Designing a Low-Level Virtual Machine for Implementing Real-Time Managed Languages

Javad Ebrahimian Amiri
Australian National University
Australia
Data61, CSIRO
Australia
javad.amiri@anu.edu.au

Stephen M. Blackburn
Australian National University
Australia
steve.blackburn@anu.edu.au

Antony L. Hosking
Australian National University
Australia
Data61, CSIRO
Australia
tony.hosking@anu.edu.au

Michael Norrish
Data61, CSIRO
Australia
Australian National University
Australia
michael.norrish@data61.csiro.au

Abstract

Applications of real-time systems have grown significantly in both diversity and popularity, and the appetite for real-time software has never been higher. In contrast, the choice of programming languages used to develop such systems has stagnated, mostly limited to decades-old languages, specifically Ada and C/C++, and more recently real-time Java. We posit that the high cost and difficulty of developing new programming languages for real-time systems is the main reason for this mono-culture.

To tackle the lack of diversity, we propose the design of a micro virtual machine on which managed programming languages for real-time systems can be developed. Our design facilitates bringing the advantages of correct managed languages to the real-time domain. We build on a previously published micro virtual machine 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 resulting design is a basis for a new micro virtual machine specification we call RTMu, designed as a reliable and efficient foundation for the development of managed languages for real-time systems.

VMIL '19, October 22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6987-9/19/10…$15.00
https://doi.org/10.1145/3358504.3361226

CCS Concepts • **Computer systems organization** → **Real-time languages**; *Embedded software*; *Real-time operating systems*; Reliability; • **Software and its engineering** → **Runtime environments**.

Keywords managed programming language, micro virtual machine, real-time systems

ACM Reference Format:
Javad Ebrahimian Amiri, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 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 '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3358504.3361226

1 Introduction

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 high performance and reliability, at various levels of intensity. A soft real-time application like a video player demands high performance, 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 a flight control system, which is safety-critical, timing is paramount and performance 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 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 [14] 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 [3]. This seems surprising given the diversity and popularity of real-time systems.

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. [16] argued 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, compiler backend, and garbage collector. They propose a particular μVM design instance called Mu, having a concrete specification [15], and establish its practicability through the implementation of real-world managed languages [9].

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 mono-culture 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. To ameliorate design complexity we build on the Mu μVM specification.

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.

The contributions of this paper are:

(1) design of a μVM for real-time language implementation,
(2) identification of core language features that distinguish real-time languages,
(3) specification of IR extensions to support real-time languages,
(4) specification of other runtime changes to support real-time languages, and
(5) description of how to use the new primitives to conform to the requirements of a variety of real-time applications.

The new RTMu design, which includes the IR and API, is the basis for a concrete instance of a real-time μVM. The ultimate goal of RTMu is to introduce a reliable platform that is flexible enough to serve as target for a broad range of real-time systems.

2 Background

RTMu is the first μVM designed for real-time systems. It is based-on the Mu specification [10], and inspired by real-time programming languages (RTPL) and real-time operating systems (RTOS). In this section, we first summarize some of the most influential RTPLs. Then we introduce Mu and briefly explain the key aspects which require special attention from the viewpoint of a RTPL developer.

2.1 Programming Languages for Real-Time Systems

As mentioned earlier, 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 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 based-on Mu.

SPARK SPARK is a subset of Ada for development of high-integrity software [2]. To satisfy the stringent reliability demands of such applications, it provides development tools to perform various validation and verification tasks, from unit testing, to formal proof of an application's implementation with respect to its specification. To enable this, SPARK restricts the hard-to-analyze features of Ada, including concurrency, memory management and synchronization. For instance, multi-tasking in SPARK is only possible through the Ravenscar profile for high-integrity real-time programs, introduced by Dobbing and Burns [4].

RTSJ RTSJ [6] is a specification for a type-safe Java-based managed programming language for large-scale real-time embedded systems. Such systems may collocate hard real-time, soft real-time and non-real-time code on the same platform. To support this, RTSJ introduces several enhancements and restrictions over Java, mostly in the area of concurrency and memory management. RTSJ has been used in and proven adequate for many serious real-world applications including avionics [1, 13].

SCJ SCJ is a RTSJ-based language for safety-critical real-time systems. It is designed to be capable of certification [7] (viz. DO-178B), and comprises the minimal set of features for safety-critical systems [8]. This minimality, especially in memory and concurrency managers, makes the SCJ programming model considerably different from standard Java.

Compared to other real-time programming languages, SCJ is fairly new and still under active research and development. Schoeberl et al. [12] summarize recent work on its 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 performance overheads which can make handlers with small periods infeasible.

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 POSIX standard and its real-time extensions.

Key RTPL Features Considering the above RTPLs 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.

In high-integrity systems, verifiability is the most important requirement. From a language design perspective, verifiability 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. Second, it should be possible to verify the language and its toolchain, which again means that simplicity is essential for the language design. Languages for high-integrity systems may trade ease of programming and performance for higher levels of verifiability.

In real-time systems with resource constraints—such as limited processing power, small memory, or limited energy source—the language footprint is the most critical issue. So, languages may overlook ease of use and verifiability to consume fewer resources. In such systems, the C programming language is often used, as it has a minimal runtime processing and memory footprint. In less severe 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- and dynamic- priority-based scheduling,
b. synchronization primitives supporting Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP),
c. real-time GC accompanied by other memory management techniques which are easier-to-analyse and have more predictable timing behaviour,
d. physical memory access, and
e. time management tools.

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.

2.2 Mu

Mu is a μVM specification that facilitates development of high-performance managed programming languages for a wide range of applications. It was first introduced by Wang et al. [16], and the concrete specification is available online [10]. 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, so long as doing so does not jeopardize viability or efficiency of the three main functionalities.
- Mu assumes language implementations (μVM clients) are trusted. Hence, unnecessary overhead is 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 clearly defined behavior, admitting multiple compliant implementations.

Architecture The high-level architecture of a managed language based on Mu is depicted in Figure 1. Although not evident in the figure, Mu is only a thin abstraction layer, and the client does the bulk of the job by implementing a managed language on top of Mu's abstractions.

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. These bundles are then compiled to machine code. 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. 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.

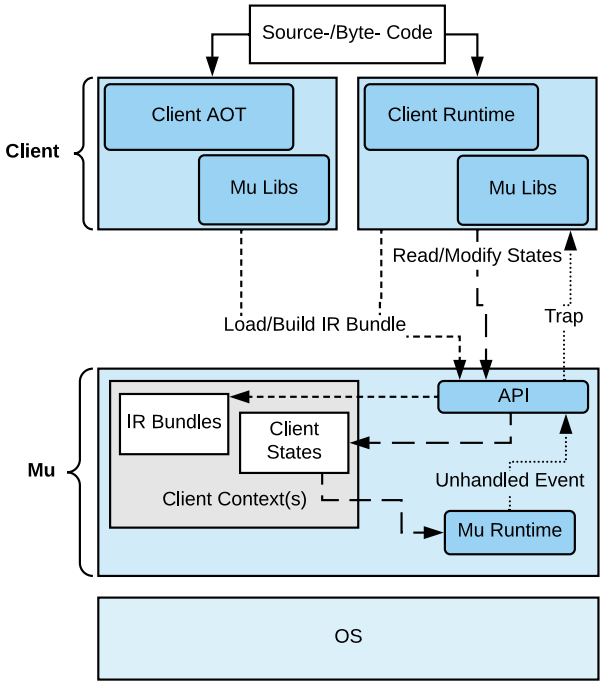

Figure 1. A Mu client, e.g. a managed language, builds Mu IR bundles and loads them to Mu to be executed. This can either happen ahead-of-time or at run time. It can also read and modify the internal state of the VM. In case of any non-trivial events, the Mu runtime traps to the client to handle it. All of the communication between Mu and the client is done through the Mu API.

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 1, categorized into four groups: (1) 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 real-world managed language type system, it still lacks some of the basic data types required by the additional features of RTPLs. These data types are explained in Section 3.3.

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 [5]. `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.

Table 1. The Mu type system is simple and low-level, and consists of primitive-numerical, aggregate, reference and miscellaneous data 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

To implement diverse synchronization primitives, Mu provides a basic tool that is similar to 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, conditon variables, and semaphores. 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.

Most real-time systems require more control over the behavior of multithreaded programs, compared to what Mu provides. We discuss the necessary concurrency abstractions in Section 3.6.

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 the current Mu memory management. However, it is too complex for some real-time systems, and imposes intolerable pauses or overhead for many of them. In Section 3.5, 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, 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 effective 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 3.4.

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 acts as a proof of implementation [15], and a high-performance implementation [9], which is still under development. To demonstrate the capability of Mu in supporting a real-world language as a client, some experiments are reported in Wang [15]. 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.

3 RTMu Design

Like Mu, RTMu provides low-level abstractions over concurrency, compiler backend and memory management, and is designed to be minimal and verifiable. RTMu aims to support a wide range of real-time systems, including high-integrity systems.

3.1 Scope

RTMu is a concrete μVM specification designed to facilitate development of correct programming languages for real-time systems. This paper restricts itself to an RTMu design, not an implementation or related tools. In this section, we specify the range of real-time systems covered by this design.

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:

- 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.
- 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.
- Basic tools for time measurement and time-triggered events.

Considering the already-demonstrated capability of Mu in implementing real-world languages, we argue that the above features make RTMu capable of supporting the implementation of real-world RTPLs such as RTSJ. However, some of these features, such as dynamic memory allocation and dynamic-priority scheduling, are not necessary in 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.

Assumptions Following Mu, we assume the client is trusted, and 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, to avoid the related overheads.

Non-Goals To narrow the scope of this project, we declare some non-goals. First, we do not aim to produce an implementation to compete with or improve on mature, heavily invested programming languages used in real-time systems such as real-time Java. Instead, we are *designing* a reliable foundation to facilitate the emergence of new high-quality real-time managed languages to increase the breadth of the RTPL ecosystem. Second, preserving minimality, this design drops support for popular general-purpose language features like JIT compilation, interpretation and dynamic class loading, which introduce huge delays or make timing and correctness analysis of real-time applications very hard. Third, 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.

3.2 Architecture

RTMu-based managed language implementations roughly follow the same high-level architecture as Mu, as depicted in Figure 1. The major difference is that RTMu only supports AOT compilation which means there will be no IR load/build through the API at run-time. Also, the RTMu runtime manages a new explicitly-managed memory area in addition to the heap, immortal memory, and stacks.

3.3 Type System

For the RTMu type-system, we reuse the whole Mu type system and add five types to support newly added features for real-time systems. The added types are shown in Table 2.

For the basic time management primitives of RTMu, we add the `timeval` type to store time values, and the `timerref` type to identify timers. These two types and their usage are explained in Section 3.7.

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 encapsulate all these attributes in a new type named `rtattr`. An object 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.

3.4 Compiler Backend and IR

The RTMu system is responsible for executing the IR code given to it by the client. As 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.)

Table 2. RTMu adds new data types, required for its new real-time features.

Type	Description
`timeval`	Time values
`timerref`	References to timers
`regionref`	References to explicitly managed memory regions
`rtattr`	Thread attributes
`futexref`	References to futexes

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.

3.5 Memory Management

The 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:

- Stack area,
- Garbage collected heap,
- Explicitly managed memory area,
- Region area,
- Immortal memory area.

A summary of these memory areas and the operations a client can perform on them is shown in Figure 2. Each of these areas are 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. There is one new instruction for stacks, and the rest act on the new EMM and region areas.

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 specific GC algorithms. Instead, the client 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 [13]. 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 GC complexities.

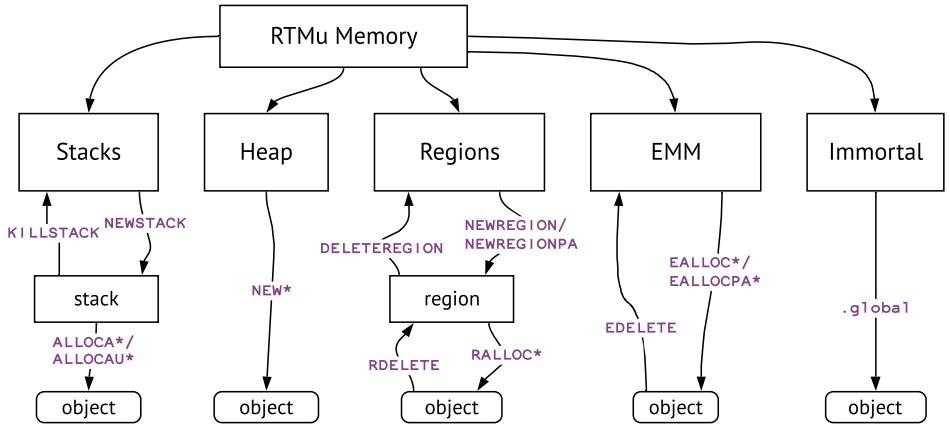

Figure 2. The RTMu memory is divided into five areas, each serving a range of higher level memory managers. Among them, stacks, garbage collected heap and immortal (static or global) areas are common in non-real-time managed languages. For real-time languages, we add EMM and regions which are highly 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.)

Table 3. RTMu adds several instructions to support implementation of the common memory managers in RTPLs and more. The first instruction in the list is essential for providing stacks for tasks not using the heap. The next seven instructions serve region-based memory, like the RTSJ scoped memory. The last five instructions mainly aim manual dynamic memory, like `malloc` and `free` in C.

Operation	Description
`uptr<T> ALLOCAU/ALLOCAUHYBRID (T)`	Allocate an untraced fixed/variable-size object of type `T` on stack.
`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 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`.
`uptr<T> RALLOC/RALLOCHYBRID (regionref regref, T)`	Allocate a fixed/variable-size object of type `T` on the region `regref`.
`uptr<T> EALLOC/EALLOCHYBRID (T)`	Allocate a fixed/variable-size object of type `T` on the EMM space.
`uptr<T> EALLOCPA/EALLOCHYBRIDPA (T, uptr<void> addr)`	Allocate a fixed/variable-size object of type `T` at the specified physical memory address.
`void EDELETE (uptr<T> ptr)`	Delete the object pointed by `ptr` from the EMM space.
`void BINDOBJECT (uptr<T> ptr)`	Disable swap-out to disk for the EMM object pointed by `ptr`.
`void UNBINDOBJECT (uptr<T> ptr)`	Re-enable swap-out to disk for the EMM object pointed by `ptr`.

Stacks In Mu, on-stack allocation 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 as 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 the complexities or delays. Such tasks should not experience any interference from the GC. Consequently, we add **`ALLOCAU`** and **`ALLOCAUHYBRID`** instructions for untraced allocation on stack, which return an untraced pointer to the allocated object. By restricting the client to only use these operations, the GC will not need to scan the stack for references.

Explicitly Managed Memory The EMM area of RTMu provides low-level primitives to implement memory managers, which support both allocation and delete, at the granularity of an object. For instance, the `malloc` and `free` functions in C can be built in 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 on 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. In this case, the OS memory manager may swap the memory page(s) holding the object out to a 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 then use the **UNBINDOBJECT** instruction to allow the 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 memory page swapping, the **BINDOBJECT** instruction will not do anything. Also, if an RTOS restricts the access to physical memory or specific physical addresses, the relevant RTMu instruction will not be available.

Regions The *region* 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 *region* 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. 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 the related region. The RTMu region area 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 **UPTRTOREG** intrinsic, which gets 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. Listing 1 shows a scenario in RTSJ, where an exception is thrown to indicate a potential dangling reference. A translation of the this scenario to RTMu IR is depicted in Listing 2.

As explained for EMM objects, the client may need to bind and unbind regions to the 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 new **NEWREGIONPA** instruction.

Similar to the stack, the client may allocate objects, with or without references to heap, on a region. A RTMu implementation is responsible for guaranteeing that a region with no references to the heap is not affected (e.g. accessed or traced) by the GC. Additionally, the GC must have access to all the required data to collect heap garbage correctly and leave no dangling references.

Immortal Memory The immortal memory area is the preferred tool to implement memory managers which do not

```
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 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, which the RTSJ VM must detect and trigger an exception at the point of the assignment.

```
regionref _inner_scope = NEWREGION (
    const_size)
// client-written function to keep scopes
    and update current_scope
CALL push_scope (_inner_scope)
uptr<uptr<SomeList_t>> _head_next =
    GETFIELDIREF PTR <_headptr, 0>
uptr<SomeList_t> _tail = RALLOC (
    current_scope, SomeList_t)
// check lifetimes
regionref _dest = UPTRTOREG (_head_next)
regionref _src = UPTRTOREG (_tail)
// client-written function to quantitate
    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 <<uptr<SomeList>> _head_next,
      _tail
```

Listing 2. A translation of the last four lines of the RTSJ code in Listing 1 into to (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, namely preventing dangling references.

```
Fn parent()
BEGIN
    // the period is 1 milli-seconds
    timeval _period = 1_000_000ns;
    stackref _stack = NEWSTACK (entry);
    CALL init_thread (_stack, _period);
END

Fn entry(uptr<void> arg)
BEGIN
    CALL wait (self.cond);
      // THREAD BODY GOES HERE
    BRANCH BEGIN
END

// initialize the periodic thread
Fn init_thread(stackref _stack, timeval
    _period)
BEGIN
  uptr<rtattr> _attr = NEW rtattr;
  threadref _thread = NEWRTTHREAD (_stack,
      _attr);
  timerref _timer = NEWTIMER (); // not
      started yet
  // call wake_thread periodically
  SETTIMER (_timer, _period, wake_thread,
      _thread);
END

Fn wake_thread(threadref _thread)
BEGIN
    CALL wake(_thread.cond);
END
```

Listing 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.

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.

3.6 Concurrency

RTMu improves the concurrency primitives of Mu for real-time systems in several ways. 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. Each priority level may accommodate zero or more tasks. Both the maximum number of priority levels and the number of tasks at each level depend on the limitations of the underlying RTOS scheduler. 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 as the most popular scheduling approach in real-time systems [3]. The last one is a foundation for 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.

To enable the scheduler, three attributes are added to RTMu threads. (1) The **`priority`** attribute is a number between zero (the highest priority) and the number of scheduler'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 a value of type **`timeval`**, 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 **`rtattr`**. When a client creates a new real-time thread, they should pass the initial attributes of the thread as an object of this type. The internal structure of the **`rtattr`** type is platform-dependent, and modifying objects of this type is only possible through the provided RTMu instructions.

To implement synchronization primitives, RTMu provides a basic tool similar to RT-Linux PI-Futex, which supplies both PIP and PCP to prevent the Priority Inversion and Deadlock problems. As with Mu, RTMu clients are responsible for implementing higher-level language-specific synchronization mechanisms, like Mutex and Semaphore, using futex and atomic operations.

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

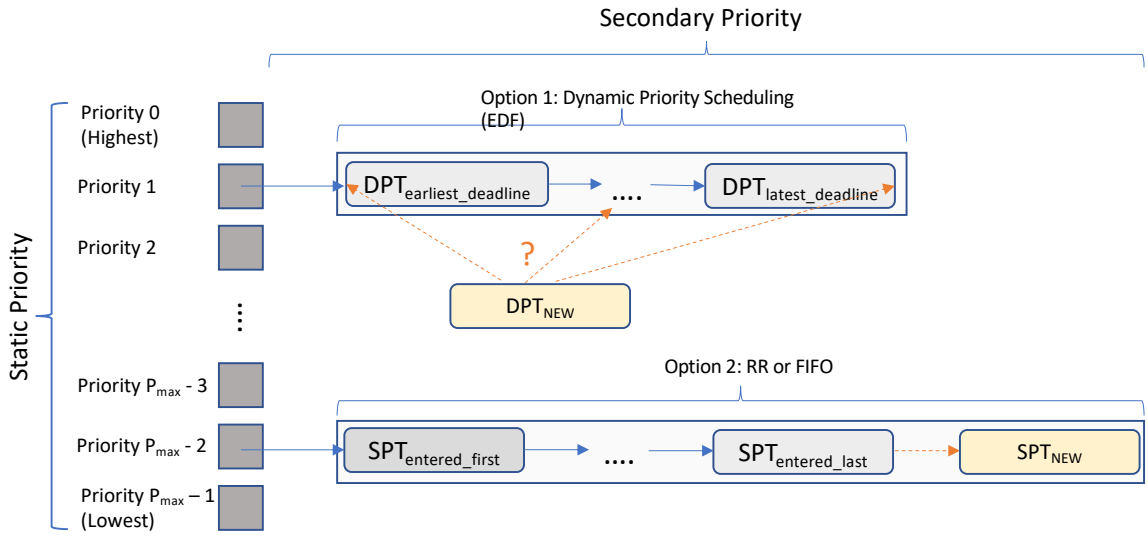

Figure 3. The RTMu Scheduler consists of a number of (static) priority levels. For each level, the scheduler keeps a queue of ready tasks. The position of a new ready task in the queue depends on the scheduling policy at that priority level. For RR and FIFO, the new task is always the last in the queue, and for EDF, tasks with smaller deadlines are inserted closer to the queue head.

RTMu implementation on RT-Linux may choose not to provide EDF at all priority levels if the complexity or overhead is too high.

3.7 Clock and Timer

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. RTMu adds two new types for this purpose. The first one is the `timeval` type, an integer number used to keep all RTMu time values in nanoseconds. The second is the `timerref` type that saves timer IDs.

All time-related RTMu instructions are mentioned in Table 4. They can be categorized to: (a) read or modify the current system clock value, and (b) create, monitor, modify and delete Time-Triggered (TT) events. TT events are one of the basic elements of many real-time applications. As an example, the pseudo-code for creating a periodic task using a timer is shown in Listing 3.

3.8 RTMu API

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 as 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 behavior 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.

4 Conclusion and Future Work

Currently, the choice of programming languages for real-time systems is mostly limited to Ada, C/C++ and real-time Java. To relieve this lack of diversity, this paper presents our design of RTMu, a μVM specification of a runtime system to execute managed languages for real-time systems.

We analyze an indicative range of real-time languages to extract their key features. Then, building on the Mu μVM specification, we introduce a minimal set of modifications to its IR, API and runtime sufficient to support the required features. The modifications include (a) giving the client more control over concurrency, (b) a new memory manager that supports colocation of various strategies on the same system, and (c) removal of the features that are either too complex for the level of validation needed in most real-time systems, or introduce large delays. We also argue that the compiler backend of Mu can be reused in RTMu.

RTMu is designed to be minimal and language-neutral. Thus, it can be used for a large range of real-time systems with various requirements. It is also designed to be formally verifiable, enabling its use in applications requiring the highest levels of validation.

The concrete specification of RTMu is available online [11]. We are also developing an open-source, performant implementation of the RTMu specification. As a long-term goal, we plan to provide a formally verified RTMu implementation. Another goal to investigate in the long-term would be to integrate WCET analysis in RTMu. We are optimistic that RTMu will provide a suitable platform for development of managed programming languages for real-time systems.

Table 4. The new clock and timer methods in RTMu include two basic operations to read and modify the clock, and four basic operations to manage timers.

Method	Description
`timeval GETTIME ()`	Returns the current system time
`void SETTIME (timeval tm)`	Resets the current system time to `tm`
`timerref NEWTIMER ()`	Creates a new timer and returns a handle
`void SETTIMER (timerref tmr, timeval tm, ufuncptr fn, uptr<void> arg)`	Activates the timer `tmr` to call `fn(arg)` after an interval `tm`
`void CANCELTIMER (timerref tmr)`	Deactivates the timer "`tmr`"
`void DELETETIMER (timerref tmr)`	Deletes the timer "`tmr`"

References

[1] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. 2007. A real-time Java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems* 7, 1 (dec 2007), 1–49. https://doi.org/10.1145/1324969.1324974

[2] John Barnes. 1997. *High integrity Ada: the SPARK approach.* Addison-Wesley Professional.

[3] Alan Burns and Andy Wellings. 2009. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX* (4th ed.). Addison-Wesley Educational Publishers Inc, USA.

[4] Brian Dobbing and Alan Burns. 1998. The Ravenscar tasking profile for high integrity real-time programs. *ACM SIGAda Ada Letters* XVIII, 6 (nov 1998), 1–6. https://doi.org/10.1145/301687.289525

[5] Stephen Dolan, Servesh Muralidharan, and David Gregg. 2013. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization* 9, 4 (jan 2013), 1–25. https://doi.org/10.1145/2400682.2400695

[6] JSR-282 Expert Group. 2018. *Realtime and Embedded Specification for Java.* http://aicas.com/cms/sites/default/files/rtsj{_}65.pdf

[7] The Open Group. [n.d.]. The Java Community Process(SM) Program — JSRs: Java Specification Requests — detail JSR #302. https://jcp.org/en/jsr/detail?id=302

[8] The Open Group. 2017. *Safety-Critical Java Technology Specification.* http://download.oracle.com/otn-pub/jcp/safety{_}critical-0{_}109-edr3-spec/scj-EDR3.pdf?AuthParam=1528777257{_}a53849c4e96ca1c7bfc7be12c23019e8

[9] Yi Lin. 2019. *An efficient implementation of a micro virtual machine.* Ph.D. Dissertation. The Australian National University. https://doi.org/1885/158122

[10] Mu 2018. The specification of Mu. https://gitlab.anu.edu.au/mu/mu-spec

[11] RTMu 2019. The specification of RTMu. https://gitlab.anu.edu.au/mu/rtmu-spec

[12] Martin Schoeberl, Andreas Engelbredt Dalsgaard, René Rydhof Hansen, Stephan E. Korsholm, Anders P. Ravn, Juan Ricardo Rios Rivas, Tórur Biskopstø Strøm, Hans Søndergaard, Andy Wellings, and Shuai Zhao. 2017. Safety-Critical Java for embedded systems. *Concurrency and Computation: Practice and Experience* 29, 22 (nov 2017), e3963. https://doi.org/10.1002/cpe.3963

[13] D.C. Sharp, Edward Pla, and K.R. Luecke. 2003. Evaluating mission critical large-scale embedded system performance in real-time Java. In *Proceedings. 2003 International Symposium on System-on-Chip (IEEE Cat. No.03EX748).* IEEE Comput. Soc, 362–365. https://doi.org/10.1109/REAL.2003.1253283

[14] Alexander D Stoyenko. 1992. The evolution and state-of-the-art of real-time languages. *Journal of Systems and Software* 18, 1 (apr 1992), 61–83. https://doi.org/10.1016/0164-1212(92)90046-M

[15] Kunshan Wang. 2018. *Micro Virtual Machines: A Solid Foundation for Managed Language Implementation.* Ph.D. Dissertation. Australian National University. https://doi.org/1885/147871

[16] Kunshan Wang, Yi Lin, Stephen M. Blackburn, Michael Norrish, Antony L. Hosking, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett. 2015. Draining the Swamp: Micro Virtual Machines as Solid Foundation for Language Development. *1st Summit on Advances in Programming Languages (SNAPL 2015)* 32 (2015), 321–336. https://doi.org/10.4230/LIPIcs.SNAPL.2015.321