{j} on JamaicaVM-8.3 as a RTSJ implementation. We also run CD_{j} on Java-11 (Hotspot JVM-11) to highlight the difference between a non-real-time language that is highly optimized for average-case performance, and real-time languages.

We run all tests on a 64 bit Kubuntu-18.04.3 PC. The Linux kernel version is 4.19.23-SMP with PREEMPT-RT real-time patch. The processor on this machine is a quad-core Intel Core-i7 4790, clocked at 3.6 GHz. The machine also has 24 GB of dual channel DDR3-1600 RAM.

6.2.1 RT-RPython

We test two implementations of RT-RPython. The original one (RT-RPython) that emits write barriers to check the referencing rules in Table 5.2, is evaluated in this chapter. The second one (ucRT-RPython) does not perform these checks. We report ucRT-RPython's results in Figure 6.4. Comparing these two implementations, we can estimate the overhead of write barriers. More efficient implementation and emission of these write barriers in the future can help us reduce the performance gap.

6.2.2 JamaicaVM

JamaicaVM is a commercial hard real-time implementation of RTSJ by *aicas*, that targets a wide range of embedded and real-time systems including automotive, IoT and other critical embedded systems [AicasWebPage]. It claims to be capable of running standard Java code in real-time embedded systems through its deterministic real-time GC. We compare the predictability of JamaicaVM's real-time GC against its scoped memory in Appendix Figure 1.

To enable fine-grained control over the scheduling in Java code, JamaicaVM performs a one-to-one mapping of Java threads to native RTOS threads [Siebert, 1999]. For the real-time GC of RTSJ, JamaicaVM implements an incremental mark-sweep GC that can run concurrently with the application, and may use multiple CPUs to run GC work in parallel [Siebert, 2010]. It also provides a stop-the-world GC, for applications that require better average-case performance and have no real-time constraints [Aicas, 2019].

The workloads we use for our evaluations (introduced in Section 6.2.4) do not use the garbage-collected heap; they use scoped memory regions for all dynamic allocations. However, the choice of GC may still affect the performance results. So, we compare the results of the two JamaicaVM GCs in Appendix Figure 1, and choose to use JamaicaVM with the real-time (incremental) GC in our evaluation, because of its more predictable behaviour.

A limitation JamaicaVM is not an open-source VM and we only have access to the documents published on *aicas*'s website [AicasWebPage] (including JamaicaVM's user manual and some research papers). Additionally, our academic-use-only license does not include all of JamaicaVM's development tools (e.g. JamaicaTrace) which might help us extract more information about the timing behaviour of the benchmark. As a result, we are unable to present detailed analysis of JamaicaVM's results in some of our tests. We contacted *aicas* for more information (e.g., about the relatively high release jitter), but at the time of writing this thesis, we have not received any response.

6.2.3 Hotspot-11

Hotspot JVM-11 is the most recent stable (LTS) release of Java virtual machine by Oracle at the time of writing. It provides four garbage collectors, each with different performance characteristics [Oracle, 2018]:

(1) The *serial* GC is a single-thread generational collector which is suitable for single-processor machines or applications with a small amount of data (e.g. less than 100 MB).

(2) The *parallel* GC is a multi-thread generational collector optimized for throughput. It is suitable for applications with medium to large amount of data that run on multi processors.

(3) The *Garbage-First (G1)* GC is a generational, incremental, parallel, mostly concurrent, stop-the-world, and evacuating collector which aims at achieving high throughput while maintaining small pause times with high probability.

(4) The *ZGC* is a new experimental collector which aims at small pause times (e.g. less than 10 ms) on small to very large heap sizes (e.g. 100 MB to multi-TB).

We compare the real-time performance of all these GCs in Appendix Figure 2, and choose to use Hotspot JVM-11 with ZGC in our evaluation.

6.2.4 Workloads

CD_j comes with many predefined workloads, and tools to generate new workloads. It also provides a number of configuration options such as memory noise parameters, workload size and complexity, memory area sizes, and dumping debug information.

CD_rr can run any workload from CD_j. However, we only evaluate using the two main workloads from [Kalibera et al., 2009], named COL and NOI. These workloads are summarized in Table 6.1. As mentioned in Section 6.1.2, RT-RPython does not provide a real-time GC. Consequently, we use our version of the NOI workload, called NOI_nn, which is different from the original one in that it does not generate any noise to test the real-time garbage collection, on all tested VMs.

A.6.1- add no noise on all VMs

Table 6.1: A summary of COL and NOI_{nn} workloads.

	COL	NOI_{nn}
Period	10 ms	4 ms
Collisions	YES	NO
Number of Aircraft	40	20
Duration	100 s	80 s

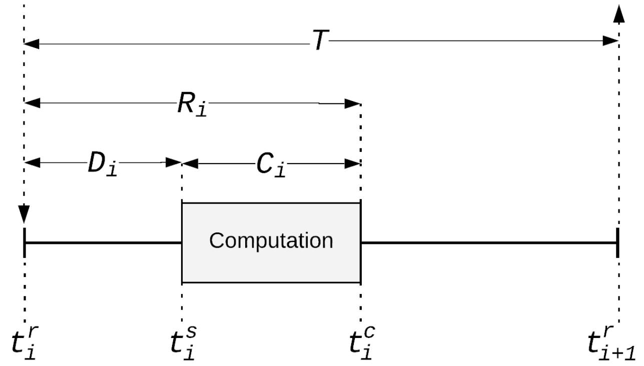


Figure 6.1: CD is a single periodic task with a period and deadline equal to T. The i_{th} release of CD is expected to occur at t_i^r. For this release to meet deadline, its response time (R_i) must be less than its deadline (T). Response time (R_i) equals the sum of release delay (D_i) and computation time (C_i). Computation time (C_i) is the time from the i_{th} actual start time (t_i^s) of the task, to its i_{th} completion time (t_i^c).

6.3 Metrics

Many real-time systems are also control systems. In control theory, sampling and actuation are expected to happen periodically at highly deterministic times [Åström and Wittenmark, 1997]. In practice, timing predictability of sampling and actuation are degraded, respectively because of the jitter in the release times of the control task (*release jitter*), and the variations in its *computation time*s. Timing unpredictability leads to control performance degradation and even instability [Marti et al., 2001]. Therefore, *release time* and *computation time* are two critical metrics to evaluate the performance of an implementation of a real-time task. We explain these in more detail now.

6.3.1 Measurement Method

Like other benchmarks in the CD_{x} family, CD_{rr} has a periodic real-time task named CD. To evaluate the performance of CD, we use *release jitter* and *computation time* as our metrics (the same metrics as proposed by Kalibera et al. [2009]). To explain how we measure these metrics, we use the notations in Figure 6.1 that show the important milestones in the lifetime of a periodic task.

Release jitter is the amount of fluctuation in the actual release times of the CD task.

Ideally, all task instances are released at their designated release times:

$$\forall i \in \{1..\text{total number of release}\} : t_i^r = t_i^s$$

Where t_i^r is the expected time of the $\mathrm{i_{th}}$ release of the task, and t_i^s is the actual start time. In practice, the release times of a periodic task like CD vary due to factors such as system timer accuracy, RTOS service overheads (e.g. scheduling), and the language VM overheads (e.g. GC). Hence, the release delay (D_i) is always non-negative:

$$\forall i \in \{1..\text{total number of release}\} : D_i = t_i^r - t_i^s \geq 0$$

Yet, it may still be tolerable as long as the release delay is predictable. So, we measure *(absolute) release jitter* as the difference between the maximum and the minimum measured release delays [Buttazzo and Cervin, 2007], because it shows the domain within which the delay changes:

$$\text{release jitter} = max(D_i) - min(D_i)$$

Computation time is the total time a task is actively being run. For a single task application like CD, the *computation time* of the i_{th} release of the task (C_i) is the time between the actual start time of the task (t_i^s) and its completion (t_i^c):

$$C_i = t_i^c - t_i^s$$

In practice, preemptions may occur between the start and completion times, for various reasons such as background RTOS services and collecting garbage by the language VM. In real-time systems, RTOSs and language VMs are expected to put tight bounds on preemptions unless they are servicing the current task. In addition to RTOS and the language runtime, processors introduce unpredictability (e.g., due to cache hit/miss, branch mis/prediction) to the computation time too.

6.4 Results

In this section, we compare the computation times and release jitter of the CD task, with both COL and $\mathrm{NOI_{nn}}$ workloads, on RT-RPython, JamaicaVM-8.3 and Hotspot-11. For each combination of VMs and workloads, the reported results are obtained from repeating the benchmark 100 times. This means the CD task is run 1 000 000 times with the COL workload, and 2 000 000 times with the $\mathrm{NOI_{nn}}$ workload.

6.4.1 Computation Time

COL In the COL workload, the CD task is released periodically every 10 ms. Since CD's deadline and period are equal, the task must finish 10 ms after its expected release time in order to meet its deadline.

Figure 6.2 shows the distributions of the computation times for the COL workload.

In this workload, the average-case computation time (ACCT) for RT-RPython is 11.0 % less than JamaicaVM (1.206 ms vs. 1.355 ms), its 99.9-percentile is 8.9 % less (2.066 ms vs. 2.341 ms), and its worst-case computation time (WCCT) is 72.5 % less (3.157 ms vs. 11.486 ms).

The better performance of RT-RPython in this test may partly be related to the additional features provided by JamaicaVM, such as the real-time GC. Although we limit our benchmarks to use the features available in RT-RPython (most importantly, the garbage collected heap is not used), some of the unused features may impose static overhead. For instance, even if the application does not use the heap, the compiler may still emit some GC code (e.g. read/write barriers). This emphasizes the importance of the *configurability* demanded by the RTMu specification.

Finally, Java has the lowest ACCT, as expected from one of the most extensively optimized general-purpose managed languages. It also has the highest WCCT, mainly because of the long pauses caused by its GC. It is worth mentioning that the JVM used in our tests is configured to use ZGC which is a low-latency GC designed to limit pauses to under 10 ms [Oracle, 2018] (our tests reported in Appendix Figure 2 confirm ZGC as the most predictable GC in Hotspot JVM-11). On the other hand, a real-time managed language avoids such pauses at the cost of ACCT.

NOI$_{\mathbf{nn}}$ Figure 6.3 shows the distributions of the computation times for the NOI$_{\mathrm{nn}}$ workload. In this workload, the ACCT for RT-RPython is 6.8 % higher than JamaicaVM (0.582 ms vs. 0.545 ms) and its 99.9-percentile is 10.9 % higher (1.024 ms vs. 0.923 ms), while its WCCT is 38.4 % lower (1.540 ms vs. 2.501 ms).

We argue that the slowdown in ACCT is acceptable mainly for these reasons:

1. RTZebu's compiler backend relinquishes some optimizations in favour of minimality.

2. RT-RPython's write barriers may considerably benefit from further optimizations. The ACCT of ucRT-RPython (RT-RPython without write barriers) in Figure 6.4 reflects the relative immaturity of RT-RPython compared to JamaicaVM, in particular: it shows an upper-bound on the potential improvement from the reference check optimization.

3. RT-RPython's RTMu IR generation still needs further optimizations.

4. RTZebu's compiler backend (mostly borrowed from Zebu) is under-developed, and would benefit from more engineering and research effort (Lin [2019] enumerates some potential improvements).

Regarding the WCCT, the same argument from the COL workload still applies here. Also, Java has the best ACCT and worst WCCT, just as in the COL workload.

6.4.2 Release Jitter

As explained in Section 6.3.1, release jitter is defined as the difference between the highest and the lowest release delays. Figures 6.5 and 6.6 show the distributions of

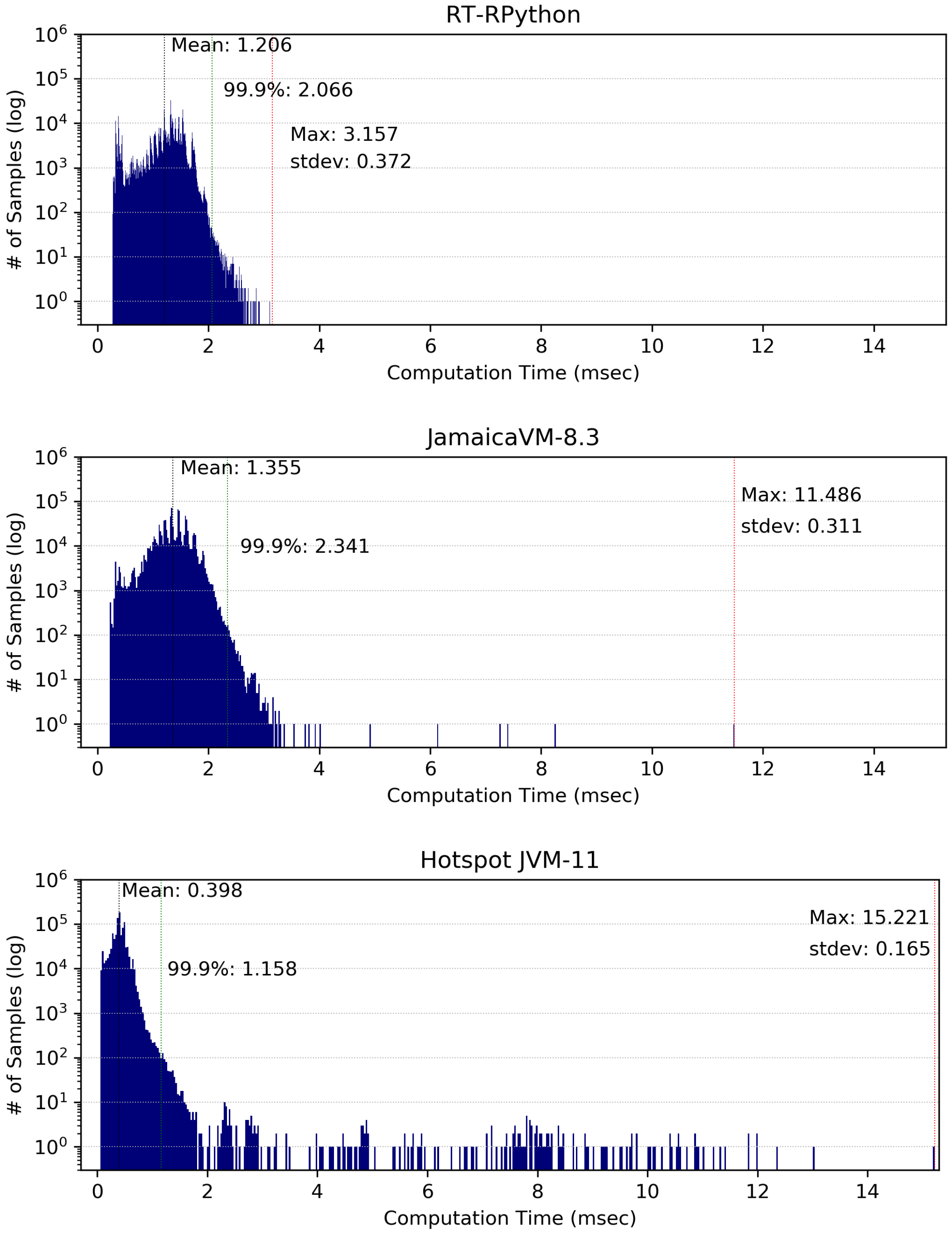


Figure 6.2: Computation times for the COL workload. RT-RPython outperforms JamaicaVM in both average-case computation time (1.206 ms vs. 1.355 ms) and worst-case computation time (3.157 ms vs. 11.486 ms). Also, both real-time VMs (RT-RPython and JamaicaVM) perform significantly slower than the non-real-time VM (Hotspot JVM-11) in the average-case, while they achieve better worst-case computation times.

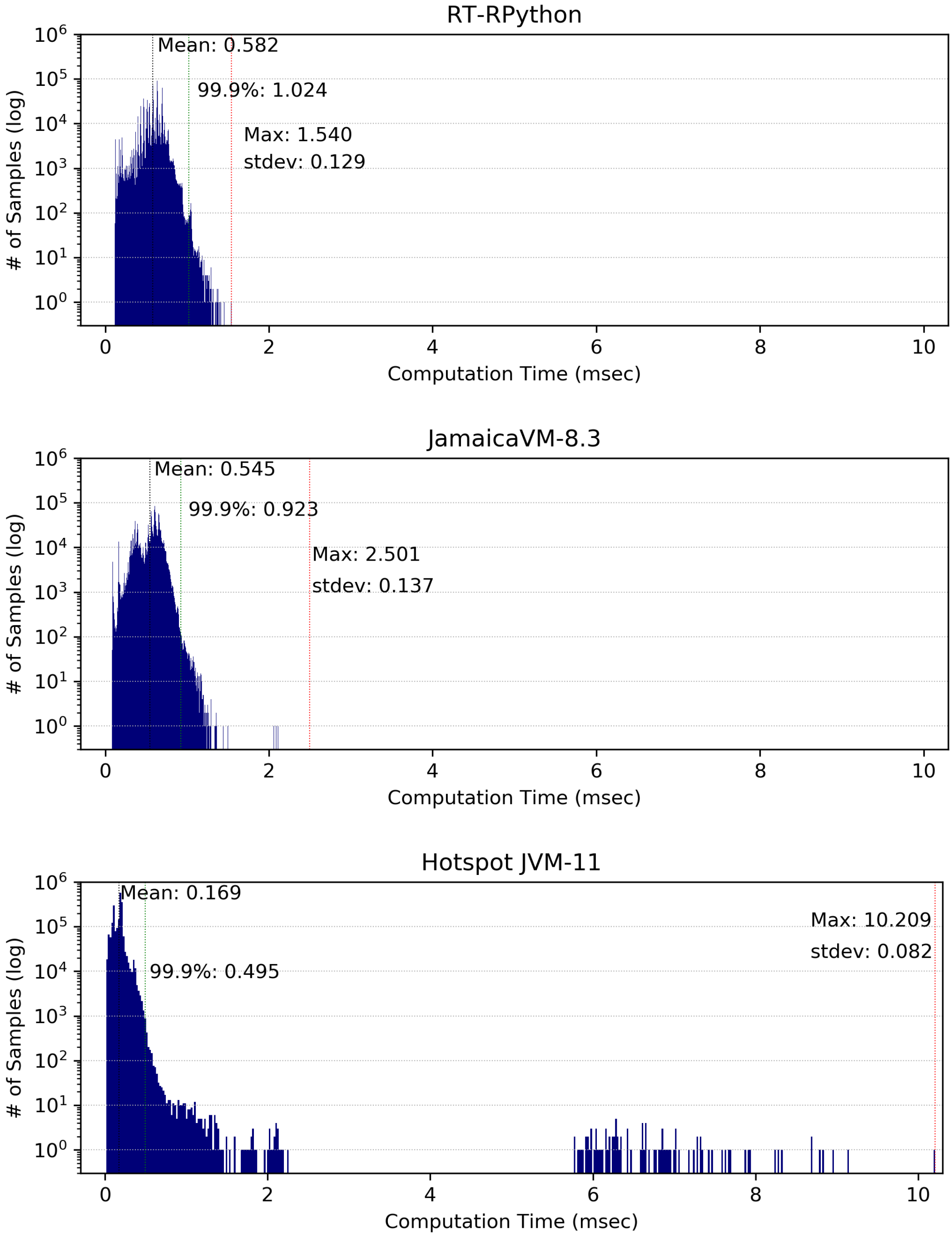


Figure 6.3: Computation times for the NOI_{nn} workload. RT-RPython outperforms JamaicaVM in the worst-case computation time (1.540 ms vs. 2.501 ms), and JamaicaVM achieves a better average-case computation time (0.545 ms vs. 0.582 ms). Also, both real-time VMs (RT-RPython and JamaicaVM) perform significantly slower than the non-real-time VM (Hotspot JVM-11) in the average-case, while they achieve better worst-case computation times.

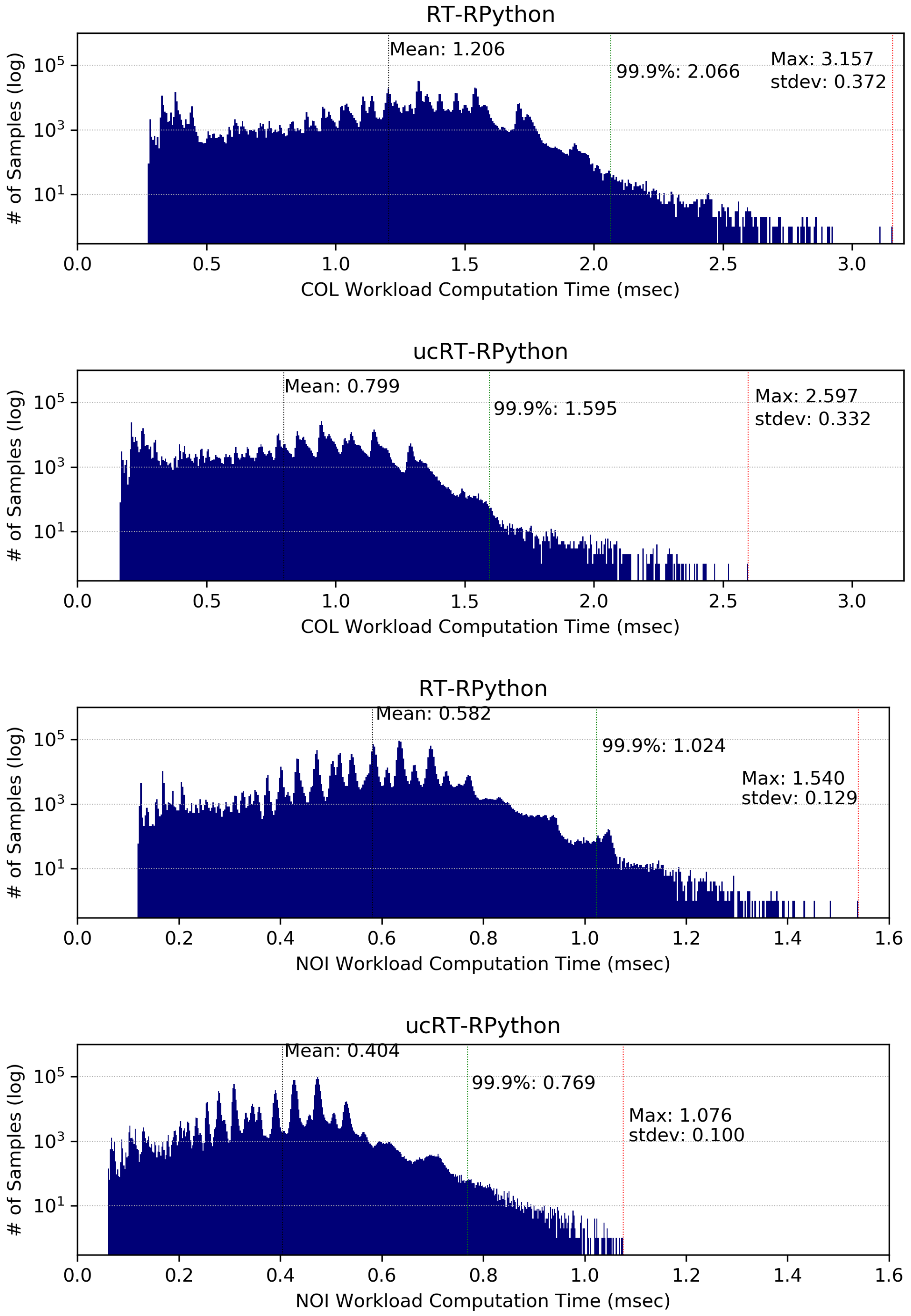


Figure 6.4: Overhead of the RT-RPython reference write barriers. RT-RPython emits write barriers on all RTMu store operations where the source operand is of reference type. The write barrier checks the memory area for the source and destination operands, and throws an exception if the destination has a longer lifetime, because it leads to dangling references to objects in scopes. ucRT-RPython is an implementation of RT-RPython that does not emit write barriers to check reference lifetimes. Comparing the computation times of ucRT-RPython to the default RT-RPython shows that optimizing our trivial implementation of the reference write barriers should significantly improve both the average- and worst-case computation times of the benchmarks.

the release delays for the COL and $\mathrm{NOI}_{\mathrm{nn}}$ workloads where the periods of the CD task are 10 ms and 4 ms respectively. According to the figures, RT-RPython outperforms JamaicaVM in the release jitter in both workloads (1.250 ms vs. 9.965 ms in COL, and 1.113 ms vs. 3.999 ms in $\mathrm{NOI}_{\mathrm{nn}}$). This demonstrates the higher predictability of RT-RPython in creating periodic releases, compared to JamaicaVM.

An unexpected point seen in Figures 6.5 and 6.6 is that the release jitter for JamaicaVM (a real-time language VM) is very close to Hotspot JVM (a non-real-time language VM) in both workloads (9.965 ms vs. 9.997 ms in COL, and 3.999 ms vs. 4.000 ms in $\mathrm{NOI}_{\mathrm{nn}}$). However, considering the mean and standard deviation values, JamaicaVM rarely creates delayed releases, while for Hotspot JVM, release delays are more evenly distributed throughout a period. This demonstrates that JamaicaVM is more predictable in creating periodic releases compared to the Hotspot JVM.

As discussed in Section 6.4.1, the overhead of JamaicaVM's unused features, especially the real-time GC, may partly be responsible for the unpredictability in JamaicaVM's timing behaviour. This holds true for the higher release jitter of JamaicaVM compared to RT-RPython. To check this, we modified the periodic task (CD) to have an empty body, and left the rest of the benchmark unchanged. This ensures that the benchmark does not require any of the JamaicaVM's runtime features (e.g. GC). We call the modified benchmark NOP. Figure 6.7 compares the release delay results from running NOP with the COL workload to the original benchmark with the same workload. The comparison does not show any significant change to the release jitter of the periodic task. This confirms that the jitter is not caused by the features used by the original benchmark, with the exception of clock and timer which are also used by NOP. We conducted experiments to determine the role of the clock and timer in this jitter.

Clock To assess the effect of clock on the release jitter of JamaicaVM, we measured the computation times of NOP. The main body of NOP consists of getting the start time (actual release time) and the completion time, without any other instructions in between. So, it only includes the clock related part of the benchmark. According to Figure 6.8, the maximum computation time for NOP is 0.0806 ms. This is a negligible amount of time compared to the reported release delays. Hence, the potential inaccuracy of JamaicaVM's clock is not a determinant factor. This applies to all other performance metrics reported for JamaicaVM in this thesis.

Timer To check whether JamaicaVM's implementation of `waitForNextPeriod()` is causing the high release jitter, we employed tracing tools (e.g. strace and ftrace in Linux) to extract the underlying timer-related system calls or library calls used by JamaicaVM. We found `pthread_cond_timedwait()` (timed wait on a POSIX condition variable) to be the underlying library call. Thus, we reimplemented RT-RPython's `wait_for_next_period()` function (equivalent to `waitForNextPeriod()` in RTSJ) using the `pthread_cond_timedwait()` function and compared it to the default implementation using the `nanosleep()` function. According to the results in Figure 6.9, the two RT-RPython implementations achieve similar results which indicates that the high

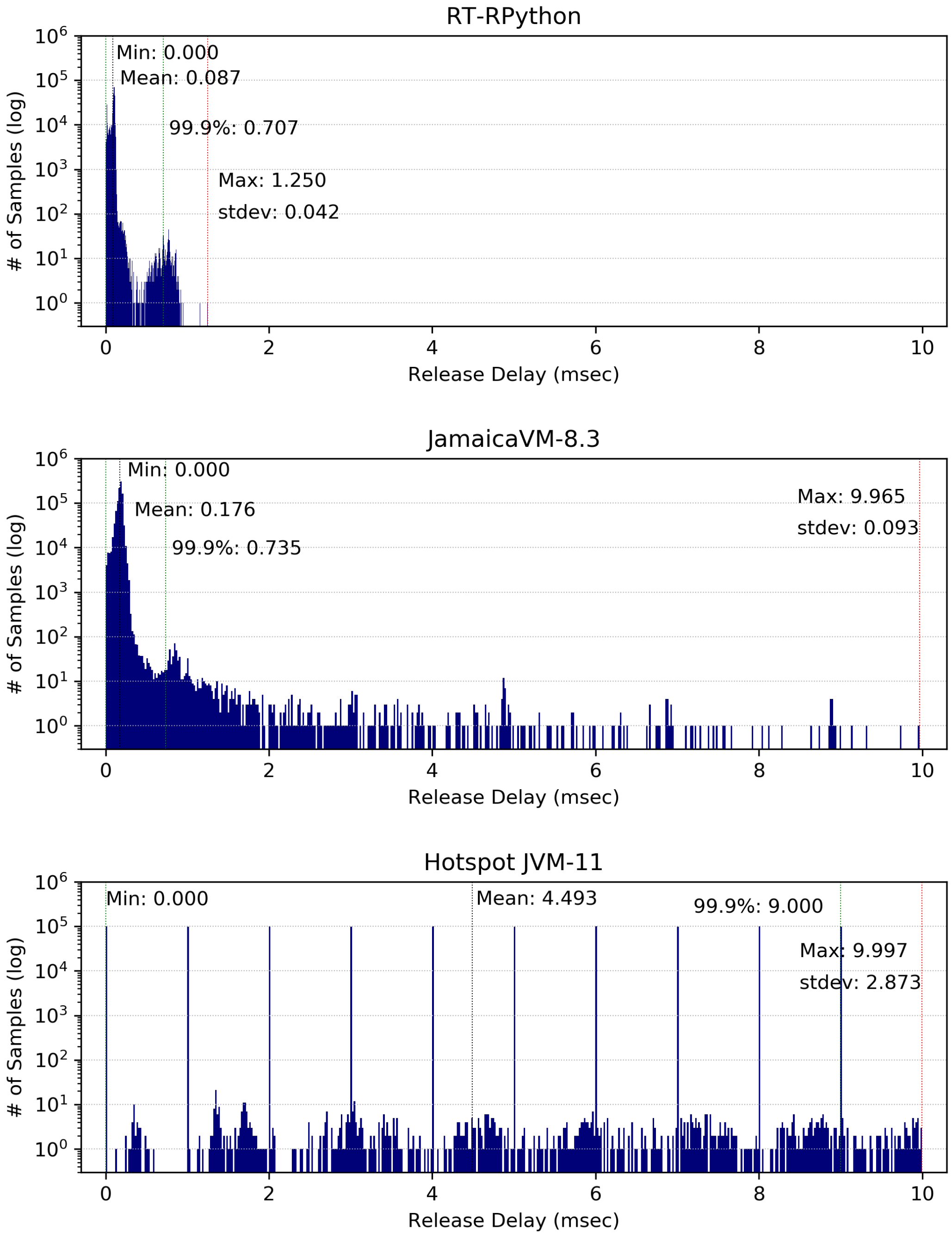


Figure 6.5: Release jitter for the COL workload (Period=10 ms). Because the minimum release delays for all VMs in the figure are zero, their release jitter is equal to their maximum release delay. RT-RPython outperforms JamaicaVM in release jitter (1.250 ms vs. 9.965 ms). It also performs significantly more predictably than the Hotspot JVM. Although the release jitter for JamaicaVM is very close to Hotspot, the mean and standard deviation values of the diagrams show that JamaicaVM rarely creates delayed releases, while for Hotspot, release delays are almost evenly distributed throughout a period. This diagram demonstrates the higher predictability of RT-RPython in creating periodic releases, compared to JamaicaVM. It also shows how unpredictable a non-real-time VM can be on the same measure.

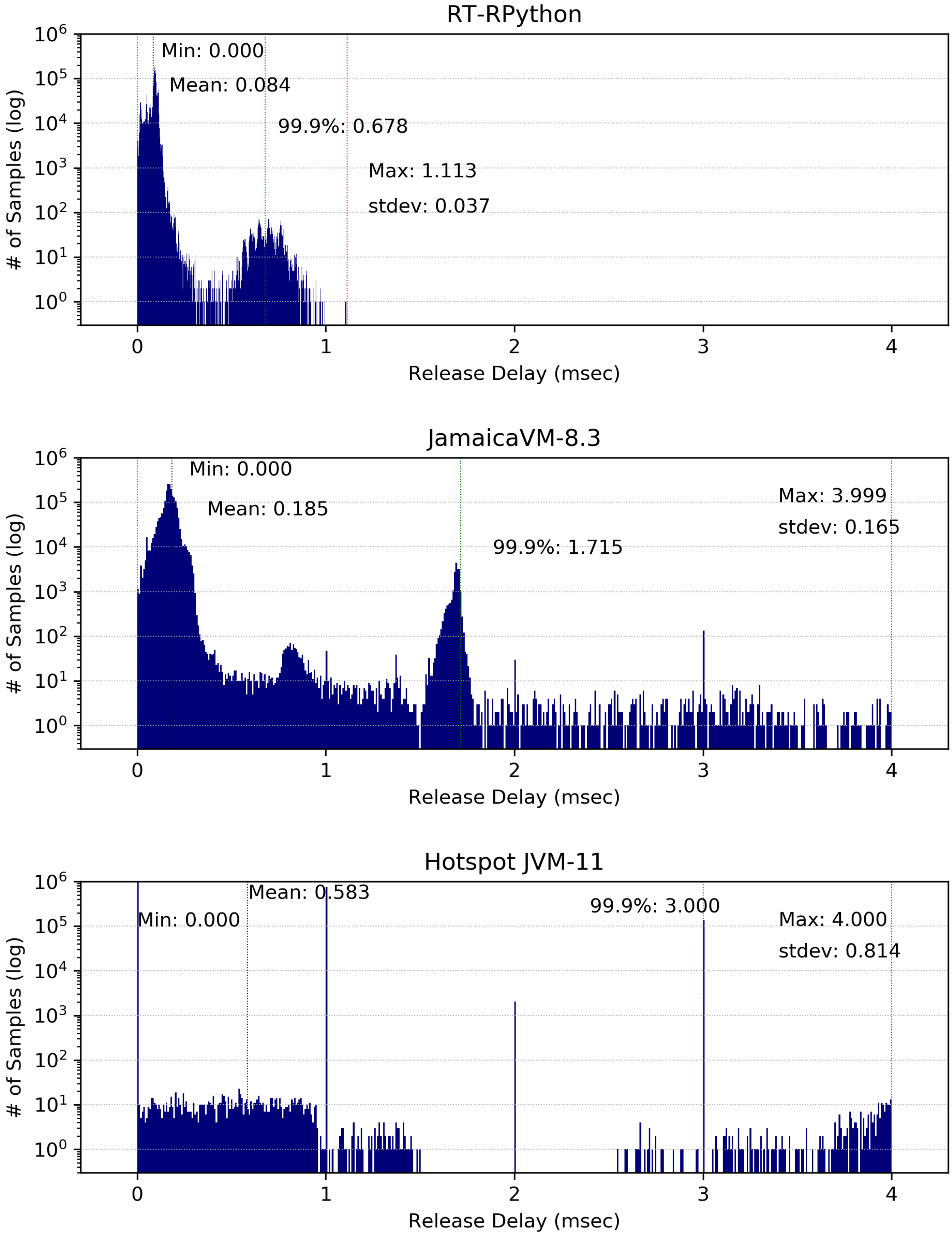


Figure 6.6: Release jitter for the NOI_{nn} workload (Period=4 ms). Because the minimum release delays for all VMs in the figure are zero, their release jitter is equal to their maximum release delay. RT-RPython outperforms JamaicaVM in release jitter (1.113 ms vs. 3.999 ms). It also performs significantly more predictably than the Hotspot JVM. Although the release jitter for JamaicaVM is very close to the Hotspot JVM, the mean and standard deviation values of the diagrams show that JamaicaVM rarely creates delayed releases, while for Hotspot, release delays are distributed throughout a period. Confirming Figure 6.5, this diagram demonstrates the higher predictability of RT-RPython in creating periodic releases, compared to JamaicaVM and the Hotspot JVM.

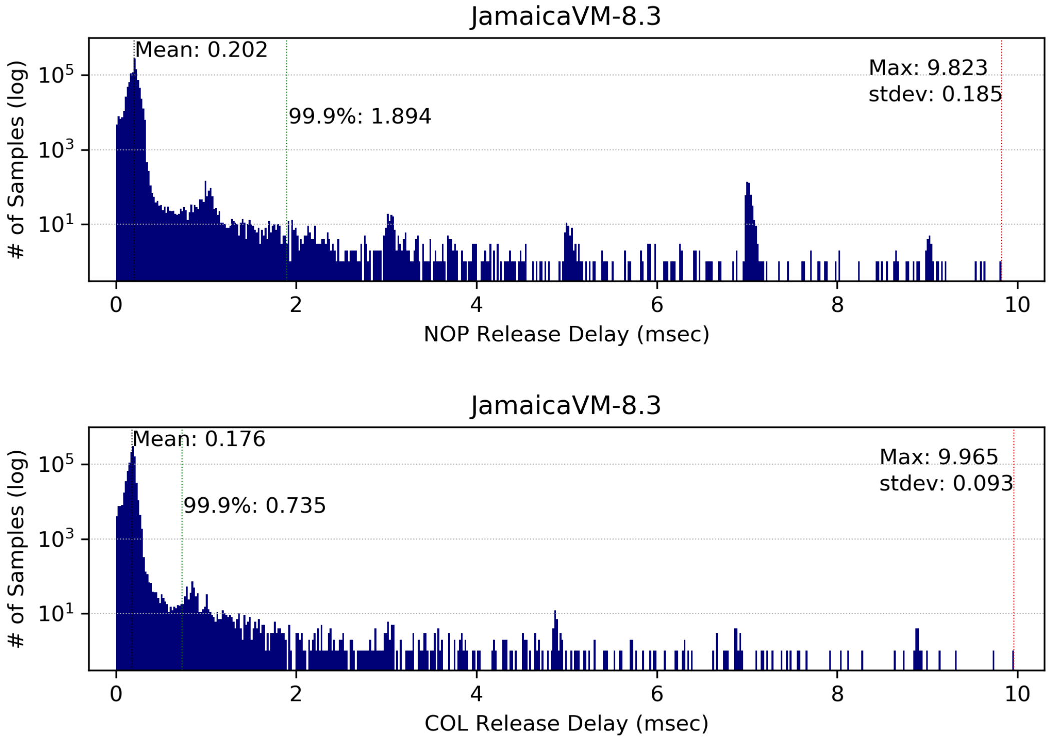


Figure 6.7: Comparison of the release delays in the COL workload to an empty workload (NOP) on JamaicaVM. Although the release jitter of NOP is slightly better (lower) than COL, its standard deviation and the 99.9 percentile are worse (higher). This indicates that the high release jitter of JamaicaVM is not caused by the features used by COL, with the exception of clock and timer which are also used by NOP.

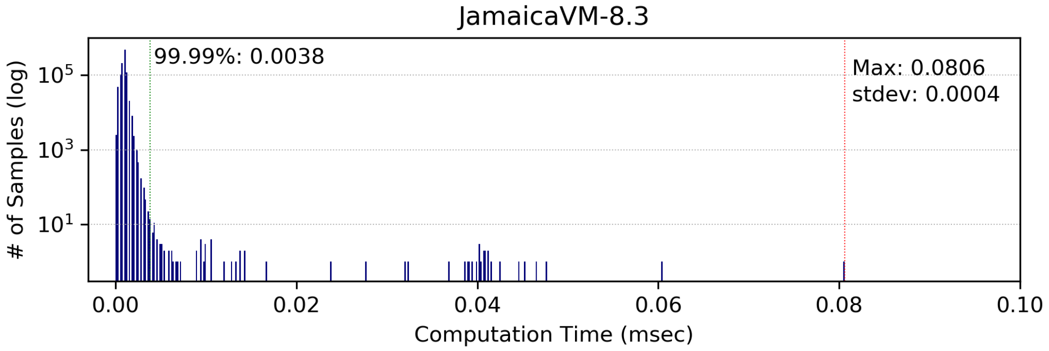


Figure 6.8: Computation time of an empty workload (NOP). The computation time of the NOP workload shows the delay that JamaicaVM's clock imposes on all time measurements. The maximum delay is 80.6 µs which is negligible compared to JamaicaVM's release jitter. This demonstrates that the inaccuracy of JamaicaVM's clock is not an effective element in the benchmark's metrics.

release jitter of JamaicaVM is not a result of how the `waitForNextPeriod()` function is implemented in JamaicaVM.

The above experiments demonstrate that the high release jitter of JamaicaVM is not caused by clock measurement errors, or the inaccuracy of the underlying RTOS timer facility.

6.4.3 Release Miss Rate

As shown in Figure 6.1, the i_{th} release of a periodic task has a time window which starts with its expected release time (t_i^r) and ends with its deadline which is often equal to the $i + 1_{\text{th}}$ expected release time of the periodic task ($t_{\text{i+1}}^r$). If the release is not created in this time window, it is called a *missed release*.

Table 6.2 shows that RT-RPython is the only VM that does not miss any releases in any of the workloads. Next is JamaicaVM that misses 0.0023 % of the releases in the COL workload and 0.0069 % in the NOI_{nn}. Finally, the Hotspot JVM misses 10.36 % of releases in the COL workload and 8.53 % in the NOI. Hence, the release miss ratio of the Hotspot JVM is 4504 times higher than JamaicaVM in the COL workload, and 1236 times higher in the NOI_{nn} workload. This confirms our argument in Section 6.4.2, that JamaicaVM is dramatically more predictable in creating periodic releases compared to the Hotspot JVM.

Inter-Release Time

When there are release misses in a VM, reporting its release delays/jitter can often be misleading, because it ignores critical parts of the periodic task's timeline. Figure 6.10 depicts three execution scenarios for a periodic task with a period of 4 ms. The scenario with no missed release has the highest release jitter, followed by a scenario with one missed release, and the best release jitter belongs to the scenario where two releases are missed. Here, reporting only the release jitter is misleading, because it overlooks the frequency or pattern of missing releases.

Therefore, as an alternative to reporting release delays, and to complement it, we report the *inter-release time*s of all VMs in Figure 6.11. Compared to release delays, inter-release times do not ignore the missed releases. They also reflect how these misses are distributed throughout the benchmark execution. For instance, an execution with two consecutive release misses is depicted differently from another execution with two release misses that happen separately, because they will lead to different inter-release times.

The inter-release times in Figure 6.11 uncover a new critical point in our VM comparison: that the Hotspot JVM is the only VM in which the average inter-release time is not equal to the period. This means the periodic releases generated by Hotspot JVM are not accurate even in the average-case. Additionally, while inter-release times in JamaicaVM are distributed over a wider range compared to Hotspot JVM, the 0.1 and 99.9 percentiles in JamaicaVM show that it creates a significant portion of releases close to their expected release times. Finally, the result confirms that periodic task

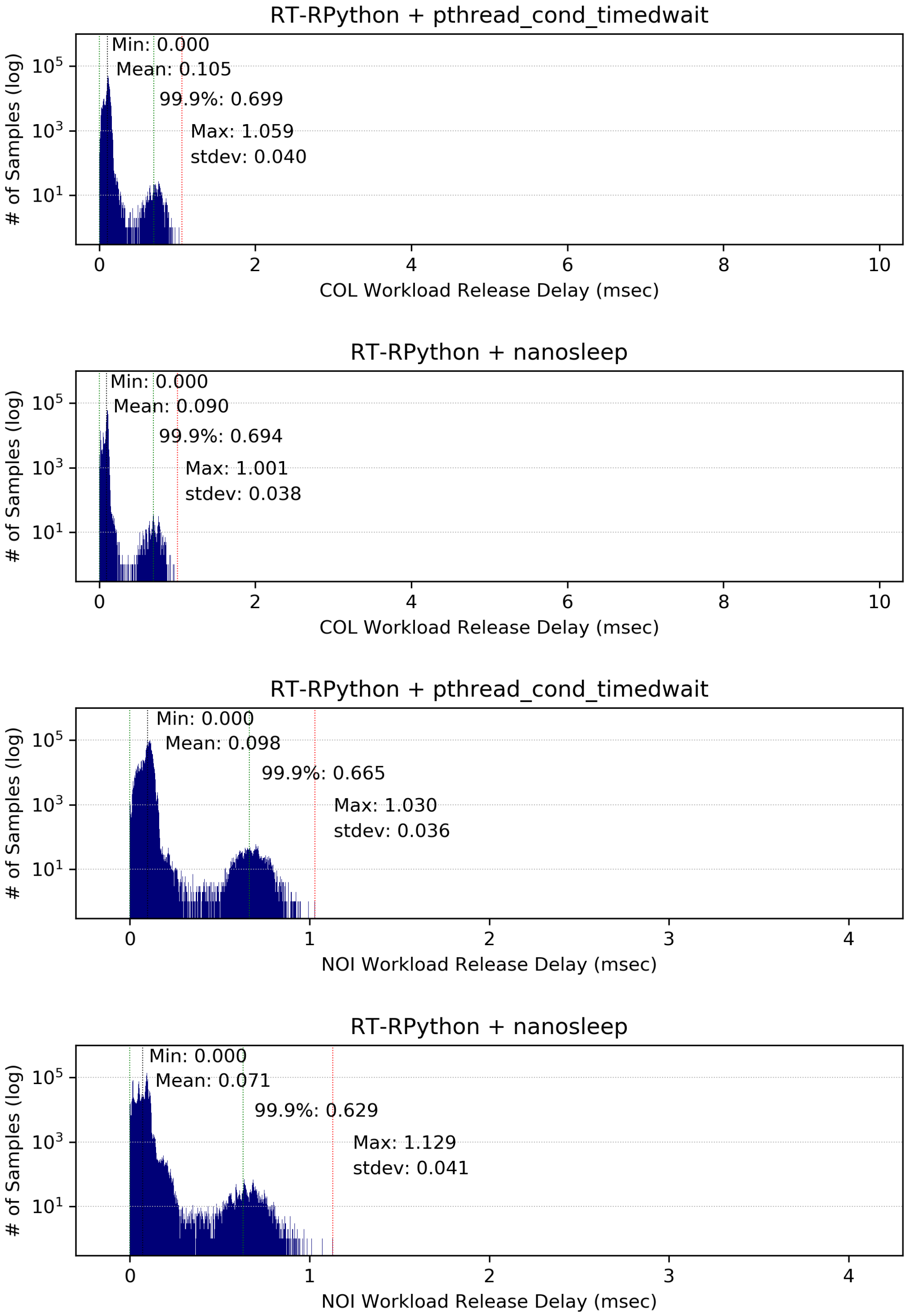


Figure 6.9: Implementing the `waitForNextPeriod()` function as a timed wait operation on a POSIX condition variable (JamaicaVM's approach), or a `nanosleep()` function call (RT-RPython's default approach), lead to very similar periodic release delays in both workload. This suggests that the high release jitter of JamaicaVM is not a result of how the `waitForNextPeriod()` function is implemented in JamaicaVM.

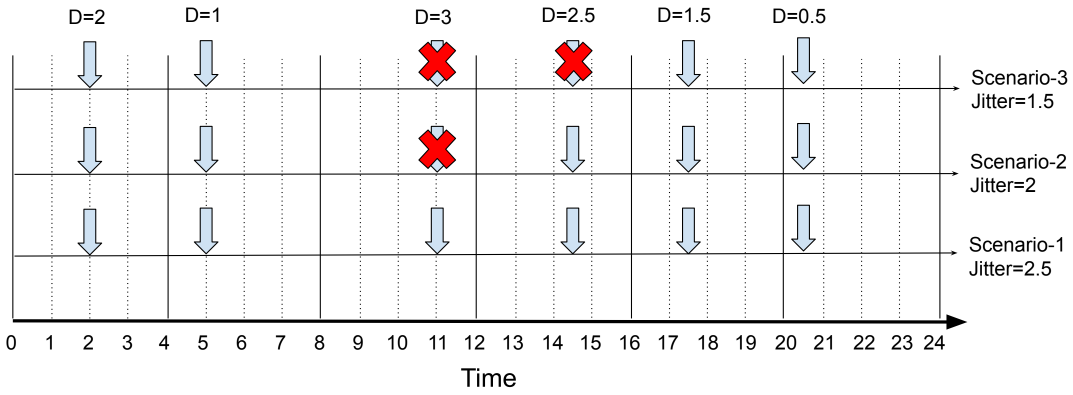


Figure 6.10: This figure shows three scenarios for a periodic task that starts at time zero, and has a period of 4 ms. The release jitter for scenario-1 where the periodic task doesn't miss any releases is 2.5 ms which is higher than the other two scenarios. In such cases where there are missed releases, release jitter does not reflect the predictability of releases, because it overlooks critical parts of the information. For instance, the period from 8 ms to 12 ms in scenario-2, and the period from 8 ms to 16 ms in scenario-3 are not reflected in release jitter. Thus, release jitter may be misleading in the presence of release misses.

Table 6.2: Release Miss Rate on the Tested VMs. RT-RPython is the only VM that does not miss any releases on any of the workloads. JamaicaVM misses 23 out of 1 000 000 releases in the COL workload, which has a period of 10 ms. The number of misses rises to 69 per 1 000 000 releases in the NOI$_{nn}$ workload which has a smaller period of 4 ms. The release miss ratio for Hotspot JVM is 4504 times higher than JamaicaVM in the COL workload, and 1236 times higher in the NOI$_{nn}$ workload.

	COL	NOI$_{nn}$
RT-RPython	0 %	0 %
JamaicaVM-8.3	0.0023 %	0.0069 %
Hotspot JVM-11	10.36 %	8.53 %

releases are more predictable in RT-RPython, compared to the other VMs.

6.5 Summary

In this chapter, we presented our evaluation of RT-RPython as a real-time programming language. We chose an existing real-time application benchmark suite targeting Java and RTSJ, and explained its implementation in RT-RPython. Then, we compared the performance results of RT-RPython to JamaicaVM (as an RTSJ implementation) and Hotspot (a highly-optimized non-real-time language VM). We used computation time and release jitter as the measurement metrics. Although RT-RPython does not provide all of JamaicaVM's features (e.g. real-time GC), we tried to keep the comparison as fair as possible by only using their common features in our tests (e.g. the benchmark does not allocate to the garbage collected heap).

A.6.2-aknowledgement of limitation

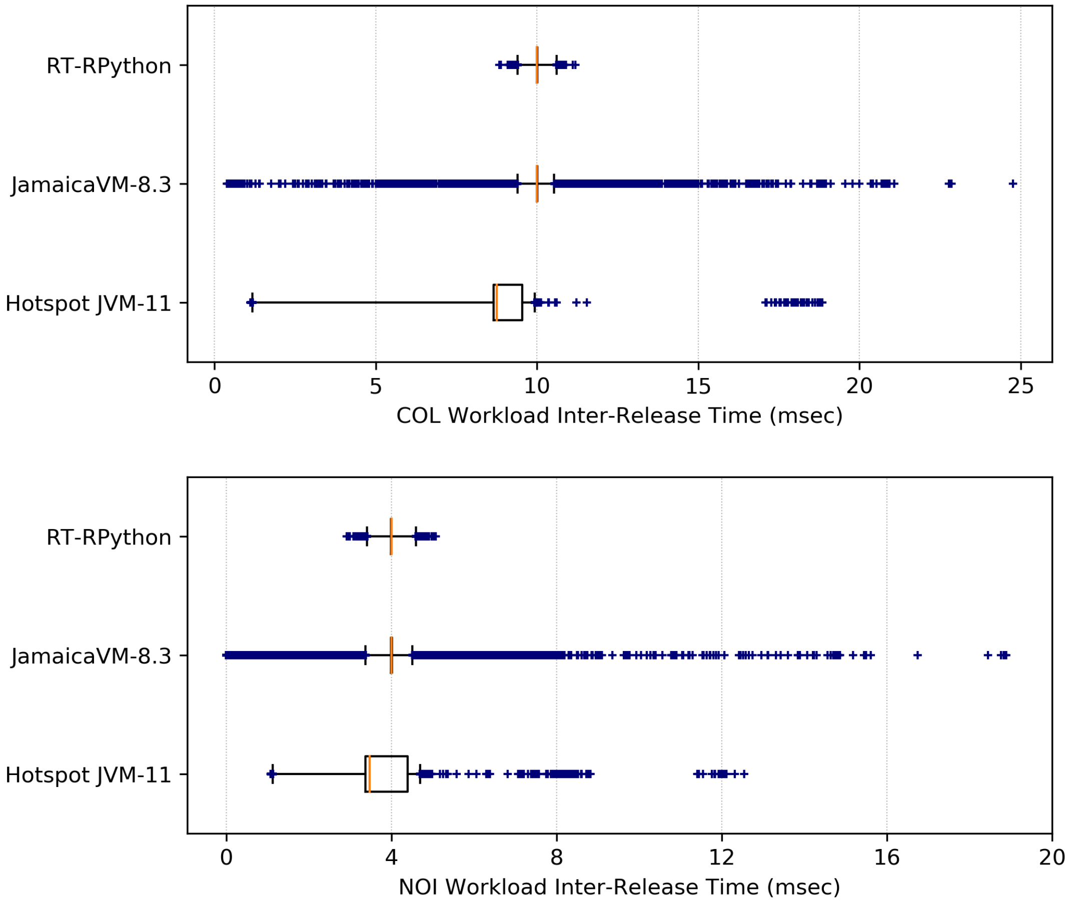


Figure 6.11: Inter-release times for COL and $\mathrm{NOI}_{\mathrm{nn}}$ workloads (outliers $\notin$ [0.1%, 99.9%]). The CD task's period is 10 ms in COL and 4 ms in $\mathrm{NOI}_{\mathrm{nn}}$. Ideally, the time between two subsequent releases (inter-release time) of CD should always be 10 ms and 4 ms respectively. In practice, various elements including the programming language runtime add unpredictability to task release times. Comparing the maximum and the standard deviation of inter-release time for RT-RPython to JamaicaVM in both workloads shows that RT-RPython is adding significantly less unpredictability. Despite the larger maximum inter-release time of JamaicaVM compared to Hotspot, its standard deviation is smaller, and its 0.1 percentile is much closer to the period, which shows periodic releases are created with less variation in JamaicaVM. Finally, Hotspot is the only VM in which the average inter-release time is not equal to the period. This means periods generated by Hotspot are not accurate on average.

Our measurements show that RT-RPython outperforms JamaicaVM in most test scenarios, including the average and worst-case computation time, release jitter, and release miss rate. The only exception is the average-computation time on the $\mathrm{NOI}_{\mathrm{nn}}$ workload where JamaicaVM slightly outperforms RT-RPython. By comparing the performance results of RT-RPython to JamaicaVM (as a commercial real-time language VM), we do not aim to prove RT-RPython's high efficiency. Rather, we believe that these results for RT-RPython on aircraft collision detection workloads demonstrate that RTMu, as a real-time µVM, can provide an *efficient foundation* for the development of programming languages suitable for building real-time software.

Chapter 7

Conclusion

Real-time systems have grown considerably in both diversity and popularity, and the demand for real-time software has never been higher. In contrast, the choice of programming languages used to develop these systems has mostly remained limited to decades-old languages, namely Ada and C/C++, and more recently real-time Java. This seems surprising given the diversity and popularity of real-time systems, and the flourishing ecosystem of general-purpose languages. In fact, many programming languages were developed or adapted to build real-time software, but only a few survived. This suggests the need for diverse programming languages in real-time systems.

We postulate that the main reason for this monoculture is the difficulty of developing new programming languages for real-time systems, due to their strict correctness requirements. Therefore, we propose using a new µVM to relieve the difficulty and reduce the cost of implementing new languages for real-time systems. A µVM is minimal and low-level, which allows supporting a wide range of languages for diverse real-time systems. Its minimality also makes correct implementation and formal verification of the platform easier, which is vital for many real-time systems.

So, we try to answer the following question in this thesis: *'What is a suitable µVM design for real-time systems, and how can we verify its suitability?'*

As an answer to the first part of the question, we design RTMu, the first µVM instance targeting programming languages for real-time systems. To arrive at this design, we studied an extensive range of programming languages built or adapted for real-time systems, and extracted a set of key features that distinguish real-time and non-real-time languages. With an existing µVM specification, named Mu, as the starting point, and the goal of supporting the implementation of these key features, we propose a set of modifications to Mu's abstractions over concurrency and memory management, and reuse its compiler backend. Given the already-demonstrated capability of Mu in implementing real-world managed languages, we argue that these changes make RTMu capable of implementing new real-time languages or reimplementing existing ones.

To confirm the feasibility of the RTMu's abstractions, we build a performant implementation of the RTMu specification, named RTZebu, based on Zebu, a performant implementation of Mu. We use Rust, a thread-, memory-, and type-safe language to implement RTZebu, and discuss some interesting points in implementing a language

VM (or other system software) for real-time systems in a safe language.

To answer the second part of the question, *'how can we verify its suitability?'*, we demonstrate that RTMu *'can provide an efficient and usable foundation for the development of programming languages suitable for building real-time software'*.

We design a real-time programming language, RT-RPython as a real-time extension to RPython, a fully static dialect of Python. To demonstrate the *usability* of RTMu, we implement RT-RPython on top of RTMu. We also reimplement the Collision Detection benchmark suite, a real-time application benchmark originally targeting Java and RTSJ VMs, in RT-RPython to show that RT-RPython is *'suitable for building real-time software'*.

Finally, we demonstrate RTZebu's *efficiency* as a foundation for implementing real-time programming languages. We run the collision detection benchmark on RT-RPython and compare its real-time performance to JamaicaVM (a commercial implementation of RTSJ). RT-RPython outperforms JamaicaVM in both the average- and worst-case for all performance metrics, with one exception where JamaicaVM performs marginally better. We conclude that RT-RPython is efficient, and the occasional lack of efficiency is not inherent to the RTMu design, rather a sign that more research and engineering effort is required.

In summary, this thesis presents our design and implementation of RTMu, a µVM that acts a proof of concept, establishing the use of µVMs to build new high-quality programming languages for real-time systems. It also provides an empirical demonstration of performance and predictability for µVMs in the real-time domain. The source code developed as part of this thesis are open-sourced under the Apache License, Version 2.0. We believe that RTMu can help in tackling the current lack of diversity in programming languages for real-time systems.

7.1 Future Work

This thesis and its products, including the RTMu specification (Chapter 3), RTZebu (Chapter 4), and RT-RPython (Chapter 5) may be followed by a wide range of research topics, from low-level compiler backend optimizations to design and implementation of new real-time languages. Here, we only mention some of the topics we were most concerned about throughout doing this thesis.

7.1.1 A Real-Time Garbage Collector

It has been an explicit non-goal of this thesis to design and implement a real-time garbage collector (RTGC) for RTZebu. However, RTGC is an essential part of the RTMu specification.

There has been much work on designing RTGCs that ensure short *pauses* and predictable *minimum mutator utilization*, and much progress has been made [Jones et al., 2011]. RTGCs are already available in RTSJ implementations such as JamaicaVM [AicasWebPage] and FijiVM [Pizlo et al., 2009, 2010b]. Thus, while design and implementation of a RTGC for RTZebu may benefit a rich literature, there are still interesting challenges when we take RTMu's requirements (as mentioned in Section 3.3.4) into account.

7.1.2 Integrated WCET Analysis

Schedulability analysis is an essential part of designing a hard real-time system, and WCET analysis of the tasks in the system is an integral component of schedulability analysis. The literature on real-time systems offers an abundance of research on estimating the WCET of time-critical tasks, and a number of commercial WCET analysis tools are also available [Wilhelm et al., 2008].

Supporting WCET analysis at the RTMu IR level has the major benefit of being portable which will hugely simplify supporting WCET analysis of applications in all of RTMu's client languages. Related work such as [Bernat et al., 2000; Frost et al., 2011] has already been done on WCET analysis of Java bytecode. Another major benefit of supporting WCET analysis at the RTMu IR level is that it may enable WCET-aware optimizations in the compiler backend. Here, a closely related work is the WCET-aware register allocation published by Falk et al. [2011] which reports significant improvements to the average- and worst-case performance of the tested tasks.

7.1.3 A Formally-Verified Implementation

Feasibility of building formally verified implementations is a key design goal for RTMu (and Mu), which makes it suitable for highly reliable systems. As stated in Section 2.1.4, many real-time systems are safety-critical and are required to undergo strict validation and certification processes. Hence, a formally verified implementation of RTMu is an appealing foundation for the development of programming languages for real-time systems, specially the safety-critical ones. It also helps widen the adaptation of RTMu in the real-time systems domain which will ultimately lead to improvement in the language diversity in this domain.

7.1.4 Optimizations

Following Mu's principle of minimality, the only optimizations performed by the compiler backend of RTZebu (Section 4.3.1) are register allocation and instruction selection. Other important optimizations are expected to be implemented at the RTMu's client level. While RTZebu is capable of achieving reasonable performance according to our evaluations (Chapter 6), it may still benefit further research and engineering effort, specially on the compiler backend. Besides, the RT-RPython client of RTZebu may significantly benefit optimizations to its RTMu IR generation. Hence, further studies on adding or refining optimizations in the RTZebu's compiler backend, and on top of it, as IR optimizer libraries, are both essential and effective in improving the performance of RTMu's clients.

Bibliography

The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. (cited on pages xv and 10)

Aicas, 2019. JamaicaVM 8.3-User Manual Java Technology for Critical Embedded Systems. Technical report. https://www.aicas.com/download/manuals/JamaicaVM-8.3-Manual.pdf. (cited on page 65)

AicasWebPage. JamaicaVM | aicas.com. https://www.aicas.com/wp/solutions/jamaicavm/. (cited on pages 12, 65, 66, and 84)

Amiri, J. E.; Blackburn, S. M.; Hosking, A. L.; and Norrish, M., 2019. Designing a low-level virtual machine for implementing real-time managed languages. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL)* (Athens, Greece, Oct. 2019). doi:10.1145/3358504.3361226. (cited on pages 21, 44, and 45)

Armbruster, A.; Baker, J.; Cunei, A.; Flack, C.; Holmes, D.; Pizlo, F.; Pla, E.; Prochazka, M.; and Vitek, J., 2007. A real-time Java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems*, 7, 1 (2007), 5:1–5:49. doi:10.1145/1324969.1324974. (cited on page 12)

Bacon, D. F.; Cheng, P.; and Rajan, V. T., 2003. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (New Orleans, Louisisana, USA, Jan. 2003). doi:10.1145/640128.604155. (cited on page 8)

Bambagini, M.; Marinoni, M.; Aydin, H.; and Buttazzo, G., 2016. Energy-aware scheduling for real-time systems: A survey. *ACM Trans. Embed. Comput. Syst.*, 15, 1 (Jan. 2016). doi:10.1145/2808231. https://doi.org/10.1145/2808231. (cited on page 7)

Barnes, J., 1997. *High integrity Ada: the SPARK approach*. Addison-Wesley Professional. (cited on page 12)

Bernat, G.; Burns, A.; and Wellings, A. J., 2000. Portable worst-case execution time analysis using Java byte code. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS)* (Stockholm, Sweden, Jun. 2000). IEEE Computer Society. doi:10.1109/EMRTS.2000.853995. (cited on page 85)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of* *the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Tucson, AZ, USA, Jun. 2008). doi:10.1145/1375581.1375586. (cited on page 40)

Boehm, H. and Adve, S. V., 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Tucson, AZ, USA, Jun. 2008). doi:10.1145/1375581.1375591. (cited on page 11)

Boehm, H.; Demers, A. J.; and Shenker, S., 1991. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Toronto, Ontario, Canada, Jun. 1991). doi:10.1145/113445.113459. (cited on page 40)

Bollella, G. and Gosling, J., 2000. The real-time specification for Java. *IEEE Computer*, 33, 6 (2000), 47–54. doi:10.1109/2.846318. (cited on page 3)

Bollella, G. and Gosling, J., 2000. The real-time specification for java. *Computer*, 33, 6 (2000), 47–54. doi:10.1109/2.846318. (cited on page 12)

Burns, A.; Dobbing, B.; and Romanski, G., 1998. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the Ada-Europe International Conference on Reliable Software Technologies*, vol. 1411 of *Lecture Notes in Computer Science* (Uppsala, Sweden, Jun. 1998), 263–275. Springer. doi:10.1007/BFb0055011. (cited on page 12)

Burns, A. and Wellings, A. J., 2009. *Real-Time Systems and Programming Languages - Ada, Real-Time Java and C / Real-Time POSIX, Fourth Edition*. International computer science series. Addison-Wesley. ISBN 978-0-321-41745-9. (cited on pages 1, 5, 8, 32, and 36)

Buttazzo, G. and Cervin, A., 2007. Comparative assessment and evaluation of jitter control methods (Loria, Nancy, France, Mar. 2007), 163–172. (cited on page 68)

Castaños, J. G.; Edelsohn, D.; Ishizaki, K.; Nagpurkar, P.; Nakatani, T.; Ogasawara, T.; and Wu, P., 2012. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Tucson, AZ, USA, Oct. 2012), 195–212. doi:10.1145/2384616.2384631. (cited on page 18)

Chang, Y. and Wellings, A. J., 2010. Garbage collection for flexible hard real-time systems. *IEEE Transactions on Computers*, 59, 8 (2010), 1063–1075. doi:10.1109/TC.2010.13. (cited on page 8)

Davis, R. I. and Burns, A., 2011a. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43, 4 (Oct. 2011). doi:10.1145/1978802.1978814. https://doi.org/10.1145/1978802.1978814. (cited on page 7)

Davis, R. I. and Burns, A., 2011b. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43, 4 (2011), 35:1–35:44. doi:10.1145/1978802.1978814. (cited on page 36)

Dolan, S.; Muralidharan, S.; and Gregg, D., 2013. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization*, 9, 4 (Jan. 2013), 36:1–36:25. doi:10.1145/2400682.2400695. (cited on page 17)

Doukas, G. S. and Thramboulidis, K., 2011. A real-time-Linux-based framework for model-driven engineering in control and automation. *IEEE Transactions on Industrial Electronics*, 58, 3 (2011), 914–924. doi:10.1109/TIE.2009.2029584. (cited on page 9)

EUROCAE, 2011. Software considerations in airborne systems and equipment certification: ED12C. Technical report. https://eshop.eurocae.net/eurocae-documents-and-reports/ed-12c/#. (cited on page 9)

Falk, H.; Schmitz, N.; and Schmoll, F., 2011. WCET-aware register allocation based on integer-linear programming. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)* (Porto, Portugal, Jul. 2011), 13–22. IEEE Computer Society. doi:10.1109/ECRTS.2011.10. (cited on page 85)

Frost, C.; Jensen, C. S.; Luckow, K. S.; and Thomsen, B., 2011. WCET analysis of Java bytecode featuring common execution environments. In *Proceedings of the 9th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)* (York, United Kingdom, Sep. 2011). doi:10.1145/2043910.2043916. (cited on page 85)

Henties, T.; Ag, S.; Hunt, J.; Locke, D.; Nilsen, K.; Na, A.; Schoeberl, M.; and Vitek, J., 2009. Java for safety-critical applications. *Electronic Notes in Theoretical Computer Science - ENTCS*, (01 2009). (cited on page 9)

IEEE and TheOpenGroup, 2018. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) - Redline*, (2018), 1–6900. (cited on page 14)

Jensen, E.; Locke, C.; and Tokuda, H., 1985. Time-driven scheduling model for real-time operating systems. In *Unknown Host Publication Title*, 112–122. IEEE. (cited on page 6)

Jibaja, I.; Blackburn, S. M.; Haghighat, M. R.; and McKinley, K. S., 2011. Deferred gratification: engineering for high performance garbage collection from the get go. In *Proceedings of the ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC)* (San Jose, CA, USA, Jun. 2011), 58–65. doi:10.1145/1988915.1988930. (cited on page 11)

Jones, R. E.; Hosking, A. L.; and Moss, J. E. B., 2011. *The Garbage Collection Handbook: The art of automatic memory management*. Chapman and Hall / CRC Applied

Algorithms and Data Structures Series. CRC Press. ISBN 978-1-4200-8279-1. (cited on page 84)

Kalibera, T.; Hagelberg, J.; Pizlo, F.; Plsek, A.; Titzer, B. L.; and Vitek, J., 2009. Cd$_X$: a family of real-time Java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, ACM International Conference Proceeding Series (Madrid, Spain, Sep. 2009), 41–50. ACM. doi:10.1145/1620405.1620412. (cited on pages ix, 61, 62, 66, and 67)

Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S., 2009. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09 (Big Sky, Montana, USA, 2009), 207–220. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1629575.1629596. https://doi.org/10.1145/1629575.1629596. (cited on page 39)

Lin, Y., 2019. *An efficient implementation of a micro virtual machine*. Ph.D. thesis, The Australian National University. doi:1885/158122. (cited on pages 2, 19, 39, 40, and 69)

Lin, Y.; Blackburn, S. M.; Hosking, A. L.; and Norrish, M., 2016. Rust as a language for high performance GC implementation. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)* (Santa Barbara, CA, USA, Jun. 2016). doi:10.1145/2926697.2926707. (cited on pages 40 and 41)

Lippiello, V.; Villani, L.; and Siciliano, B., 2007. An open architecture for sensory feedback control of a dual-arm industrial robotic cell. *Industrial Robot*, 34, 1 (2007), 46–53. doi:10.1108/01439910710718441. (cited on page 9)

Liu, C. L. and Layland, J. W., 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20, 1 (1973), 46–61. doi:10.1145/321738.321743. (cited on page 7)

Manson, J.; Pugh, W.; and Adve, S. V., 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Long Beach, CA, USA, Jan. 2005). doi:10.1145/1040305.1040336. (cited on page 11)

Marti, P.; Villa, R.; Fuertes, J. M.; and Fohle, G., 2001. On real-time control tasks schedulability. In *Proceedings of the 2001 European Control Conference (ECC)* (Porto, Portugal, Sep. 2001), 2227–2232. (cited on page 67)

McCarthy, J., 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3, 4 (1960), 184–195. doi:10.1145/367177.367199. (cited on page 11)

McCormick, J. W. and Chapin, P. C., 2015. *Building High Integrity Applications with SPARK*. Cambridge University Press. doi:10.1017/CBO9781139629294. (cited on page 12)

Mu, 2018. The specification of Mu. https://gitlab.anu.edu.au/mu/mu-spec. (cited on pages ix and 15)

Oracle, 2018. HotSpot Virtual Machine Garbage Collection Tuning Guide. https://docs.oracle.com/en/java/javase/11/gctuning/available-collectors.html{#}GUID-F215A508-9E58-40B4-90A5-74E29BF3BD3C. (cited on pages 66 and 69)

Pizlo, F.; Ziarek, L.; Blanton, E.; Maj, P.; and Vitek, J., 2010a. High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10 (Paris, France, 2010), 69–82. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1755913.1755922. https://doi.org/10.1145/1755913.1755922. (cited on page 2)

Pizlo, F.; Ziarek, L.; Maj, P.; Hosking, A. L.; Blanton, E.; and Vitek, J., 2010b. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Toronto, Ontario, Canada, Jun. 2010). doi:10.1145/1806596.1806615. (cited on pages 8 and 84)

Pizlo, F.; Ziarek, L.; and Vitek, J., 2009. Real time Java on resource-constrained platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, ACM International Conference Proceeding Series (Madrid, Spain, Sep. 2009). doi:10.1145/1620405.1620421. (cited on pages 12, 21, and 84)

PyPyDoc. PyPy documentation. https://doc.pypy.org/en/latest/introduction.html{#}. (cited on page 49)

RPyDoc. RPython Documentation. https://rpython.readthedocs.io/en/latest/index.html. (cited on pages 15, 49, and 58)

RTCA, 2011. Software considerations in airborne systems and equipment certification: DO-178C. Technical report, RTCA. https://standards.globalspec.com/std/1459138/RTCA%20DO-178. (cited on page 9)

RTSJ, 2018. *Realtime and Embedded Specification for Java*. https://www.aicas.com/download/rtsj/rtsj_76.pdf. (cited on pages 12, 49, and 52)

RTZebuGit, 2021. RTZebu VM (a fast implementation of RTMu Micro VM). https://gitlab.anu.edu.au/mu/mu-impl-fast/-/tree/rtmu-dev. (cited on page 39)

RustTeam. Rust Programming Language. https://www.rust-lang.org/. (cited on page 39)

Schoeberl, M.; Dalsgaard, A.; Hansen, R.; Korsholm, S.; Ravn, A.; Rivas, J.; Strøm, T.; Søndergaard, H.; Wellings, A.; and Zhao, S., 2016. Safety-critical java for embedded systems. *Concurrency and Computation: Practice and Experience*, 29 (12 2016). doi:10.1002/cpe.3963. (cited on page 2)

Schoeberl, M.; Dalsgaard, A. E.; Hansen, R. R.; Korsholm, S. E.; Ravn, A. P.; Rivas, J. R. R.; Strøm, T. B.; Søndergaard, H.; Wellings, A. J.; and Zhao, S., 2017. Safety-critical Java for embedded systems. *Concurrency and Computation: Practice and Experience*, 29, 22 (Nov. 2017). doi:10.1002/cpe.3963. (cited on pages 14 and 22)

Sha, L.; Rajkumar, R.; and Lehoczky, J. P., 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39, 9 (1990), 1175–1185. doi:10.1109/12.57058. (cited on page 7)

Sharp, D. C.; Pla, E.; and Luecke, K. R., 2003. Evaluating mission critical large-scale embedded system performance in real-time Java. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)* (Cancun, Mexico, Dec. 2003), 362–365. IEEE Computer Society. doi:10.1109/REAL.2003.1253283. (cited on pages 12 and 29)

Shin, K. G. and Ramanathan, P., 1994. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82, 1 (1994), 6–24. (cited on page 6)

Siebert, F., 1999. Hard real-time garbage-collection in the Jamaica virtual machine. In *Proceedings of the 6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA)* (Hong Kong, China, Dec. 1999). IEEE Computer Society. doi:10.1109/RTCSA.1999.811198. (cited on page 65)

Siebert, F., 2010. Concurrent, parallel, real-time garbage-collection. In *Proceedings of the 9th International Symposium on Memory Management (ISMM)* (Toronto, Ontario, Canada, Jun. 2010). doi:10.1145/1806651.1806654. (cited on pages 12, 21, and 65)

Stoyenko, A. D., 1992. The evolution and state-of-the-art of real-time languages. *Journal of Systems and Software*, 18, 1 (1992), 61–83. doi:10.1016/0164-1212(92)90046-M. (cited on page 1)

TheOpenGroup, 2017. *Safety-Critical Java Technology Specification*. https://jcp.org/aboutJava/communityprocess/edr/jsr302/index4.html. (cited on page 13)

TheRustProjectDevelopers, 2020. libc - crates.io: Rust Package Registry. https://crates.io/crates/libc. (cited on page 41)

Turon, A., 2015a. Abstraction without overhead: traits in Rust. https://blog.rust-lang.org/2015/05/11/traits.html. (cited on page 39)

Turon, A., 2015b. Fearless Concurrency with Rust. https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html. (cited on page 39)

Wang, K., 2018. *Micro Virtual Machines: A Solid Foundation for Managed Language Implementation*. Ph.D. thesis, Australian National University. doi:1885/147871. (cited on pages xv, 2, 9, 11, and 19)

Wang, K.; Lin, Y.; Blackburn, S. M.; Norrish, M.; and Hosking, A. L., 2015. Draining the swamp: Micro virtual machines as solid foundation for language development. In *1st Summit on Advances in Programming Languages (SNAPL)*, vol. 32 of *LIPIcs* (Asilomar, CA, USA, May 2015), 321–336. doi:10.4230/LIPIcs.SNAPL.2015.321. (cited on pages ix, 2, and 15)

Wilhelm, R.; Engblom, J.; Ermedahl, A.; Holsti, N.; Thesing, S.; Whalley, D. B.; Bernat, G.; Ferdinand, C.; Heckmann, R.; Mitra, T.; Mueller, F.; Puaut, I.; Puschner, P. P.; Staschulat, J.; and Stenström, P., 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7, 3 (2008), 36:1–36:53. doi:10.1145/1347375.1347389. (cited on page 85)

Zhang, J. J., 2015. *MuPy : A First Client for the Mu Micro Virtual Machine*. Honours thesis, Australian National University. (cited on pages 15, 49, and 55)

Åström, K. J. and Wittenmark, B., 1997. *Computer-controlled systems: theory and design (3 ed.)*. Prentice-Hall. ISBN 0-13-314899-8. (cited on page 67)

Appendixes

This chapter consists of figures that complement the main thesis chapters as follows:

(1) Figure 1 (referred in Section 6.2.2) presents our evaluation of the real-time performance of JamaicaVM's real-time garbage collector and its implementation of scoped memory. This figure justifies the JamaicaVM configuration we used in our evaluation.

(2) Figure 2 (referred in Section 6.2.3) presents our evaluation of the real-time performance of the garbage collectors available in Hotspot-11. This figure justifies our decision to use Hotspot with ZGC in our evaluation.

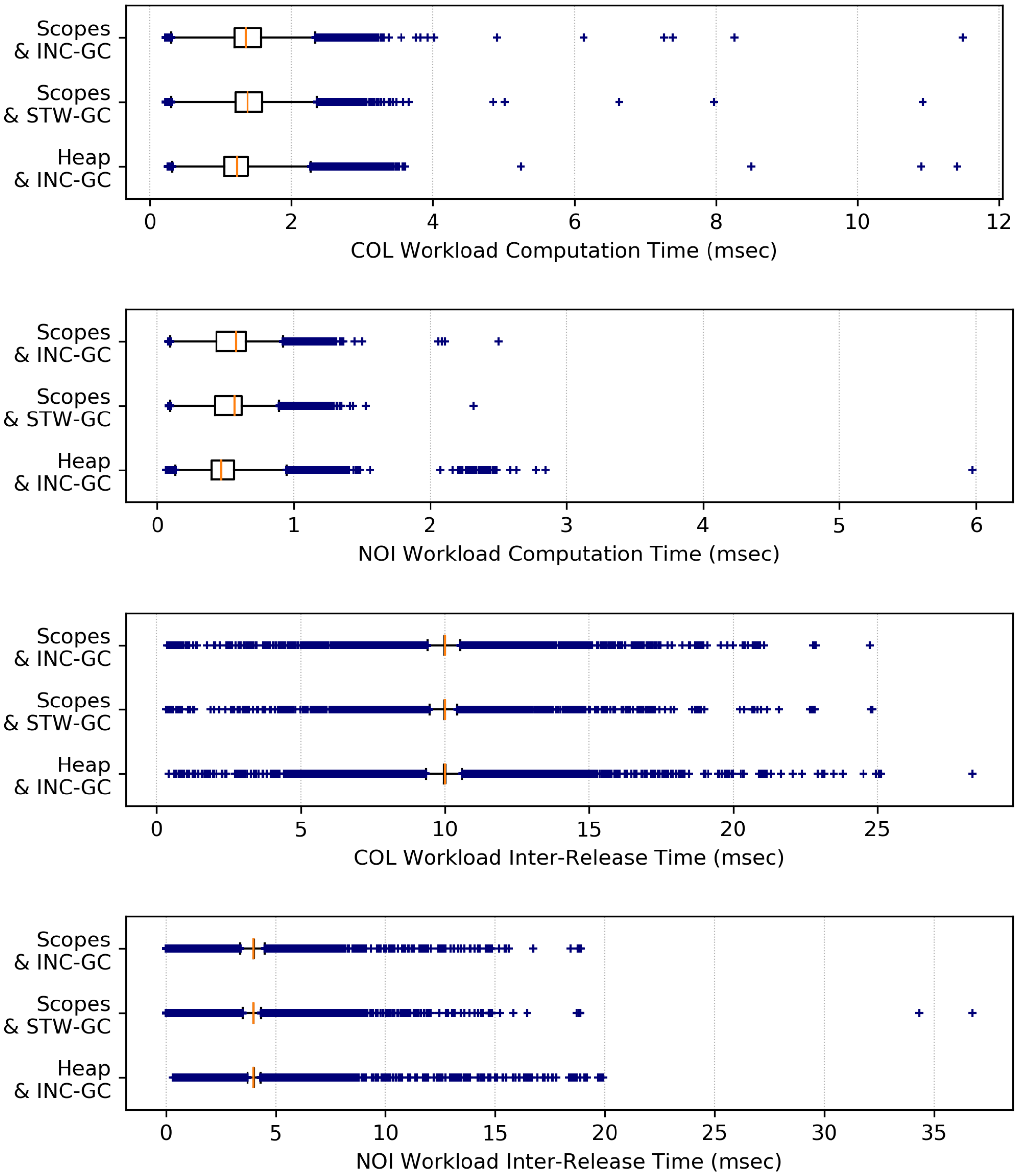


Figure 1: JamaicaVM provides two garbage collectors: A real-time GC and a stop-the-world GC (STW-GC in the figure). JamaicaVM's real-time garbage collector is an incremental, parallel and concurrent mark-sweep GC (INC-GC in the figure). We run the RTSJ version of the CD$_j$ benchmark on JamaicaVM with both GCs, to test the effect of the choice of GC on an application that uses only the immortal and scoped memory. We also run the Java version of the CD$_j$ benchmark that uses heap as its allocation context, on JamaicaVM with real-time GC, to compare its predictability to scoped memory. STW-GC and INC-GC show very similar results, except in the inter-release time of the NOI$_{nn}$ workload, where INC-GC has better worst-case behaviour. Hence, we chose to use the INC-GC in our evaluation. Finally, the Java version of the benchmark shows the best average-case performance, but its worst-case behaviour in the computation time of the NOI$_{nn}$ workload indicates its inferiority to using scoped memory with INC-GC.

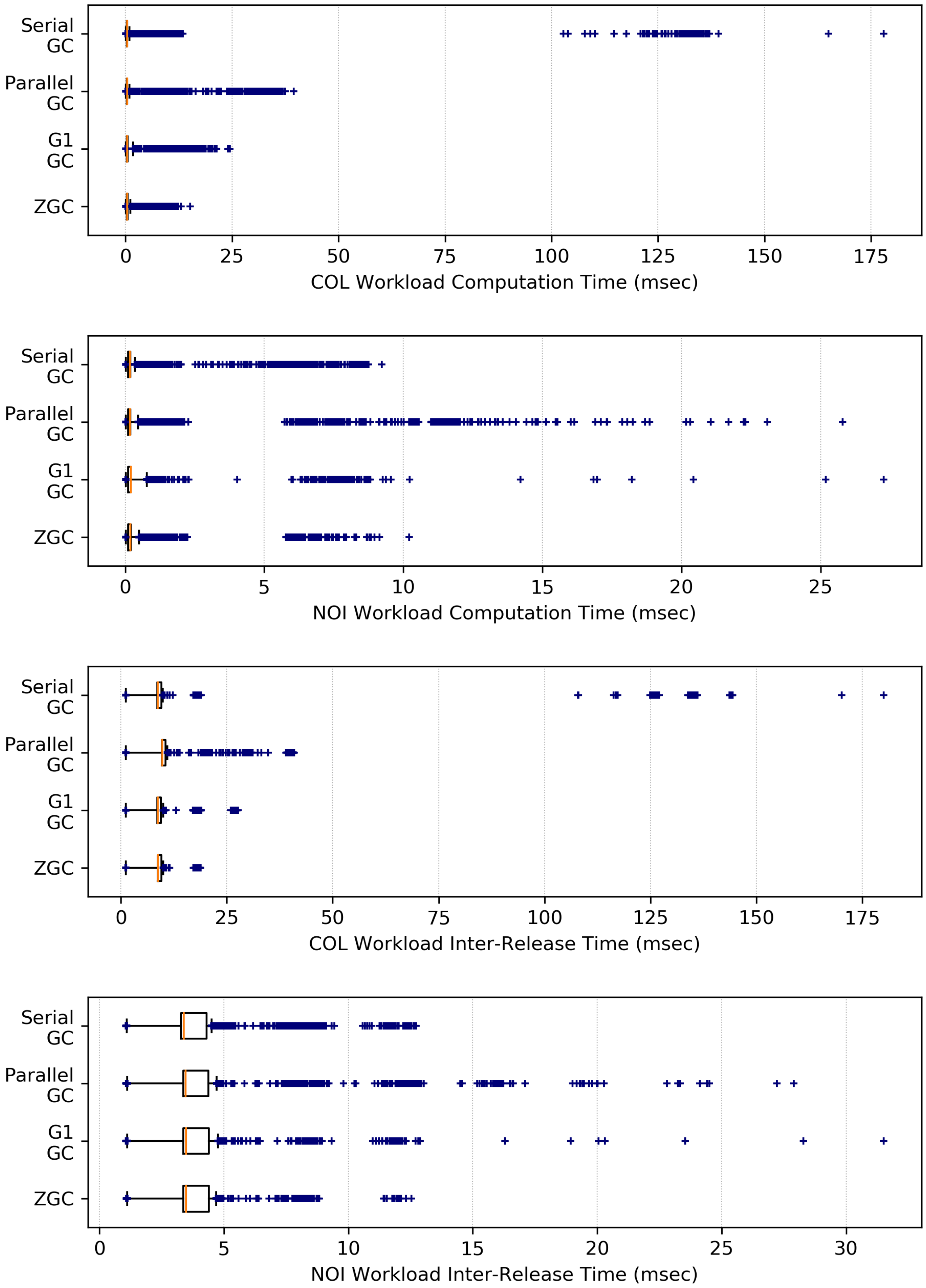


Figure 2: Hotspot-11 provides four garbage collector options: serial GC, parallel GC, garbage-first (G1) GC, and ZGC. Among them, ZGC is specifically designed for low pause times (under 10 ms). The computation times and inter-release times of both workloads confirm that ZGC is outperforming other GCs. Therefore, we use Hotspot JVM with ZGC in our evaluations.