Draining the Swamp: Micro Virtual Machines as Solid Foundation for Language Development

Kunshan Wang[1], Yi Lin[1], Stephen M. Blackburn[1], Michael Norrish[2,1], and Antony L. Hosking[3]

1 **Research School of Computer Science, Australian National University***
108 North Road, Canberra, ACT, Australia
kunshan.wang@anu.edu.au, yi.lin@anu.edu.au, steve.blackburn@anu.edu.au

2 **Canberra Research Lab., NICTA†**
7 London Circuit, Canberra, ACT, Australia
michael.norrish@nicta.com.au

3 **Department of Computer Science, Purdue University‡**
305 N. University St., West Lafayette, IN, USA
hosking@purdue.edu

Abstract

Many of today's programming languages are broken. Poor performance, lack of features and hard-to-reason-about semantics can cost dearly in software maintenance and inefficient execution. The problem is only getting worse with programming languages proliferating and hardware becoming more complicated. An important reason for this brokenness is that much of language design is implementation-driven. The difficulties in implementation and insufficient understanding of concepts bake bad designs into the language itself. Concurrency, architectural details and garbage collection are three fundamental concerns that contribute much to the complexities of implementing managed languages.

We propose the *micro virtual machine*, a thin abstraction designed specifically to relieve implementers of managed languages of the most fundamental implementation challenges that currently impede good design. The micro virtual machine targets abstractions over memory (garbage collection), architecture (compiler backend), and concurrency. We motivate the micro virtual machine and give an account of the design and initial experience of a concrete instance, which we call Mu, built over a two year period. Our goal is to remove an important barrier to performant and semantically sound managed language design and implementation.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases virtual machines, concurrency, just-in-time compiling, garbage collection

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.321

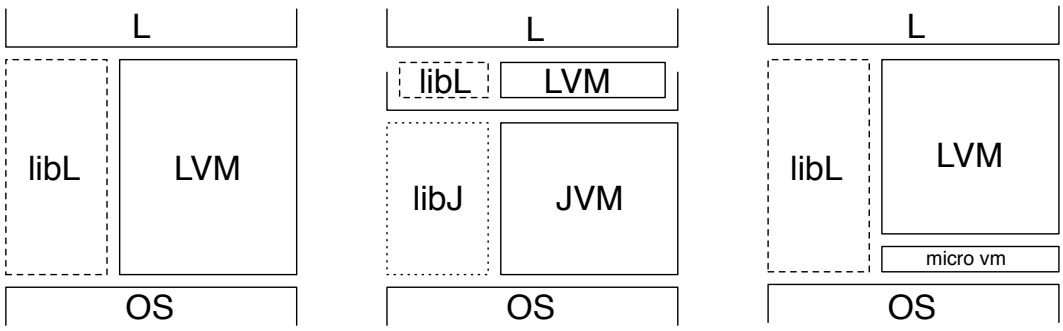
1 Introduction

Today's programming landscape is littered with otherwise important languages that are inefficient and/or hard to program. The proliferation of these languages is not symptomatic of a disease but evidence of a vibrant programming language ecosystem. The appeal of such languages has seen them become heavily used in critical settings. Unfortunately, the

* Supported by ARC grant no. DP0666059.
† NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.
‡ Supported by National Science Foundation grant no. CCF-0811691.

1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 321–336

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) Monolithic **(b)** VM on Macro VM **(c)** VM on Micro VM

Figure 1 Three implementation strategies for a managed language L. Traditional VMs (left) are typically monolithic designs, reusing little or nothing. Macro VMs such as the JVM and .NET (center) provide heavyweight reuse but offer far more than many language runtimes need. A micro virtual machine (right) provides only a thin abstraction over core concerns.

performance overheads in many cases are best measured on a log scale, leading to very real costs for the companies that have come to depend on them. Likewise, inscrutable semantics cost dearly as systems grow and maintenance becomes a nightmare. The vibrancy of the language ecosystem and rapid changes in the nature of the systems on which they are deployed suggests to us that this is a problem that is set only to get worse.

Our position is that many of the performance and semantic issues that befall such languages can be traced to fundamental implementation challenges. For example, PHP's confounding copy-on-write semantics [16], can be traced directly to a bug report dating to 2002, in which a user first observed the behavior [15]. Five days later, upon realizing that the fix would be challenging, the developers declared the "broken" semantics to be a feature of the language, and it has remained so to this day. The engineering challenge of implementing a garbage collector has led many languages to depend on naïve reference counting in their earliest implementations despite its well-known performance limitations and inability to collect cycles [11, 2]. When the owners of the language then make virtue of necessity and expose reference counting semantics in the language, this expensive but expedient implementation choice gets baked in. Similarly, the intellectual challenge of correctly supporting concurrency has led many languages to have a weak, broken, or absent model of concurrency, a limitation that grows increasingly embarrassing as stock hardware offers higher and higher degrees of parallelism.

Three fundamental concerns contribute to much of the complexity of language implementation: compiler back ends, garbage collection, and concurrency. Each are technical minefields in their own right but when brought together in a language runtime, their respective complexities combine in very challenging ways. Each of these concerns is rich enough to warrant a well developed research sub-community and rich literature of its own. We surveyed a handful of language maintainers and found a near-universal desire to be unburdened of the need to deal with these elements of language implementation, which cannot be ignored and yet frequently distract from the language implementer's principal interest in the higher levels of the language's design and implementation.

We propose a *micro virtual machine* as a robust, performant and lightweight abstraction over just three concerns: hardware (compiler back end), memory (garbage collector), and concurrency (scheduler and memory model for language implementers). Such a foundation

attacks many of the issues that face existing language designs and implementations, leaving language implementers free to focus on the higher levels of the language design.

Unlike LLVM [13], a micro virtual machine is not a compiler framework, it natively supports garbage collection and concurrency, while performance-critical language-specific optimizations are performed outside the micro virtual machine in higher layers of the run-time stack. Unlike monolithic language implementations, a micro virtual machine explicitly supports cross-language reuse of demanding implementation details. Unlike the JVM, CLR, and LLVM, a micro virtual machine is minimalist, and explicitly designed to support the development of new languages, targeting a low level of abstraction, minimizing the semantic gap [5].

We have embarked on the ambitious project of designing and constructing a concrete instantiation of a micro virtual machine, with the goal of testing our hypothesis that it will advance the cause of language design and implementation. This paper reports on our motivation and two years of experience in designing and building a prototype micro virtual machine, which we call Mu. The current Mu specification is available online [14]. We will discuss a few of the more interesting aspects of the system design and implementation, including the type system, exact garbage collection, dynamism including OSR, the IR and API, and the pervasive design requirement of minimalism. We will include discussion of our preliminary experience in targeting Lua and Python implementations to Mu.

Ambitious as it is, our primary goal is that future languages will be less likely to have their semantics and performance dictated by fundamental implementation hurdles imposed upon the designers early in the language's life. A secondary goal is to improve existing language implementations, unburdening developers from elements of the language implementation that are critical but relatively uninteresting to them. It is not a goal of our project to improve upon language implementations that have benefited from massive commercial investments, such as Java.

2 Motivation

A large fraction of today's software is written in *managed languages*. Examples include JavaScript, PHP, Objective-C, Java, C#, Python, and Ruby. These languages are economically important. For example, Facebook depends on servers running PHP for its core business of efficiently delivering hundreds of billions of page views a month, Apple depends on Objective-C for every iPhone app, while Google depends on Java for its Android apps, and uses JavaScript to power its most widely used web applications including search and Gmail. Unfortunately, some of these languages are notoriously inefficient, imposing overheads as large as a factor of fifty compared to orthodox language choices such as C. The source of this inefficiency often lies in the language implementation (rather than the language itself), and this systemically impedes all applications written in that language. Moreover, early implementation choices often impede evolution of the language and/or implementation by baking implementation decisions into the language definition. A classic example is the transition from the early implementation "mistake" that resulted in dynamic scoping for LISP to the much saner static scoping. Similarly, PHP and Perl assume that their implementations use an extremely naïve reference counting memory management strategy [18]. As another example, Python's infamous "global interpreter lock" (GIL) [3] is a relic of an early implementation decision that limits Python and its GIL-infected peers (*e.g.*, Ruby) to sequential (non-parallel) execution, preventing programs from fully exploiting modern parallel hardware.

Implementing a new language can be easy when it is done naïvely, but taking the care to avoid early bad design decisions is hard. Enthusiastic implementers want to get their

new language up and running without initially worrying about performance. But improving performance can require significant investment to achieve: witness the large investment by Sun/Oracle, IBM and others over many years to get Java to perform, and the competition among companies to make their JavaScript implementations outperform their competitors. And bad early language design and implementation decisions can cost even more to overcome.

The origins of systemic inefficiency of new languages often comes from the way they are implemented. When a language is implemented from the ground up in a monolithic fashion, developers must directly address every performance challenge. The problem is that different challenges often have subtle interactions. For example, the compiler, garbage collector, and thread subsystems must be designed to work together, but few developers have expertise in more than one of these subsystems. Fewer still will have expertise in their intersection, such as in the design and implementation of GC maps. An alternative approach is to avoid implementation challenges by building on top of existing language infrastructures, such as the JVM or .NET. These are large, heavily-invested platforms supporting advanced memory management, portability, aggressive just-in-time compilation, advanced support for concurrency, and come with extensive libraries. Both the JVM and CLR dictate a high level of abstraction tailored to the languages they support, which reduces the implementation effort, but which can be a poor match to the new language [5]. There are less obvious disadvantages. For example, the ease of integration of JRuby with Java leads developers to write performance-critical code in Java. Though inefficient compared to C, JRuby outperforms other Ruby implementations, so performance-critical Ruby applications inadvertently and unintentionally become dependent on Java.

Other infrastructures as a supporting substrate for language implementation, such as LLVM, have their own deficiencies. LLVM's focus on C means that support for garbage collection, concurrency, and dynamic typing are minimal or weak. Moreover, LLVM's focus on heavy-duty static optimizing compilation of C/C++ leaves its support for dynamic just-in-time (JIT) compilation as somewhat of an afterthought, receiving less attention to quality and maintenance. In contrast, dynamic languages rely heavily on on-going run-time (re)compilation to achieve good performance. Nonetheless, LLVM offers a good model with respect to intermediate language design and level of abstraction. Other infrastructures such as VMkit are composed from discrete library components that do not address the need for cross-cutting designs that span compilation, concurrency, and memory management, as discussed earlier.

The lack of suitable infrastructures drives us to propose *micro virtual machines* as a unifying substrate for language implementers that will support efficient execution of the abstractions it presents. We describe a micro virtual machine instance that will execute low-level code issued by high-level language compilers and/or run-time systems, sitting at the base of language implementations. It will take care of the most fundamental concerns of managed languages, with the bulk of the client run-time system above providing the personality of the specific client language. A micro virtual machine offers implementers a "third way," giving them access to state of the art foundations, whilst freeing them to implement language-specific semantics with maximum liberty and efficiency. Moreover, we hope that by keeping its code base as small as possible (approximately 25K lines of code), our micro virtual machine will present a suitable target for verification, allowing it to join the trusted computing base.

3 Mu: A Concrete Micro Virtual Machine

We now flesh out the design and implementation of Mu, our initial instantiation of a micro virtual machine. Space constraints limit the discussion here to a high level with a few key details, but Mu is open source, and both the specification and the source code of reference implementations are available on our website [14].

The broad architecture of Mu is reminiscent of other virtual machines such as the JVM or .NET. It executes dynamically loaded code by interpretation or compilation. A language-specific run-time system and supporting libraries sits above it. The principal difference is in the much lower level of abstraction at which micro virtual machines operate. The Mu instruction set is SSA-based rather than stack-based, the type system is much simpler, the concurrency primitives are low-level, and all high-level optimizations are the responsibility of the client run-time system, not the micro virtual machine.

A number of principles underpin the design of Mu: (1) Mu is explicitly *minimal*; any feature or optimization that can be deflected to the higher layers will be. (2) Minimalism will be compensated for by *client libraries* that sit above Mu, implementing higher level features, conveniences, transformations, and optimizations common to more than one language client. (3) The specification allows for *formally verifiable* instantiations, supporting our long term goal of a formally verified Mu instance. (4) Mu's client is *trusted*; it is allowed to shoot itself in the foot. (5) We use LLVM IR [13] as a *common frame of reference* for our IR, deviating only where we find compelling cause to do so. (6) We *separate specification and implementation*; Mu is an open specification against which clients can program and which different instantiations may implement. (7) We aim to support *diverse* language clients. (8) The Mu design will facilitate high performance language implementations.

Our goal is that Mu will facilitate the design and implementation of new and existing languages by abstracting over three of the most fundamental implementation concerns. Our focus is on languages most exposed to these concerns, namely dynamic managed languages. However, we are exploring clients for languages as diverse as Erlang, Haskell, Go, Lua and Python. It is *not* our goal to provide an implementation layer that will compete with mature, highly tuned runtimes such as the HotSpot JVM, which have benefited from enormous investment over a decade or more.

3.1 Mu Architecture

The Mu specification consists of the *Mu intermediate representation* and the *Mu client interface*. The Mu IR is the low-level language accepted and executed on Mu, while the Mu client interface defines the programming interface for client language runtimes. The client language runtime is responsible for (JIT-)compiling source code, bytecode or recorded traces into Mu IR. Mu IR code is delivered to Mu in the unit of *code bundles*, the counterpart of LLVM modules and Java class files. The Mu client interface specifies how the client may directly manipulate the state of Mu, including loading Mu IR bundles by sending messages to Mu, and how Mu-generated asynchronous events are handled by the client.

The abstract state of an executing Mu instance comprises some number of execution engines (`threads`), execution contexts (`stacks`), and memory accessed via references. Mu's abstract threads are similar to (and may directly map to) native OS/hardware threads. Stacks contain frames, each containing the context of a function activation, including its current instruction and values of local variables. Memory consists of a garbage-collected heap, a global memory, and memory cells allocated on the stacks. The abstract state can be changed by executing Mu IR code directly or by invocation of operations by the client through the Mu client interface.

The Mu project is under active development. The website [14] includes an initial Mu specification and source code for a reference implementation (which does not consider performance). We are concurrently developing a high performance Mu implementation. The reference implementation is intended for early evaluators to experiment with Mu.

3.2 Type System

The Mu type system is very simple, and designed to support both safe and unsafe memory operations. It features integer types in varying bit-widths, two floating point types, vector types for SIMD instructions, composite types in the form of `struct`s and arrays, multiple kinds of memory reference types and opaque reference types:

$$
\begin{array}{rcl}
\tau_0 & ::= & \texttt{void} \mid \texttt{int}\langle n\rangle \mid \texttt{float} \mid \texttt{double} \\
& \mid & \texttt{vector}\langle\texttt{int}\langle n\rangle; m\rangle \mid \texttt{vector}\langle\texttt{float}; n\rangle \mid \texttt{vector}\langle\texttt{double}; n\rangle \\
& \mid & \texttt{struct}\langle\tau_0^+\rangle \mid \texttt{array}\langle\tau_0; n\rangle \\
& \mid & \texttt{func}\langle\tau_0^*; \tau_0\rangle \mid \texttt{thread} \mid \texttt{stack} \\
& \mid & \texttt{ref}\langle\tau\rangle \mid \texttt{iref}\langle\tau\rangle \mid \texttt{weakref}\langle\tau\rangle \mid \texttt{ptr}\langle\tau\rangle \mid \texttt{tagref64} \\
\tau & ::= & \tau_0 \mid \texttt{hybrid}\langle\tau_0; \tau_0\rangle
\end{array}
$$

Types can be recursive under the reference types. The most basic types are scalar and vector integer and floating point types. Integers do not have signedness, but concrete operations, including `UDIV` and `SDIV`, may treat integer operands as signed or unsigned.

Object references $\texttt{ref}\langle\tau\rangle$ are references to *objects*[1] that have been allocated in the heap, and that will be managed by the garbage collector. Internal references $\texttt{iref}\langle\tau\rangle$ provide references to memory locations that may be internal to objects (*e.g.*, `array` elements or `struct` fields). Both object references and internal references are traced, and will keep their referents alive on the heap if the reference is itself reachable from GC roots. Weak references $\texttt{weakref}\langle\tau\rangle$ are object references that may be set to `NULL` when their referent is not otherwise (strongly) reachable. The $\texttt{ptr}\langle\tau\rangle$ type is an untraced pointer type, which can be used to reference memory locations potentially visible to native programs outside Mu.[2]

The type system also includes a number of opaque reference types, referring to Mu-specific entities: `thread`, `stack`, $\texttt{func}\langle\tau_0^*; \tau_0\rangle$ referring to threads, stacks and functions, respectively.

A type of the form $\texttt{hybrid}\langle F; V\rangle$ is akin to a `struct` with a fixed prefix F, followed by an array of unspecified size having elements from (non-hybrid) type V. The size of the variable-length array part in a `hybrid` is specified at allocation time. We expect, for example, that most language clients would implement their string types with a `hybrid` type. Similarly, a Java client might represent Java arrays as a fixed prefix `int` holding the size of the array and a variable-length part for the payload.

All variables and memory locations in Mu must hold values with well-defined Mu types. This restriction eliminates the option of letting the client customize the object layout and identify references in objects as VMKit does [9]. However, the Mu type system is powerful enough to express complex high-level types using a combination of nested aggregate types including `struct`s, `array`s and `hybrid`s. Mu does not have a "union" type because a union of reference and value types will make a memory location ambiguous to the garbage collector

[1] In Mu, an *object* is defined as the unit of memory allocation in the heap. We are deliberately agnostic about the sorts of languages and type systems implemented by clients; our use of the term *object* does not presuppose any sort of object-orientation. From the client's perspective, objects are headerless.

[2] The `ptr` type is a feature planned to be added in the next version of Mu. It may not be visible in the Mu specification when this paper appears.

with respect to its contents holding a reference or not. However, Mu provides a *tagged* reference type `tagref64`. It uses several bits of a 64-bit word to indicate whether it currently holds an object reference, an integer or a `double`. In this way, exact garbage collection is still possible.

3.3 Intermediate Representation

The Mu IR is low-level and language-neutral. It is similar to the SSA-based LLVM IR [13]. This grounds our design, providing a reference against which each Mu IR design decision can be measured and audited with respect to Mu design principles.

The top-level unit of the Mu IR is a *code bundle*, which contains definitions of types, function signatures, constants, global cells and functions. A function has basic blocks and instructions. The Mu instruction set contains LLVM-like primitive arithmetic and logical instructions as well as Mu-specific garbage-collection-aware memory operations, thread/stack operations, and traps.

3.3.1 Basic Instructions

A Mu instruction can be very simple. For example, an `ADD` instruction "`%c = ADD <@i64> %a %b`" adds two numbers and a `SITOFP` instruction "`%r = SITOFP <@i64 @double> %x`" converts an integer to a floating point number, treating the integer operand as signed. For the convenience of the micro virtual machine rather than the client, the types of the operands are explicitly written as type arguments so that Mu does not need to infer the type of any instruction from the types of its operands.

3.3.2 Function Calls and Exception Handling

A `CALL` instruction "`%rv = CALL <@sig> @func (%arg1 %arg2)`" calls a Mu function.[3] Mu IR programs must explicitly truncate, extend, convert or cast the arguments to match the signature. Mu also provides a `TAILCALL` instruction which directly replaces the stack frame of the caller with a frame of the callee rather than pushing a new frame. The client must explicitly generate `TAILCALL` instructions to utilize this feature. Mu implementations need not automatically convert conventional `CALL`s into `TAILCALL`s, though an implementation might.

Mu has built-in exception handling primitives that do not depend on system libraries, unlike LLVM. The `THROW` instruction generates an exceptional transfer of control to the caller of the current function.[4] The exception is caught by the nearest caller's `CALL` instruction with an *exception clause* of the form "`CALL <@sig> @func (%arg) EXC(%nor %exc)`", which branches to the designated basic block where a `LANDINGPAD` instruction receives the exception value. Unlike LLVM, an exception in Mu is an arbitrary object reference. This kind of `CALL` unconditionally catches all exceptions and the return type of `LANDINGPAD` is `ref<void>`. The client is responsible for implementing its own exception hierarchy which can be complex (like Java's and Python's) or simple (like Lua's and Haskell's, where an error is simply a

[3] Calling a native function requires a foreign function interface (FFI) that is still under design.

[4] Exception handling within a function, for example, a `throw` statement in a `try-catch` block in Java, should be translated to branching instructions (`BRANCH` and `BRANCH2`) in the Mu IR. In this case, Mu is not aware of any exceptions being thrown.

string message). The client should generate Mu IR code to check the run-time type of the exception object, and decide whether to handle, re-throw or clean up the current context.[5]

3.3.3 Memory Operations

Support for precise (exact) garbage collection is integral to the design of the instruction set. Heap memory allocation is a primitive operation in Mu. The `NEW` and the `NEWHYBRID` instructions allocate fixed and variable-length objects in the heap, respectively, automatically managed by the garbage collector. Memory can also be dynamically allocated on stacks or statically allocated in the global memory.

To implement exact garbage collection, Mu must be able to identify all references into the Mu heap. The GC root set is precisely defined as all references in live local variables, stack memory, global memory, and those explicitly held by the client. Because all values in Mu come from the Mu type system, which never confuses references and untraced values, the micro virtual machine can perform garbage collection internally without client intervention.

3.3.4 Atomic Instructions and Concurrency

Mu is designed with multi-threading in mind. Mu has threads and a C11/C++11-like memory model, allowing annotation of memory operations with the desired memory ordering semantics. Mu threads may execute simultaneously. Like LLVM, Mu has no "atomic data types", but defines a set of primitive data types eligible for atomic accesses. The supported memory orders are `NOT_ATOMIC`, `RELAXED`, `CONSUME`, `ACQUIRE`, `RELEASE`, `ACQ_REL` (acquire and release) and `SEQ_CST` (sequentially consistent). This gives the client the freedom and responsibility to implement whatever memory model is imposed (or not) by the client language. For example, the very weak `CONSUME` order is essential in the efficient implementation of the read-copy-update (RCU) pattern which facilitates extremely low-overhead lock-free concurrent memory accesses. Outdated records left by RCU updates can be garbage-collected when unused, which has been a major difficulty in implementing RCU in the Linux kernel without GC.

Supporting relaxed memory models is not trivial. As a design principle, the client is trusted, and can to shoot itself in the foot. Abusing the memory model may result in program errors or even undefined behaviors. However, Mu does not force all users to understand the most subtle memory orders. A novice language-client implementer can exclusively use the `SEQ_CST` order even though the Mu implementation supports weaker orderings. Conversely, a conservative implementer of Mu itself can always correctly implement a stronger memory model than required, for example, implementing `CONSUME` as `ACQUIRE` or implementing all memory models as `SEQ_CST`, which will trade performance for simplicity and perhaps ease of verification of the micro virtual machine.

In addition to the standard C11-like atomic operations (such as compare-and-swap) Mu provides a futex-like [8] wait mechanism, as well as basic thread park/unpark operations. The client is responsible for implementing other shared-memory machinery such as blocking locks and semaphores. As these can be difficult and tedious to implement, they may be provided by client-level *libraries*, enabling complex implementations to be shared among multiple language clients.

---

[5] There is no `finally` in Mu, but it can be implemented as an unconditional catch followed by the actions in the `finally` block and another `THROW` instruction.

3.3.5 Stack Binding and the Swap-stack Operation

Unlike many language runtimes, Mu clearly distinguishes between threads (executors) and stacks (execution contexts).

A *thread* is an abstraction of a processor, while a *stack* is the context in which a thread runs. Each stack includes abstract execution state such as the program counter and the values of local variables in each frame. A thread is not permanently bound to any particular stack.

The `SWAPSTACK` instruction unbinds a thread from one context and rebinds it to another context [6]. When rebound, the thread continues from the corresponding instruction (usually another `SWAPSTACK`) where the destination context paused when last active (bound to a thread). This semantics directly provides an implementation of symmetric co-routines, which can in turn implement high-level language features including user-level "green" threads, the $m \times n$ threading model, generators, and one-shot continuations.

Dolan *et al.* [6] showed that this lightweight context switching mechanism can be implemented fully in user space with only a few instructions, so it is more efficient than native threads which inevitably involve transitioning through the kernel. Mu also assumes a client code generator that knows and can specify the liveness of variables at each `SWAPSTACK`, so only the needed registers are saved. This is impossible for library-based approaches, including `setjmp/longjmp`, `swapcontext` or customized assembly code, which have no information from a compiler and must conservatively save all registers.

The same stack binding and unbinding mechanisms are also used in other places besides the `SWAPSTACK` instruction, and many Mu design choices are based on this mechanism. Unbound threads and stacks are inactive. Only when a thread is bound to a stack context does it begin/resume executing from that context. Stack binding and unbinding are also used in trap handling, which are discussed below as part of the Mu client interface.

3.3.6 Traps and Watchpoints

A Mu IR program can temporarily pause execution and transfer control to the client by executing a `TRAP` instruction, which is trivial in Mu: "`%trap_name = TRAP <@RetTy>`".[6] Traps give clients the opportunity to introspect execution state to adapt and optimize the running program. The `WATCHPOINT` instruction is a conditional variant of `TRAP` which is disabled in the common case but can be enabled asynchronously by a client thread. `WATCHPOINT` is particularly useful for invalidating speculatively optimized code.

3.4 Client Interface

The *Mu client interface* is bi-directional. The client can send messages to Mu for the purposes of: (1) loading Mu IR code bundles into Mu, (2) accessing the Mu memory, and (3) introspecting and manipulating the state of Mu threads and stacks. Mu sends messages to the client if a `TRAP` or `WATCHPOINT` instruction is executed, or a declared but not defined function is called.

All Mu values, including references, may be indirectly exposed to the client via opaque handles tracked by Mu. This makes exact GC easy to implement because all externally held

[6] Lameed *et al.* [12] implemented something similar in LLVM, but it required non-trivial work in both the JIT compiler and the runtime library in order to achieve the same goal as the `TRAP` instruction in Mu. For compatibility reasons, they did not modify LLVM itself, so could not introduce new instructions. Since Mu is designed from scratch, it has no such restriction.

references are tracked. Copying and concurrent GC is also easier with a level of indirection. This design, which is also used by JNI and the Lua C API [10], also segregates the different type systems of Mu and the language the client is written in and, thus, makes the interface cleaner.

3.4.1 Bundle Loading and Code Redefinition

The client submits Mu IR code bundles to Mu via the interface. When a declared but not defined function is called, Mu will send a message to the client, which will be handled by a registered handler. The client should define the function by submitting another bundle containing the function definition. The client can also redefine existing functions. All existing call sites remain valid and Mu always calls the newest version of any function. This feature allows optimizing compilers to replace functions with (re-)optimized versions.

3.4.2 Traps and Stack Operations

Traps and watchpoints use the same stack binding mechanism as the `SWAPSTACK` instruction. When a `TRAP` or `WATCHPOINT` instruction is executed, the current thread is unbound from its current stack just before entering the registered trap handler in the client, leaving the stack in a clean state ready for introspection and on-stack replacement (OSR). During execution of the trap handler, the client can introspect the state of the stack frames, including the current function, the current instruction and the value of local variables. At the end of the trap handler, the client designates an unbound stack to which the original thread will be rebound. This stack does not need to be the same stack to which the thread was bound before the trap, so the thread may continue in a brand new context.

Some optimizations involve *on-stack replacement*: replacing a stack frame with a new frame of an optimized version of a function and continuing from the equivalent place it paused at. The Mu client interface supports this by providing two primitive operations: (1) popping a frame from a stack, and (2) making a new frame for a given function and its arguments on the top of a stack, and continuing from the beginning of that function. The client can emulate continuing from the middle of a function by inserting branches in the high-level language; a well-established approach [7].

3.4.3 Miscellaneous Operations

Many operations available as Mu IR instructions are intentionally duplicated to be used *via* the Mu client interface. The client can allocate and access the Mu memory via this interface, too. With an layer of indirection, this interface is designed for infrequent introspection. The client can also create threads and stacks, which is the proper way to start new Mu IR programs.

4 Building Mu Clients

The *client* is the program sitting on top of Mu, the micro virtual machine. It is the user of Mu and the implementer of the concrete high-level programming language, 'LVM' in Figure 1c. The Mu specification defines an interface for clients to manipulate and control the virtual machine, but does not impose any other requirements on the client. Here, we describe a number of strategies for building Mu clients, client-level libraries, and other higher-level client-specific abstractions.

4.1 Strategies for Building Clients

Mu supports a number of different approaches for implementing client languages. A client might dynamically compile to Mu IR just-in-time, compile to Mu IR ahead-of-time, or execute as an interpreter against either the Mu API or running itself as a Mu-coded client. We now discuss these options.

4.1.1 Just-in-time Compiling

The Mu client interface gives the client the power and responsibility to deliver Mu IR code to Mu. The most intuitive strategy is a compile-only approach, just-in-time compiling the higher-level language source code or byte code into the Mu IR, possibly *via* several extra layers of higher-level intermediate representations and optimizations. It us up to the client to decide how much optimisation it does to balance between compile time and code performance.

The generated Mu functions need not mirror high-level language functions. A tracing JIT compiler may deliver a recorded trace as a Mu function which may cross the boundary of many high-level functions, or may be a single loop within a high-level function.

4.1.2 Ahead-of-time Compiling

The Mu specification does not require code to be generated at run time. In fact, a valid implementation could ahead-of-time compile the high-level language program into the Mu IR before execution. In this way, the client merely loads Mu IR code from the disk and delivers it directly to Mu without further processing.

A valid full ahead-of-time implementation could also generate a single binary which behaves like an amalgamation of the micro virtual machine, a set of bundles, and a client which loads the bundles, starts with a particular Mu function and handles specific trap events. This approach is similar to the "boot image" of JikesRVM [1]. In this way, a Mu implementation of a client language might behave much like traditional ahead-of-time compilers.

4.1.3 Interpreting

For the ease of engineering, the first implementation of many languages is usually an interpreter. Although implementing the interpreter in the client and using the Mu heap *via* the Mu client interface is possible, it is not recommended because the interface is not designed for frequent lightweight calls, so will introduce an overhead.

One strategy would be to implement the interpreter itself in the Mu IR, or in any language that already compiles to the Mu IR. For example, if there is a client for Java running on Mu, then interpreters written in Java, including Jython and JRuby, will run on Mu. Currently we are working on translating RPython into Mu IR. The PyPy interpreter (written in RPython) will then run on Mu, as will other languages that have an implementation in RPython, including Prolog, Scheme and Erlang. Like Jython, this implementation strategy makes use of the concurrency and GC provided by the underlying VM, but does not directly utilize the underlying JIT compiler.

The Truffle/Graal project demonstrated that it is possible to generate a specializing JIT compiler from an interpreter by partial evaluation. In this way, the language implementer only needs to write an interpreter, but ultimately makes use of the JIT compiler.

4.2 Client-level Libraries

A micro virtual machine is minimalist, providing only a thin layer on which a language runtime can be built. Mu aims to deal with three of the most fundamental and conceptually challenging concerns within this layer. However the client is left with much to do. Rather than succumbing to the temptation to grow Mu, we adopt the principle of lifting additional features and amenities to client-level libraries, sitting above Mu. These may include helper features for generating SSA-form Mu IR, and Mu IR to Mu IR optimization libraries for common higher-level optimizations.

For example, there may be library support for dynamic languages that provides common specialization mechanisms, support for functional languages that handles higher-order functions, a library for concurrent languages which provides code snippets that implement synchronization primitives, *etc.* High-level frameworks, similar to RPython, Truffle, Terra or Lancet, can also be provided as libraries on the client side. In this way, the high-level language implementer can work on a much higher level, and does not always need to work with Mu directly.

4.3 Metacircular Clients

The client is just a program interacting with the micro virtual machine and, in theory, can be implemented in any language. It is possible to implement a *metacircular client* which runs as a Mu IR program inside the same Mu instance hosting other high-level programs. In this approach, the border between the client and the micro virtual machine is blurred. The client can directly refer to values or objects in Mu, and API messages can be implemented as simple function calls or SWAP-STACK operations. Special Mu IR instructions will give Mu IR programs access to the internals of Mu, including loading bundles, handling traps and introspecting stack states.[7]

4.4 Higher-level Abstraction

Mu is minimal, and does not provide any abstractions of classes, type hierarchies, methods, virtual dispatching, run-time type checking, strings, character encoding, higher-order functions, closures, events or message passing. The client is supposed to map its high-level language elements to the Mu IR. Two clients may map these elements differently even if they implement the same language.

Mu is designed to support multiple languages by minimizing semantic mismatching. However, unlike the .NET framework, Mu does not provide an abstract platform where programs in different languages can call each other. A client may implement such a platform by mapping different languages in a uniform way so that they can exchange data and make cross-language calls on the micro virtual machine level. The client can also implement a specific common intermediate language (like Java bytecode or the .NET CIL) above the Mu IR, and different languages can interoperate on that level. Mu understands only the Mu IR and remains oblivious of the actual high-level languages.

[7] Mu IR instructions to facilitate metacircular clients are still under consideration at the time of writing. Even without special instructions, such Mu IR-initiated introspection and controlling operations can be supported by traps and the assistance of a "conventional" client. This allows some Mu IR programs, such as an interpreter, to manage the state of Mu in a non-metacircular Mu implementation.

5 Related Work

5.1 Java Virtual Machines

The Java Virtual Machine (JVM) was originally designed for the Java programming language, but its portable Java Bytecode, clearly specified behaviors and performance attracted a wide range of language implementations to be hosted on the JVM, including Jython, JRuby, Scala, X10, *etc.*

This approach – reusing the existing JVM for new languages – bears several fundamental problems. The obvious one is the semantic gap between the new language and Java. The JVM implements many Java-specific semantics which are irrelevant to other languages. Working around them introduces overheads. On the other hand, Mu takes the lesson and is carefully designed to be language-agnostic at a much lower level to avoid the semantic gap.

Another problem is the JVM's lack of optimizations for languages other than Java. These optimizations include type inference and specialization, which are vital to dynamic languages [5], and inline caches for languages with dynamic dispatching. Mu, on the other hand, assumes most optimizations will be done by a client above it. For this reason, Mu exposes many low-level mechanisms, including vector instructions, on-stack replacement, swap-stack, tagged references, traps and watchpoints, to the client. Those mechanisms are either private or non-existent in the JVM, but they enable many advanced implementation techniques. For example, it is expensive to map Erlang processes to Java threads, which are usually mapped to native threads. In Mu, we believe stacks would work well in this role. Similarly, Mu's tagged reference type is a good candidate for implementing Lua's values, which have reference semantics, but refer to floating point numbers most of the time.

Besides, Mu aims to be a thin substrate while the JVM is a monolithic VM.

5.2 LLVM

The Low Level Virtual Machine (LLVM) [13] is a compiler framework including a collection of modular and reusable compilation and toolchain technologies. The LLVM compiler framework and its code representation (LLVM IR) together provide a combination of key capabilities that are important for language implementations. LLVM is the main reference according to which we designed Mu.

Mu also includes a number of significant differences from the LLVM. Firstly, Mu is designed to support managed languages while the LLVM is designed for C-like languages. Like C, the LLVM IR type system contains raw pointer types but not reference types. The LLVM does not provide a garbage collector, instead defining intrinsic functions including read/write barriers and yieldpoints, on top of which garbage collection must be implemented or inserted by the language frontend. Secondly, Mu is designed to be minimal while the LLVM is maximal. The LLVM tries to minimize the job of language frontends and include many optimization passes, while the Mu pushes as much work to the client as possible.

5.3 VMKit

VMKit [9] is a common substrate for the development of high-level *managed runtime environments* (client language runtimes), providing abstractions for concurrency, JIT-compiling and garbage collection. VMKit glues together three existing libraries: LLVM as the JIT compiler, MMTk as the memory manager, and POSIX threads for threading. The VMKit developers built two clients, for the CLI and JVM respectively as a proof of concept.

As the name suggests, VMKit is not a self-contained virtual machine, but a toolkit that provides incomplete abstractions over certain features. For example, VMKit leaves the object layout to be implemented by the client. As a consequence, the client runtime developed by the high-level language developer, must participate in object scanning and the identification of GC roots. VMKit's solution to concurrency is the POSIX Threads library, but threads cannot be implemented as a library [4], and require a carefully defined memory model involving both garbage collection and the JIT compiler.

Nonetheless, VMKit demonstrates that a toolkit that abstracts over the common key features can ease the burden of the development of language implementations, which is also part of the motivation for a micro virtual machine.

5.4 Others

Common Language Infrastructure

The Common Language Infrastructure (CLI) is Microsoft's counterpart to the JVM. Its Common Intermediate Language (CIL) is designed for several languages with similar level to VB.NET, C#, *etc.*, but also hosts many different languages including Managed C++, F# and JavaScript. The CLI shares similar problems to the JVM in that it is a monolithic VM and was designed for certain kinds of languages.

Truffle/Graal

Truffle and Graal [17] are reusable VM components for code execution developed by Oracle. Truffle serves as an AST interpreter with speculative execution and AST rewriting. Graal takes stabled Truffle AST and uses partial evaluation to JIT compile AST nodes. They aim to provide a reusable code execution engine for implementations of object-oriented languages. This goal sits on a much higher level than Mu.

6 Status and Future Work

During the early prototyping phase, we implemented a subset of Lua on a prototype micro virtual machine. Such a prototype could run simple Lua programs. It proved feasible to offload basic control flow analysis, including the conversion to the SSA form, up to the client. It also showed the usefulness of tagged references in the implementation of a dynamic language.

Currently we are experimenting with using Mu as a backend of RPython, the lower level of the PyPy project. Currently Mu can run simple RPython programs. This effort will eventually bring about a high-performance micro VM-based implementation of Python as well as other languages on RPython.

In the future, we will evaluate a diverse collection of languages, with Python, Haskell and Erlang having high priority, to test the ability of Mu to accommodate different languages. We will also bring transactional memory to Mu, which is known to solve the global interpreter lock (GIL) problem in Python and Ruby. Ultimately, we hope to produce a verified micro VM, perhaps combining it with the verified seL4 micro-kernel, thereby bringing us one step closer to a completely verified system with a managed runtime.

Mu abstracts over hardware. Although our immediate focus has been abstraction over ISAs and memory models, increasing hardware diversity invites a number of important future considerations. These include optimizing for energy as well as performance, abstracting over

single-ISA heterogeneous hardware, and more radical hardware heterogeneity such as GPUs and FPGAs. These are open topics for future research.

7 Conclusion

The current proliferation of new languages is evidence of a vibrant programming languages ecosystem. Unfortunately many languages are seriously inefficient, in terms of performance or maintenance, or both. We suggest that fundamental implementation challenges are a root cause of much of this inefficiency, and that a micro virtual machine may provide a solid foundation for the development of new languages. We describe a new micro virtual machine, Mu, developed over the past two years, with the specific goal of addressing this problem. Our hope is that micro virtual machines will change the way the next generation of languages are built, for the better.

References

1. Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Hummel Flynn, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'99, pages 314–324, Denver, Colorado, November 1999. doi: 10.1145/320384.320418.
2. Swift, 2015. `https://developer.apple.com/swift/`.
3. Shannon Behrens. Concurrency and Python. *Dr Dobb's*, February 2008. `http://www.drdobbs.com/open-source/concurrency-and-python/206103078`.
4. Hans-J. Boehm. Threads cannot be implemented as a library. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'05, pages 261–268, Chicago, Illinois, 2005. doi: 10.1145/1065010.1065042.
5. Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 195–212, Pittsburgh, Pennsylvania, 2012. doi: 10.1145/2398857.2384631.
6. Stephen Dolan, Servesh Muralidharan, and David Gregg. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization*, 9(4):36:1–36:25, January 2013. doi: 10.1145/2400682.2400695.
7. Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'03, pages 241–252, San Francisco, California, 2003. doi: 10.1109/CGO.2003.1191549.
8. Hubertus Franke and Rusty Russell. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, pages 479–495, Ottawa, Canada, June 2002. `http://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf`.
9. Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. VMKit: A substrate for managed runtime environments. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'10, pages 51–62, Pittsburgh, Pennsylvania, 2010. doi: 10.1145/1735997.1736006.
10. Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 2–1–2–26, San Diego, California, June 2007. doi: 10.1145/1238844.1238846.

11 Ivan Jibaja, Stephen M Blackburn, Mohammad R. Haghighat, and Kathryn S McKinley. Deferred gratification: Engineering for high performance garbage collection from the get go. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC'11, San Jose, California, June 2011. doi: 10.1145/1988915.1988930.

12 Nurudeen A. Lameed and Laurie J. Hendren. A modular approach to on-stack replacement in LLVM. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'13, pages 143–154, Houston, Texas, 2013. doi: 10.1145/2451512.2451541.

13 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, San Jose, California, March 2004. doi: 10.1109/CGO.2004.1281665.

14 The Micro Virtual Machine Project. `http://microvm.org/`.

15 Doc Bug #20993 Element value changes without asking. `https://bugs.php.net/bug.php?id=20993`.

16 Akihiko Tozawa, Michiaki Tatsubori, Tamiya Onodera, and Yasuhiko Minamide. Copy-on-write in the PHP language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'09, pages 200–212, Savannah, Georgia, 2009. doi: 10.1145/1480881.1480908.

17 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, Indianapolis, Indiana, 2013. doi: 10.1145/2509578.2509581.

18 Owen Yamauchi. On garbage collection. HipHop Virtual Machine for PHP. `http://www.hhvm.com/blog/431/on-garbage-collection`.