Hop, Skip, & Jump

Practical On-Stack Replacement for a Cross-Platform Language-Neutral VM

Kunshan Wang
Stephen M. Blackburn
Antony L. Hosking
Australian National University
kunshan.wang,steve.blackburn,antony.hosking
@anu.edu.au

Michael Norrish
Trustworthy Systems Group
Data61, CSIRO
Australia
michael.norrish
@data61.csiro.au

Abstract

On-stack replacement (OSR) is a performance-critical technology for many languages, especially dynamic languages. Conventional wisdom, apparent in JavaScript engines such as V8 and SpiderMonkey, is that OSR must be implemented in a low-level (i.e., in assembly) and language-specific way.

This paper presents an OSR abstraction based on Swapstack, materialized as the API for a low-level virtual machine, and shows how the abstraction of *resumption protocols* facilitates an elegant implementation of this API on real hardware. Using an experimental JavaScript implementation, we demonstrate that this API enables the language implementation to perform OSR without the need to deal with machine-level details. We also show that the API itself is implementable on concrete hardware. This work helps crystallize OSR abstractions and, by providing a reusable implementation, brings OSR within reach for more language implementers.

CCS Concepts • **Software and its engineering → Runtime environments**; *Just-in-time compilers*;

Keywords on-stack replacement, Swapstack, feedback-directed optimization, language implementation

ACM Reference Format:
Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2018. Hop, Skip, & Jump: Practical On-Stack Replacement for a Cross-Platform Language-Neutral VM. In *VEE '18: 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, March 24–25, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3186411.3186412

1 Introduction

On-stack replacement (OSR) is an important mechanism for high-performance language virtual machines (VMs). OSR is the 'open heart surgery' of just-in-time (JIT) compilation, allowing a compiled function to be replaced with a new version while it still has activations on thread stacks. This allows changes to code to become effective immediately, avoiding the need to wait for all related activations to complete before code can be replaced. Such a mechanism enables runtime optimization [2, 11], de-optimization for debugging [12], and recompilation of speculatively optimized code [9, 18].

The first contribution of this paper is the design of a practical, novel, platform-independent OSR API, allowing implementors to readily use OSR as a building block in language implementation. Key to the API is the insight that the Swapstack primitive [5] can be used to isolate the executing thread from the stack it is manipulating, eliminating the need to use tailored assembly code during OSR [12]. The API operates on a platform-independent, abstract view of stacks, so that the language implementation views the stack data in the abstract type system of the VM instead of byte-level stack layouts. The API is sufficiently abstract that it might be used in the implementation of any language wanting to use OSR. We demonstrate that the API can simplify the implementation of high-level languages using JS-Mu, an experimental JavaScript client for the Mu micro VM [22].

The second contribution of this paper is the realization of our OSR API on real hardware using an abstraction that allows for an elegant implementation. Derived from calling conventions, we introduce the concept of *resumption protocols*, which describe the machine-level rules to pass values to suspended frames and continue their execution. The correct implementation of the OSR API is therefore reduced to matching the *returning resumption protocol* with the *expected resumption protocol* between adjacent frames. We demonstrate that this API is implementable on concrete hardware using examples on the x64 and the AArch64 architectures.

With low-level details of OSR abstracted away, more language implementers will be able to keep their focus on high-level optimizations, leading to more high-performance language implementations.

The remainder of this paper is structured as follows. Section 2 introduces the background of Mu, OSR and Swapstack. Section 3 presents the details of the OSR API of the Mu VM. Section 4 demonstrates the API using an experimental JavaScript engine. Section 5 presents the abstractions that facilitate implementation of the API, and demonstrates that this API is implementable using examples on concrete architectures. Section 6 discusses related work about OSR. Section 7 offers conclusions and implications for future language implementations.

2 Background

The OSR abstractions and API presented in this paper were developed to support the Mu micro VM project, which we briefly introduce below. We also provide an overview of on-stack replacement implementations.

2.1 The Mu Micro Virtual Machine

The Mu micro virtual machine [21, 22] is inspired by the formal verification of the seL4 microkernel [13]. A micro virtual machine is a minimal, language-agnostic substrate that focuses only on the three major concerns that contribute to the difficulties of language implementation, namely dynamic 'just-in-time' (JIT) compilation, concurrency, and automatic memory management ('garbage collection' (GC)) [8, 22]. This design aims to provide a reliable low-level virtual machine to facilitate language implementation.

Like a microkernel which offloads the implementation of traditional operating system functions to user-land programs, a micro VM offloads the implementation of concrete languages to a separate layer—a *client*—that sits above the micro VM. The client is responsible for loading source code or bytecode, performing most optimizations[1] , and translating the program into the Mu intermediate representation (IR), which Mu JIT-compiles and executes. On the other hand, the Mu micro VM helps the client perform run-time optimization with trap instructions and the OSR API which we will introduce in this paper.

2.2 Threads and Stacks

In this work we distinguish between threads and stacks, because the difference is particularly important for on-stack replacement.

A *thread* is a flow of control that can progress concurrently with other threads. Modern operating systems provide threads (native threads) as kernel-level scheduling units that share resources with others in the same process,[2] and can be executed concurrently on parallel hardware. Programming languages, on the other hand, may implement language-level threads as native threads, strictly user-level 'green' threads mapped to a single kernel thread, or via an $M \times N$ mapping that multiplexes user-level over kernel-level threads.

The execution *context* of a thread includes the call stack as well as other thread-local state, such as a thread-local allocation buffer for GC. The *call stack*, or simply the *stack*, records the activations of dynamically nested function calls. Each activation record or *frame* on a stack records the values of local variables and the saved program counter of a function activation.

In conventional languages such as Java, each thread is only ever associated with one stack. More generally, however, a thread can switch among different stacks, such as when switching between coroutines, or handling UNIX signals.[3]

2.3 On-stack Replacement

Normally, frames are pushed when calling a function, and popped when returning. However, high-performance language implementations need to manipulate the stack in unconventional ways as part of their optimization and de-optimization processes.

Run-time optimization is usually *feedback-directed*—it detects frequently executed functions and loops using run-time profiling, and uses profile information to guide the optimization of those hot functions. For example, when a counter at a loop back-edge triggers re-compilation, the optimizer recompiles the function into a new, optimized, version of the function. The thread must then resume execution of the new code at the logically equivalent point, with a corresponding (possibly new) frame configured accordingly. Moreover, there may be other activations of the function with active frames spread across multiple stacks and active at different locations. Replacement of code for active functions in this way is known as *on-stack replacement* (OSR).

On-stack replacement was initially developed for the Self virtual machine for de-optimization [12] and optimization [11]. It is now a crucial part of many high-performance VMs.

Traditionally, 'on-stack replacement' has been used to refer to the entire process of getting execution states from stack frames, mapping the old execution context to the equivalent context of the newly compiled function, removing old frames, and creating new frames. In this paper, we use 'on-stack replacement' more narrowly to refer to the removal and replacement of stack frames, distinguishing it from the related but distinct task of getting execution states from frames which we refer to as *'stack introspection'*. We consider stack introspection as an orthogonal mechanism to OSR, because it can be used for purposes other than run-time re-compilation,

[1]Many of the optimizations which have the greatest impact on the performance of languages are language-specific. One example is specialization, as observed by Castanos et al. [3]. This is why Mu lets the client handle most optimizations. Low-level optimizations, such as register allocation, are still performed by Mu.

[2]Some operating system kernels, such as Linux or seL4, only provide the abstraction of 'tasks' rather than actual 'threads' and 'processes'. Tasks can implement threads, processes or other kinds of isolated containers, depending on what resources are shared among the tasks.

[3]The `sigaltstack` POSIX function can specify an alternative stack where the signal handler runs, instead of the regular user stack.

such as recording a stack trace for exception handling, and performing a security check by examining the call stack. Although this paper focuses on OSR, we will also introduce stack introspection mechanisms in Mu for completeness.

2.4 The Swapstack Operation

Swapstack is a context-switching operation that saves the execution state of the current thread's top-most activation on its active stack, switches the thread to use a different destination stack, and restores the thread's execution state to that of the the destination stack. The whole operation happens within user space without entering the OS kernel. Swapstack effectively provides the abstraction of symmetric coroutines. The term Swapstack was first used by Dolan et al. [5] to specifically refer to their efficient language-neutral compiler-assisted mechanism in LLVM for context-switching and message passing between lightweight threads of control ('green threads'). However, as we describe below, the ideas underlying Swapstack have been implemented before, including in the Self virtual machine [11, 12].

In this paper, we use the term Swapstack[4] in a broader sense—it is an abstract operation that rebinds the current thread to a different stack, regardless of its implementation.[5]

Swapstack is an integral part of Mu. It not only supports coroutines and green threads, but is also a foundation for many VM mechanisms, such as thread creation, trap handling, and OSR.

Swapstack and OSR are apparently orthogonal ideas. However, the Swapstack primitive can overcome some of the difficulties in implementing OSR. We now take a look at those difficulties, and see how Swapstack can address the problems.

2.5 OSR Without Swapstack

As is apparent from the widely used V8 [9] and SpiderMonkey [18] JavaScript engines, implementing OSR without Swapstack is difficult because of the need for implementers to write assembly-coded utility routines. This dependency on hand-written assembly code during OSR can be observed from both implementations' code for handling *de-optimization*, which occurs when an optimized function is invalidated and must be reverted to a baseline version.

Consider V8. During de-optimization, V8 first generates the contents of new stack frames in temporary buffers in the heap. Next, an assembly routine[6] uses a complicated two-level loop to copy the frame contents from the buffers to the actual stack. Since the stack pointer (SP) is constantly changing, the programmer must avoid any SP-relative addressing and carefully manage the register use in assembly. Even though V8 is mainly written in C++, this two-level loop cannot be written in C++ because the C++ compiler does not expect the stack pointer to be modified outside the generated code. Similarly, SpiderMonkey also copies stack contents from a side buffer using assembly code.[7]

Hand-writing assembly code of such complexity while properly managing stack layout is tedious and error-prone, and the code only works on one platform.

This problem could have been avoided if the virtual machines had access to a Swapstack mechanism. When a thread is modifying a stack that it is *not* executing on, the stack can be treated like binary data, and manipulated without worrying about the stack pointer of the *current* thread. Therefore, the OSR routines could have been implemented in any high-level language as long as it could access the relevant parts of memory. However both V8 and SpiderMonkey have only one stack per thread. Without Swapstack, the run-time system has to handle stack operations with care.

Having seen the problem with OSR without Swapstack, we now introduce the Swapstack-based OSR API that hides platform details from its user.

3 An API for Stack Operations

In this section, we describe our platform-independent API for stack operations, capable of supporting on-stack replacement. In Section 4 we will demonstrate usage of the the API through its application in a minimal JavaScript implementation. Finally, in Section 5, we show how the API can be implemented.

3.1 Overview

To perform OSR, the stack must first be unbound from the thread by performing a Swapstack operation, such as executing the `TRAP` Mu IR instruction.

To make the following discussion concrete, we use primitives from the Mu API. We list the relevant functions in Figure 1. The client uses the *frame cursor* API to iterate through stack frames, and uses the *stack introspection* API to get the execution context which guides the optimization or de-optimization. After the client generates a new function, it uses the OSR API to replace stack frames. The client then lets the program continue from the new frame with another

[4]Following the convention of Dolan et al. [5], we use Swapstack (Small Caps) to denote the abstract operation, and use `SWAPSTACK` (ALL CAPS) for the concrete instruction in the Mu instruction set.

[5]Therefore, in our terminology, the `Boost.Context` library [14], the (deprecated) `swapcontext` POSIX function, as well as the LLVM primitive created by Dolan et al. [5], are all implementations of Swapstack, although the work of Dolan et al. [5] usually out-performs others because it only saves as many registers as necessary. Mu also provides an implementation of Swapstack.

[6]See `src/x64/deoptimizer-x64.cc` line 136 in git revision `01590d660d6c8602b616a82816c4aea2a251be63` of V8. The source code is attached in Appendix A.1 for the convenience of the reader.

[7]See the `MacroAssembler::generateBailoutTail` function in the file `js/src/jit/MacroAssembler.cpp` in the current Mercurial revision `65b0ac174753` at the time of writing. See https://hg.mozilla.org/mozilla-central/file/65b0ac174753/js/src/jit/MacroAssembler.cpp#l1429.

Kind	API Function and Description
Frame Cursor	`FrameCursor* new_cursor(Stack* stack)` Create a new frame cursor pointing to the top frame of the given stack.
	`void next_frame(FrameCursor* cursor)` Move the frame cursor to the next frame, moving down the stack from called to caller.
	`void close_cursor(FrameCursor* cursor)` Destroy the cursor.
Introspection	`int cur_func(FrameCursor* cursor)` Return the function ID of the current frame.
	`int cur_func_ver(FrameCursor* cursor)` Return the function version[8] ID of the current frame.
	`int cur_inst(FrameCursor* cursor)` Return the instruction ID of the current frame.
	`void dump_keepalives(FrameCursor* cursor, MuValue values[])` Dump the values of all keep-alive variables of the current instruction of the current frame.
OSR	`void pop_frames_to(FrameCursor* cursor)` Remove all frames above the current frame of the given frame cursor.
	`void push_frame(FrameCursor* cursor, void (*func)())` Create a new frame on the top of the stack pointed by the frame cursor.

Figure 1. Summary of Mu API functions related to stack introspection and OSR.

SWAPSTACK operation, allowing it to return from the trap handler.

The usage can be summarized as 'hop, skip and jump'—hopping away from the stack, skipping several frames to create new frames, then jumping back to the stack.

Before introducing the instructions and the API, we present an abstract view of the stack that forms the foundation of the API.

3.2 Abstract View of Stack Frames

In this paper we adopt the convention of stacks growing up. The 'top' frame is the most recently pushed frame, and is near the top of the page in diagrams.

A stack consists of one or more frames. A frame contains the states of a function activation. A frame is *active* if it is the top frame of a stack bound to a thread. Otherwise, the frame is *inactive* because the code using it is not being executed. Specifically, if one function calls another function, the frame of the caller is stopped, expecting a value from the callee as a return value.

An inactive frame can receive a value of an expected type, and become active again. Specifically, when a function returns, its caller's frame receives the return value from the callee, and continues execution. Therefore, every inactive frame is *expecting a value*, and will eventually *return a value* to its caller. Symbolically, we write

$$frm : (E) \to (R)$$

to denote that the frame *frm* is expecting a value of type E in order to resume, and itself returns a value of type R. We can generalize this to multiple return values, writing:

$$frm : (E_1, E_2, \ldots) \to (R_1, R_2, \ldots)$$

We call this the *expect/return type notation.*

For example, in Figure 2, `foo` calls `bar`, `bar` calls `baz`, and `baz` calls `moo`. The expected type and the return type of the frames of `foo`, `bar` and `baz` appear in Figure 2(b) in expect/return type notation. As we can see, the expected type of a frame is determined by the call site (and the callee), and the function signature of a frame determines its return type.

A stack is *return-type consistent* if the return type(s) of every frame matches the expected type(s) of the frame below. It is obvious that if all stack frames are created by function calls, the stack is always return-type consistent.[9] However, the OSR API can create stack frames of arbitrary expected and return types. Therefore, the *client* must take care to ensure that the stack is return-type consistent at all times.

With our abstract stack view in mind, we now introduce the operations in the API.

3.3 Frame Cursor Abstraction

A *frame cursor* is an iterator of stack frames. A frame cursor always points to one frame at any time, and can move from top to bottom frame by frame. The API for both introspection and OSR depends on frame cursors.

[8]In Mu, the client can redefine a function, giving it a new function body (i.e., version) to be executed in subsequent invocations.

[9]In dynamic languages a function can return a value of any type—they do not enforce return-type consistency. However, Mu IR is statically typed. When implementing dynamic languages like Python, the Mu-level return type should be the most general type, such as `PyObject`, and all Python frames should have (`PyObject`) → (`PyObject`), which is always return-type consistent with respect to the Mu type system.

```
long moo();

long baz() {
    long x = moo();    // stop here
    return x;
}

double bar() {
    long x = baz();    // stop here
    double y = (double)(x + 1);
    return y;
}

int foo() {
    double y = bar(); // stop here
    int z = printf("%lf\n", y);
    return z;
}
```

(a) Example Code

baz : (long) → (long)

bar : (long) → (double)

foo : (double) → (int)

(b) Expected and Return Types

Figure 2. Example of nested calls. The expected types are determined by the call sites, and the return types are determined by the functions' return types. The return type of each frame must match the expected type of its caller's frame.

3.4 The Swapstack Operation

A stack bound to a thread always has its top frame active and other frames inactive, because the thread executes on its top frame. Swapstack [5] *unbinds* a thread from its stack, and *rebinds* it to another stack. Swapstack deactivates the top frame of its old stack, and reactivates the top frame of the destination stack, optionally passing one or more values. After the top frame of the origin stack becomes inactive, the frame waits for another Swapstack operation to bind a thread (any thread, not necessarily the original thread) to it and reactivate its top frame, optionally receiving one or more values.

It is easy to observe that an inactive frame stopping on a Swapstack site is similar to an inactive frame stopping at a call site. Both of them are expecting values, and can be reactivated by receiving values. The only difference is whether the values are received by returning or Swapstack. Therefore, the expect/return type notation $frm : (E) \to (R)$ is still applicable for Swapstack, where E is the type of the value expected from the incoming Swapstack operation.

In the Mu instruction set, the TRAP and the WATCHPOINT instructions perform a Swapstack operation.[10]

The TRAP instruction rebinds a Mu thread to a client stack, where a client-provided *trap handler* is activated. TRAP lets the client handle events which Mu cannot handle internally, such as loading programs on demand, and optimizing hot functions detected at run time. WATCHPOINT is a special kind of TRAP that can be turned on and off asynchronously. It is usually used to guard speculative code, such as a devirtualization optimization which can be invalidated by class loading.[11]

The Swapstack operation deactivates all frames of an unbound stack, making it ready for introspection and manipulation. All API functions related to stacks require the stack to be in the unbound (inactive) state. We now proceed to the introspection mechanisms before moving on to OSR.

3.5 Stack Introspection

As introduced in Section 2.3, the client needs to extract the execution state, including the program counter and the values of local variables, to guide optimisation and de-optimisation.

As shown in Figure 1, the cur_func, cur_func_ver and cur_inst report the current code position of a frame.

In Mu IR, all call sites (the CALL instruction) and Swapstack sites (the SWAPSTACK, TRAP and WATCHPOINT instructions) may have a *keep-alive clause* that specifies which variables are eligible for introspection. Consider the following snippet:

```
[%trap1] TRAP <> KEEPALIVE (%v1 %v2 %v3)
```

When the TRAP instruction %trap1 executes, local variables %v1, %v2 and %v3 are kept alive in the frame, and their values can be introspected using the dump_keepalives API function. Other local variables are not guaranteed to be live, which leaves Mu much room for machine-level optimization.

We let the client decide what variables are introspectable. The client, which compiled the high-level language into Mu IR, has full knowledge about what variables are relevant for the desired kind of run-time re-compilation, such as optimization or de-optimization. In the extreme, the client can retain all local variables, thereby preserving full information about the execution.

We now introduce the way the API allows modification of the stack.

[10]There is also a separate SWAPSTACK instruction (written in ALL CAPS like all Mu instructions), which rebinds a thread from one Mu stack to another Mu stack, and is useful for implementing coroutines. We do not discuss it further because it is not directly related to OSR.

[11]An aggressive optimizer can replace virtual calls with non-virtual calls (i.e., de-virtualization), provided that the virtual function is never overridden in any class. This optimization is speculative because new classes loaded at run time can override the method, making the assumption of 'never overridden' invalid. If this happens, functions that include the speculatively de-virtualized calls must be recompiled.

3.6 Removing Frames

The pop_frames_to API function removes all frames above the current frame. This will expose the current frame, an inactive frame below the top frame, to the stack top. Remember that the Swapstack operation passes values to the destination stack's top frame. When a thread rebinds to this stack using Swapstack, it will reactivate the current frame which is stopping at a call site instead of a Swapstack site. The values passed via Swapstack will be received by the frame as if the values were the return values from the call site's original callee.

Popping frames will lose information about the removed frames. The client should use dump_keepalives to save the execution states before popping frames if needed.

The pop_frames_to API can only remove frames above the specified frame. If the client desires to replace a frame when the frame is re-entered, usually due to de-optimization, the client should insert the WATCHPOINT instruction into the guarded function.

3.7 Creating New Frames Using Return-oriented Programming

The push_frame function pushes a frame for a given function onto the top of a stack. The frame stops at the entry point of the function.

Our approach to creating new frames is based on *return-oriented programming* (ROP).[12] We define a *ROP frame* to be a stack frame that is stopped at the entry point of a function: its return address is the entry point of the function. In contrast, the return address of a normal frame is the next instruction after a call site or Swapstack site. By definition, frames created by the push_frame API function are ROP frames.

When a ROP frame resumes, it receives the values returned by the frame above it, or passed during a Swapstack operation, as the arguments of its function which now executes from the entry point.

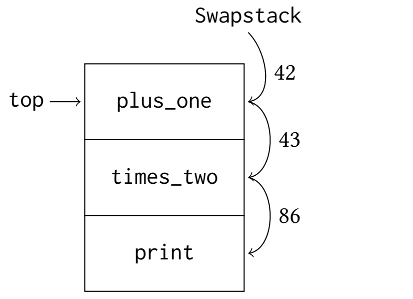
Consider the code snippet in Figure 3. If the client has pushed three ROP frames on a stack for the three functions respectively, in the order of print, times_two and then plus_one, then plus_one will be on the top of the stack and print at the bottom. When a subsequent Swapstack reactivates the stack, passing the value 42 to the top frame, then the top frame executes as if it was a call to plus_one(42). It returns 43, transferring control to an activation of times_two with the argument y receiving the value 43. This returns 86, transferring control to the activation of print with the formal parameter z bound to this value. This prints 86 to standard output and exits.

```
int plus_one(int x) {
    return x + 1;
}

int times_two(int y) {
    return y * 2;
}

void print(int z) {
    printf("%d\n", z);
    exit(0);
}
```

(a) C-like Pseudocode



(b) Stack Structure

Figure 3. ROP example. These three functions do not call each other. But if the run-time pushes three ROP frames for print, times_two and plus_one, respectively, the three functions will execute one after another from the top of the stack to the bottom (in the order of plus_one, times_two and then print). Each function passes its return value as the argument of the next function, making it a pipeline or, in cyber-security terms, a ROP chain.

Just like frames created by function calls, ROP frames also expect values and eventually return values. We can describe the stack in the previous example in the expect/return type notation as:

$$plus\_one : (int) \to (int)$$
$$times\_two : (int) \to (int)$$
$$print : (int) \to ()$$

Unlike frames stopped at call sites and Swapstack sites, the *parameter types* of the stopped function determine the expected types of ROP frames, as opposed to the call site. Users of the API can push frames for any functions, and as many frames as they desire, as long as the stack remains *return-type consistent* as defined in Section 3.2.

[12]ROP originates from the field of computer security to describe a particular attack technique [19]. The attack uses malicious data to cause a buffer overflow on the stack, overwriting existing stack frames to create new stack frames. The return address of each frame is set to the *entry point* of the next function to execute. Therefore, after each function returns, the processor will execute the next function specified by its return address. This lets the attacker drive control flow using return instructions to transfer control to the next function, hence the name. This chain of frames is called the *ROP chain.*

The pushed new frame is always a ROP frame which starts at the function entry point. However, in the typical run-time re-compilation scenario, the new frame must resume at the program counter equivalent to where the old frame stopped. Fink and Qian [7] solved this problem by inserting assignments and a jump instruction at the beginning of the new function, as follows:

```
  %var1 = %oldVar1
  %var2 = %oldVar2
  ...
  %varN = %oldVarN
  JUMP %cont
  ...
%cont:
  ...
```

D'Elia and Demetrescu [4] call these instructions '*compensation code*'. The assignments set local variables of the new frame to their values in the old frame, and the jump instruction jumps to the equivalent PC. Recall that the `cur_inst` and the `dump_keepalives` API gives the client the old PC and variable values. Therefore, while the Mu API requires the new frame to start at the entry point, the client still has all the information and capability to transition the old frame state to the equivalent new state.

4 Demonstration of the OSR API

To demonstrate the utility of the OSR API as an aid to language implementers, we built JS-Mu,[13] a prototype JavaScript client. It implements a small subset of JS, including operators such as the JS addition operator '+', which applies to both numbers and strings. Such dynamism gives the specializer a chance to showcase speculative optimization that requires OSR. We implemented JS-Mu in Scala. Examples in this section are modified to match the current version of the Mu API because Mu has evolved since JS-Mu was developed.

Like SpiderMonkey and V8, the JS-Mu execution engine consists of a baseline compiler and an optimizing compiler. There is no JS interpreter. The baseline compiler plays the role of the lowest-tier execution engine, while the optimizing compiler will optimize hot functions and loops.

4.1 Baseline Compilation and Trap Placement

The JS-Mu baseline compiler does not attempt to infer the concrete types of JS variables. All JS variables are represented as tagged references [10]. All operations, such as addition, subtraction, etc., accept all types of values, and raise type errors at run time.

We use counters to detect hot loops at loop headers.

```
%header(...):
  ...
  %c0 = LOAD <@i64> @COUNTER
  %c1 = ADD  <@i64> %c0 @CONST_1
  STORE <@i64> @COUNTER %c1
  %hot = SGE <@i64> %c1 @THRESHOLD
  BRANCH2 %hot %body(...) %trapbb(...)

%trapbb(...):
  [%trap1] TRAP <> KEEPALIVE (%v1 %v2 ...)
```

We use Mu IR code to increment the counter, and execute the `TRAP` instruction when the counter reaches a threshold. The `KEEPALIVE` clause annotates local variables for introspection. The client maintains a simple `HashMap` to record the AST node and compiler metadata relevant to each `TRAP`.

```
class TrapInfo(val blFunc: BaselineFunction,
  val node: Node,    // AST node
  val headBB: MuBB, val trapBB: MuBB)
val trapInfoMap =
  new HashMap[String, TrapInfo]()
```

4.2 Optimization and On-stack Replacement

The `TRAP` instruction transfers control to the trap handler.

```
def handleTrap(ctx: Context, st: MuStackRefValue,
    ... ): TrapHandlerResult {
  val cursor = ctx.newCursor(st)
  val inst = ctx.curInst(cursor)
  val localVars = ctx.dumpKeepalives(cursor)
  val trapInfo = trapInfoMap(nameOf(inst))
  ...
```

The trap handler uses `cur_inst` to identify the executed `TRAP`, and uses `dump_keepalives` to recover the current values of local variables. Using the Mu-level instruction ID and the `HashMap` described above, the optimizer finds the high level implementation of the hot loop.

The main optimization performed is specialization, which is crucial to dynamic languages [3]. Using the type information encoded in tagged references, the optimizer infers the types of JS variables, lowering the types to more concrete types and eliminating run-time type checking operations where possible. After specialization, the client generates Mu IR code, and loads the code into the micro VM. As we described in the end of Section 3.7, the optimized function takes the old local variables as parameters, and uses assignments and a jump to transfer to the equivalent code point[14] [4, 7].

After JIT compilation, the client uses `pop_frames_to` and `push_frame` to replace the stack frame.

```
  ...
  val newFunc = compileFunction(...)
  ctx.nextFrame(cursor)
  ctx.popFramesTo(cursor)
  ctx.pushFrame(cursor, newFunc)
  ctx.closeCursor(cursor)
```

[13] Source code: https://gitlab.anu.edu.au/mu/obsolete-js-mu

[14] See Appendix A.2 for a concrete snippet.

```
    Rebind(st, PassValues(localVars))
```

When the client returns from the trap handler, it uses SWAPSTACK to rebind the current thread to the old stack, passing the old values of local variables. The JS application resumes execution, executing the optimized version, continuing with the equivalent state to that at the time the optimization was triggered.

4.3 Results

Appendix A.2 gives a concrete example of a simple JS function, JIT-compiled during OSR triggered at a hot loop. The OSR API provided sufficient type information that the optimized Mu IR code for the tight loop is almost equivalent to the LLVM IR code that Clang might have generated from an equivalent C program with static types.

JS-Mu is built on a proof-of-concept reference implementation [20] of Mu which is unsuitable for performance evaluation. However, the implementation is sufficient to demonstrate the completeness and correctness of the API. Note that the OSR mechanism itself is *not* performance critical, since it executes just once for each recompilation, which will be dominated (by many orders of magnitude) by the subsequent execution of the optimized code in any typical OSR setting.

5 Implementing the OSR API

We have introduced the platform-independent OSR API. However, the API does not remove the fundamental complexity of OSR. Rather, it hides it beneath a layer of abstraction.

In this section we turn to the question of whether such an API is realizable in a realistic setting. Our approach is to introduce the abstractions of *resumption protocols* and *resumption points*. We use the x64[15] and the AArch64[16] architectures as our concrete setting, but the ideas are not specific to those architectures.

To demonstrate how resumption protocols can guide the implementation of the OSR API in a more realistic scenario, we developed another proof-of-concept project `libyugong`[17] that implements the SWAPSTACK operation and the OSR API for native programs (in C, C++, LLVM, etc.) which follow the platform ABI on GNU/Linux.

5.1 Frame Cursors and Introspection

For completeness, before we start discussing OSR, we briefly introduce how frame cursors and stack introspection can be implemented. Although they are integral parts of the API, they are well-developed technologies, and this paper does not attempt to make improvements over existing approaches.

[15]Also known as AMD64, x86-64 or Intel64, an extension to the IA32 instruction set architecture.

[16]AArch64 is the 64-bit execution mode of the ARMv8 architecture. The instruction set is called A64.

[17]Source code: https://gitlab.anu.edu.au/kunshanwang/libyugong

The frame cursor is an abstraction over *stack unwinding*, the process of restoring register states of frames below the top frame of a stack. Key to the implementation is how to restore callee-saved registers of the caller given the program counter. C++ compilers, such as GCC, generate stack unwinding metadata in the DWARF [6] format on GNU/Linux for exception handling.

Stack introspection uses *stack maps*, a data structure that maps machine-level execution states (including stack frame contents and callee-saved register values) to high-level language states (values of local variables). Stack maps are required for exact garbage collection, therefore many virtual machines, such as JikesRVM [1], already implement stack maps. LLVM also provides the '`statepoint`' intrinsic which generates stack maps to decode frames for LLVM-level variable values.

The rest of this section assumes these techniques are readily available, and focuses on OSR on top of those techniques.

5.2 Resumption Point

We define a *resumption point* as the point in the function body where an inactive frame stopped. In Mu, a resumption point can be a *call site*, a *swap-stack site*, or the *entry point* of a function (ROP frame).

The resumption point is an internal execution state not visible to its neighboring frames, while the expected type and returned type are the 'interface' through which the frame communicates with the frames above and below. However, the resumption point determines the frame's resumption protocol which we now define.

5.3 Resumption Protocol

The concept of resumption protocol is related to calling conventions. A *calling convention* describes the rules of function calling at the machine level, including the responsibility to set up and tear down the frames, the registers and stack locations used to pass parameters and return values, and the set of registers preserved across function calls (i.e., the callee-saved registers). The calling convention is the agreement between the caller and the callee.

However, we are more interested in the resumption of frames than the set-up of frames. We define the *resumption protocol* as the machine-level rules governing the passing of values to an inactive frame to resume its execution.

The concrete rules are determined by the resumption points. We now describe the resumption protocols of each resumption point.

5.3.1 Resumption at Call Sites

When a frame is stoped at a call site, the resumption protocol is the 'returning' part of the calling convention.

For example, consider a function that is suspended, having called a function that returns a 32-bit integer. On x64 on GNU/Linux, the resumption protocol is:

> 'Move the return value into register `EAX`, and then pop and jump to the return address at the top of the stack.'

The resumption protocol on AArch64 is:

> 'Move the return value into register `w0`, and then restore the program counter from the link register `x30`.'

We use '`CCC_Ret(int)`' (C Calling Convention: Returning) as the symbolic notation for the resumption protocol. We can also generalize it to '`CCC_Ret(T)`' for the protocol of returning type `T` using the C calling convention. We omit the platform name in the symbol, because the C calling convention refers to the definition in the ABI of the platform.

At a call site, the function receives the return value from the callee according to the callee's calling convention; when the function itself returns, it will pass the return value to its caller according to the function's own calling convention. Therefore, every frame has both an *expected resumption protocol* which is the resumption protocol to re-activate the frame itself, and a *returned resumption protocol* which is used to re-active its caller. We use the following notation:

$$frm : \mathtt{rp}_1\,(E_1, E_2, \ldots) \rightarrow \mathtt{rp}_2\,(R_1, R_2, \ldots)$$

to denote that the frame *frm* itself is resumed using the resumption protocol $rp_1\,(E_1, E_2, \ldots)$, and will re-activate its caller using the resumption protocol $rp_2\,(R_1, R_2, \ldots)$ when it returns. We call this notation the *expect/return protocol notation* because it involves not only the types but also the resumption protocols.

Consider the example in Figure 2. The expected and returned protocols of `baz`, `bar` and `foo` are straightforward:

$$baz : \mathtt{CCC\_Ret}\,(long) \rightarrow \mathtt{CCC\_Ret}\,(long)$$
$$bar : \mathtt{CCC\_Ret}\,(long) \rightarrow \mathtt{CCC\_Ret}\,(double)$$
$$foo : \mathtt{CCC\_Ret}\,(double) \rightarrow \mathtt{CCC\_Ret}\,(int)$$

because both `baz`, `bar` and `foo` are stopped on call sites.

We now proceed to describe the resumption protocol for Swapstack sites.

5.3.2 Resumption at Swapstack Sites

Recall that the Swapstack operation unbinds the thread from one stack and rebinds it to another stack. Similar to function calls, Swapstack involves preserving the context and transferring control. Currently, there are no widely-applicable standards about the implementation of Swapstack. However, when implementing Swapstack, there must not be any 'swappee-saved'[18] registers because it is unpredictable where the swappee stack will swap back to the swapper, or whether it will swap back from that stack at all.[19] Therefore, the Swapstack operation must treat all machine registers as swapper-saved registers.

[18]Similar to 'callee' which means the function called by a call site, we use the word 'swappee' for the destination stack in a Swapstack operation. The original stack of a Swapstack operation is called the 'swapper'.

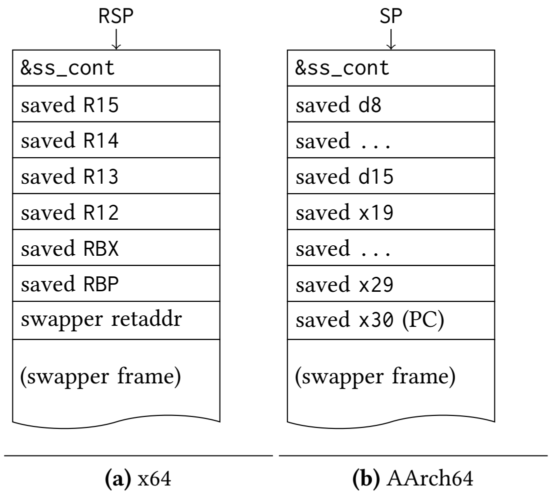


Figure 4. Stack-top Structure of Unbound Stacks in `libyugong`. The callee-saved register values and the resumption point PC are all saved at the top of the stack. When rebinding a thread to an unbound stack, it continues from the address of the `ss_cont` routine which restores the callee-saved registers from the stack top, and returns to the swapper frame.

We use the symbolic notation $\mathtt{SS}\,(T_1, T_2, \ldots)$ for the resumption protocol of a Swapstack site that receives a value of type $T_1, T_2, \ldots$ when the stack is rebound.

This protocol can be implemented in many ways. In `libyugong`, we implemented Swapstack similar to `boost-context` [14]. The Swapstack implementation (See Appendix A.3) saves the execution state on the top of the unbound stack as in Figure 4. Therefore, resuming the swapper frame is done by restoring the callee-saved registers from the stack top structure, and jumping to the resumption point of the swapper.

5.3.3 Resumption at Entry Points

Remember that a ROP frame is stopped at the entry point of a function, expecting to receive values as its parameters. Therefore, the resumption point for ROP frames is the same as the 'calling' part of a calling convention.

For example, consider a function that takes one 32-bit integer as its parameter. On x64 on GNU/Linux, the calling convention specifies that:

[19]Intuitively, if we implement a green thread system using each stack as a light-weight task and randomly schedule the stacks to a pool of threads, then when a task yields using the Swapstack operation, it is unpredictable which task will be executed next, and which task the thread is swapping from. In general, Swapstack is much less predictable than call and return.

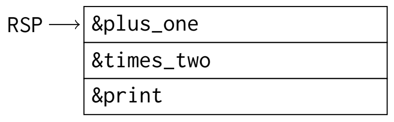


Figure 5. Stack structure of a naïve (and thus, incorrect) ROP frame implementation on x64 for the example in Figure 3. Each frame simply consists of the address of the entry point of the function. This approach will execute `plus_one`, `times_two` and `print` in order, but *will not* be able to pass return values between frames because the return values and the parameters are passed via different registers (`EAX` and `EDI`, respectively).

> 'The first 32-bit integer parameter is passed via the `EDI` register, and the return address is on the top of the stack.'

The calling convention on AArch64 is:

> 'The first 32-bit integer parameter is passed via the `w0` register, and the return address is held in the link register `x30`.'

Therefore, as long as the arguments and the return address are put in the right place, and the stack pointer is set properly, the function will start executing until it returns. We use the symbol `CCC_Entry` for the resumption protocols of ROP frames that follow the C calling convention, and `CCC_Entry(int)` denotes ROP frames that have `int` as their sole parameter.

We have introduced all three resumption points and their resumption protocols. A stack is *return-protocol consistent* if the returned protocol of every frame matches the expected protocol of the frame below.

However, it remains a question how to construct concrete ROP frames on the stack. Figure 5 shows a naïve ROP frame implementation on x64 for the example given in Figure 3. Each frame simply consists of the address of the entry point of the function, and the `RSP` register points directly at the location that holds the return address of the topmost function, `plus_one`. When the thread 'returns' by popping the return address from the stack, it will start execution from the entry point of `plus_one`. When `plus_one` returns, it will 'return' to `times_two` which will in turn start from its entry point. Eventually `print` will be executed, too.

But this naïve approach cannot pass the return value of one function to another because of the difference of register use between return values and parameters. Note that on x64, `CCC_Ret` and `CCC_Entry` expect integer values to be passed in different registers (`RAX` and `RDI`, respectively). Therefore, when the function of the next ROP frame starts, it will not find the parameter in the `EDI` register, and will not be able to receive the return value from the frame above.

Since all functions are compiled to use the `CCC_Ret` protocol for normal returning, the returned protocols of these functions are all `CCC_RET`. Using the protocol-sensitive notation, we have:

$$\begin{aligned} \textit{plus\_one} &: \texttt{CCC\_Entry}\,(\textit{int}) \to \texttt{CCC\_Ret}\,(\textit{int}) \\ \textit{times\_two} &: \texttt{CCC\_Entry}\,(\textit{int}) \to \texttt{CCC\_Ret}\,(\textit{int}) \\ \textit{print} &: \texttt{CCC\_Entry}\,(\textit{int}) \to \texttt{CCC\_Ret}\,() \end{aligned}$$

The stack is not return-protocol consistent. The returned types match the expected types (`int` and `int`), but the resumption protocols (`CCC_Entry` and `CCC_Ret`) do not. This is the reason why this naïve approach will not work. To correctly implement ROP frames, we need adapters to convert mismatching resumption protocols.

5.4 Adapter Frames

An *adapter frame* sits between two frames where the *type* of the passed values match, but the *resumption protocols* do not. The adapter frame satisfies:

$$\textit{adapter} : \texttt{rp}_1\,(T_1, T_2, \ldots) \to \texttt{rp}_2\,(T_1, T_2, \ldots)$$

for some $T_1, T_2, \ldots$, that is it converts one resumption protocol to the other while preserving the value.

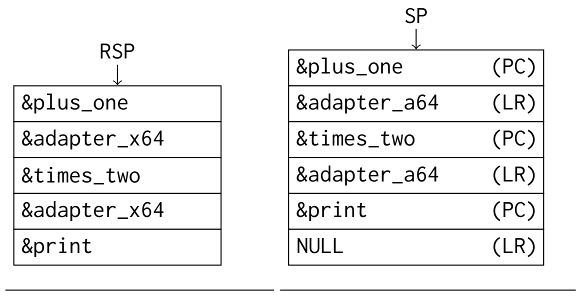
The following snippet implements an adapter frame on x64 of `CCC_Ret(int)` → `CCC_Entry(int)`, that is a frame that transfers the return value to the register which holds the parameter.

```
adapter_x64:
    MOV     EDI, EAX
    RET
```

The `MOV` instruction moves the value to the correct register, and the `RET` instruction resumes the next frame.

Figure 6 shows a correct implementation for the example in Figure 3. On x64, adapter frames are inserted between adjacent ROP frames. Each adapter frame consists of only the address of the `adapter_x64` assembly routine shown above. When `plus_one` returns, the return value is held in the `EAX` register, and the control flow jumps to the `adapter_x64`, where the `MOV` instruction moves the value from `EAX` to `EDI`. At this point, the address of the `times_two` function is at the top of the stack. Therefore, the `RET` instruction in the adapter frame resumes the `times_two` function, and the return value of `plus_one` has already been moved to the expected `EDI` register which `times_two` receives as the argument. This process will happen again when `times_two` returns, and `print` will eventually receive and print out the correct value.

AArch64 faces a different challenge. The return address of the `CCC_Entry` protocol is in the link register x30 instead of on the stack, therefore we cannot put the address of the return address on the stack and expect the ROP frame to automatically return into it. We use the adapter to restore the link register.



(a) x64 **(b)** AArch64

Figure 6. Correct ROP frame implementation with adapter frames on x64 and AArch64 for the example in Figure 3. On x64, addresses of functions are interleaved with addresses of the adapter frame. When one function returns, the adapter will be executed before the next function starts, giving the adapter a chance to move the return value to the argument register. On AArch64, every pair of addresses from the top are restored into the x30 (LR) and the PC registers respectively, allowing the ROP function

```
adapter_a64:
    LDP     x9, x30, [sp], 16
    BR      x9
```

The above code pops the ROP function address into the PC, and pops the return address of the ROP frame into x30 (LR). It used a scratch register x9 because we cannot directly load into the PC. After `plus_one` returns, it returns into x30 which holds the address of `adapter_a64`, which restores the next ROP frame. The last function, `print`, does not return, therefore x30 holds `NULL` when entering `print`.

Using the expect/return protocol notation, we have:

$$\begin{aligned} \mathit{plus\_one} &: \texttt{CCC\_Entry}\,(\mathit{int}) \to \texttt{CCC\_Ret}\,(\mathit{int}) \\ \mathit{adapter} &: \texttt{CCC\_Ret}\,(\mathit{int}) \to \texttt{CCC\_Entry}\,(\mathit{int}) \\ \mathit{times\_two} &: \texttt{CCC\_Entry}\,(\mathit{int}) \to \texttt{CCC\_Ret}\,(\mathit{int}) \\ \mathit{adapter} &: \texttt{CCC\_Ret}\,(\mathit{int}) \to \texttt{CCC\_Entry}\,(\mathit{int}) \\ \mathit{print} &: \texttt{CCC\_Entry}\,(\mathit{int}) \to \texttt{CCC\_Ret}\,() \end{aligned}$$

The returned protocol of each frame matches the expected protocol of the frame below, therefore the stack state is now return-protocol consistent.

Different types need different adapter frames. For example, on x64 on GNU/Linux, both the first floating point parameter and the floating point return value are held in the `XMM0` register. Apparently no register movement is necessary. But the ABI also has a 16-byte stack alignment rule at the entry point of every function.Thus a frame for a trivial adapter routine such as:

```
    RET
```

can be used to 'pad' the size of the one-word (8 bytes on x64) ROP frame to a multiple of 16 bytes in order to meet the alignment requirement.

One adapter routine needs to be written for each pair of resumption protocols for each type. JIT compilers can generate adapters on demand.

Adapter frames help us abstract out the differences between resumption protocols, which are essentially architectural details. Clients only need to reason about return-type consistency instead of return-protocol consistency, as we described in Section 3.2.

A language implementation can designate a particular resumption protocol as the 'default' protocol between high-level language frames, and insert adapter frames when the resumption protocols do not match. The default protocol should usually be the protocol for call sites, because the vast majority of stack frames are created by function calls, not OSR. The top stack frame, if stopped, usually stops on a Swapstack site, although OSR can push ROP frames at the top of a stack in rare cases. Therefore, it is advisable to let the top stack frame have the expected resumption protocol of Swapstack, while all other frames have the expected resumption protocol for normal return.

When a frame cursor iterates through a stack, it can identify the presence of adapter frames by their code addresses, and skip those frames so that they remain invisible to the client.

Summarizing, the correct implementation of the OSR API relies on maintaining consistent stack frame states. At the machine level, the consistency manifests as the matching of resumption protocols between adjacent frames. Different resumption points give frames different resumption protocols, but the difference can be hidden from the API user by using adapter frames which convert between protocols while preserving the value. This means that the OSR API can be simple for the client to use while still being realistically implementable on concrete machines.

6 Related Work

Self VM The Self VM [11, 12] implemented its own Swapstack mechanism which predates the work of Dolan et al. [5]. A thread switches to a dedicated VM stack for handling de-optimization, and could therefore manipulate the victim stack using high-level C++ code. The Self VM uses return address patching [20]—replacing the return address of a frame so that frame replacement can happen when the patched frame returns. Unlike Mu, the Self VM is not a multi-language

[20]'Return address patching' is unrelated to 'return-oriented programming' despite the similar names. ROP is a paradigm where the return values of a function are passed as parameters to the next frame; while return address patching is a technique that overwrites the return address so that some code can be executed after the victim function returns, but before the next frame is activated.

VM, thus it only uses Swapstack and OSR as an internal mechanism.

JikesRVM JikesRVM [1] is a JIT-compiling JVM with OSR support. JikesRVM recompiles methods and generates the contents of new frames in separate OSR threads. However, JikesRVM uses return address patching to let the victim thread execute a piece of code to replace its own frames upon returning from yieldpoints. The code is generated at run time by stitching together platform-specific assembly snippets,[21] because the victim thread is replacing stack frames on the current stack, and thus must carefully handle the stack layout.

JVM The JVM does not have a public API for its internal OSR facilities. The JVM Tool Interface (JVM TI) provides some stack-related API functions for debugging, including introspecting local variables and popping frames. However, unlike the Mu API, JVM TI does not support constructing new frames on existing stacks, therefore does not have all tools needed by OSR.

Truffle and Graal [23] provide a high-level partial evaluation-based language implementation framework and a compiler framework on the JVM. Truffle also has an abstraction of stack frames to support interpreting and de-optimization. This is a higher-level abstraction than Mu. Mu only provides a *minimal* API for the Mu client to build higher-level abstraction. Truffle can be viewed as a potential client that could be implemented *upon* Mu. If properly designed, Mu should allow the Truffle API to be implemented in terms of Mu's API.

LLVM The LLVM [16] compiler framework provides stack maps[22] for stack introspection, but no high-level OSR API.

D'Elia and Demetrescu [4] developed OSRKit, a library built on LLVM for 'dynamically transferring execution between different versions of a function at run time', which they define as 'OSR'. It is an improvement over its predecessor in McJIT by Lameed and Hendren [15]. In OSRKit, the old version of a function tail-calls a stub which recompiles the function and then tail-calls the new version. Both OSRKit and Mu achieved the goal of transferring execution under the constraint that the high-level optimizer and de-optimizer must not depend on machine-level details. Built on LLVM which does not provide any abstraction over stack manipulation, OSRKit chose to depend only on function calls, and not to 'adjust the stack so that execution can continue in f' (the new version) with the current frame' [4], because that 'requires manipulating the program state at machine-code level and is highly ABI- and compiler-dependent.' [4]. Unlike LLVM frontends, Mu clients can rely on the platform-independent OSR API of Mu to replace stack frames while still retaining platform independence. As a more powerful substrate, the Mu API is more flexible than pure call-based approach. For example, the API can easily replace multiple frames at a time, which is useful for handling inlining [11, 12].

7 Conclusion

On-stack replacement is an important mechanism for dynamic optimization. We presented an API for OSR that operates on a simple abstract view of stacks, uses the Swapstack primitive to isolate the active thread from the victim stack, and provides operations to iterate through, introspect and replace stack frames. We also presented the abstraction of 'resumption protocols' which describe the machine-level rules for resuming stopped frames, and guides the implementation of the OSR API itself on real hardware. We demonstrated the use of this API using an experimental JavaScript engine. We showed that this API is implementable on concrete hardware using example implementations of return-oriented programming on x86 and AArch64.

This API brings OSR within reach for more language implementers. Using a micro VM that supports such an API, language implementers can use OSR as a ready-made tool, and focus on the high-level optimizations that are mostly language-specific but vital to the performance of high-level languages [3]. Our hope is that these improved tools for language development will lead to more high-performance language implementations. Meanwhile, because the Mu micro VM is designed for formal verification, this abstraction over OSR can potentially lead to a formally verified VM with built-in OSR support.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1408896. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

This research was supported (partially or fully) by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project DP140-103878). The views expressed herein are those of the authors and are not necessarily those of the Australian Government or Australian Research Council.

Kunshan Wang was supported by the China Scholarship Council.

[21]See the `org.jikesrvm.osr.ia32.CodeInstaller.install` method: https://github.com/JikesRVM/JikesRVM/blob/master/rvm/src/org/jikesrvm/osr/ia32/CodeInstaller.java#L50
[22]See http://llvm.org/docs/StackMaps.html

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. Hummel Flynn, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 314–324, Denver, Colorado, Nov. 1999. doi: 10.1145/320384.320418.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, Minnesota, Oct. 2000. doi: 10.1145/353171.353175.

[3] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. 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, Tucson, Arizona, Oct. 2012. doi: 10.1145/2384616.2384631.

[4] D. C. D'Elia and C. Demetrescu. Flexible on-stack replacement in LLVM. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 250–260, Barcelona, Spain, Mar. 2016. doi: 10.1145/2854038.2854061.

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

[6] DWARF Standards Committee. Dwarf debugging information format, version 4, June 2010. URL http://www.dwarfstd.org/.

[7] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 241–252, San Francisco, California, Mar. 2003. doi: 10.1109/CGO.2003.1191549.

[8] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: A substrate for managed runtime environments. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 51–62, Pittsburgh, Pennsylvania, Mar. 2010. doi: 10.1145/1735997.1736006.

[9] Google. V8 JavaScript Engine, 2017. URL https://developers.google.com/v8/.

[10] D. Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, Department of Computer Science, The University of Arizona, Tucson, Arizona, Oct. 1993.

[11] U. Hölzle and D. Ungar. A third-generation self implementation: Reconciling responsiveness with performance. In *ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 229–243, Portland, Oregon, Oct. 1994. doi: 10.1145/191080.191116.

[12] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 32–43, San Francisco, California, June 1992. doi: 10.1145/143095.143114.

[13] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *ACM SIGOPS Symposium on Operating Systems Principles*, pages 207–220, Big Sky, Montana, Oct. 2009. doi: 10.1145/1629575.1629596.

[14] O. Kowalke. Boost.Context, 2017. URL http://www.boost.org/doc/libs/1_63_0/libs/context/doc/html/index.html.

[15] N. A. Lameed and L. J. Hendren. A modular approach to on-stack replacement in LLVM. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 143–154, Houston, Texas, Mar. 2013. doi: 10.1145/2451512.2451541.

[16] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 75–88, San Jose, California, Mar. 2004. doi: 10.1109/CGO.2004.1281665.

[17] Y. Lin. Zebu: the high-performance implementation of Mu, 2018. URL https://gitlab.anu.edu.au/mu/mu-impl-fast.

[18] Mozilla. SpiderMonkey, 2017. URL https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[19] M. Prandini and M. Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, Nov. 2012. doi: 10.1109/MSP.2012.152.

[20] K. Wang. Holstein: the reference implementation of Mu, 2018. URL https://gitlab.anu.edu.au/mu/mu-impl-ref2.

[21] K. Wang. The specification of the Mu micro virtual machine, 2018. URL https://gitlab.anu.edu.au/mu/mu-spec.

[22] K. Wang, Y. Lin, S. M. Blackburn, M. Norrish, and A. L. Hosking. Draining the swamp: Micro virtual machines as solid foundation for language development. In *Inaugural Summit on Advances in Programming Languages*, pages 321–336, Asilomar, California, May 2015. doi: 10.4230/LIPIcs.SNAPL.2015.321.

[23] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 187–204, Indianapolis, Indiana, Oct. 2013. doi: 10.1145/2509578.2509581.

A Appendix

A.1 Deoptimization in V8

Figure 7 shows how V8 performs OSR for de-optimization using hand-written platform-specific assembly code.

A.2 JS-Mu Compilation Result

Figure 8 shows the result of optimizing a simple JavaScript program.

A.3 SWAPSTACK implementation in `libyugong`

Figure 9 shows the implementation of the SWAPSTACK operation in `libyugong` on x64 on GNU/Linux.

```
1  #define __ masm()->
2
3  void Deoptimizer::TableEntryGenerator::Generate() {
4    GeneratePrologue();
5    const int kNumberOfRegisters = Register::kNumRegisters;
6    // MORE CODE HERE ...
7    ...
8    // Replace the current frame with the output frames.
9    Label outer_push_loop, inner_push_loop,
10       outer_loop_header, inner_loop_header;
11   // Outer loop state: rax = current FrameDescription**, rdx = one past the
12   // last FrameDescription**.
13   __ movl(rdx, Operand(rax, Deoptimizer::output_count_offset()));
14   __ movp(rax, Operand(rax, Deoptimizer::output_offset()));
15   __ leap(rdx, Operand(rax, rdx, times_pointer_size, 0));
16   __ jmp(&outer_loop_header);
17   __ bind(&outer_push_loop);
18   // Inner loop state: rbx = current FrameDescription*, rcx = loop index.
19   __ movp(rbx, Operand(rax, 0));
20   __ movp(rcx, Operand(rbx, FrameDescription::frame_size_offset()));
21   __ jmp(&inner_loop_header);
22   __ bind(&inner_push_loop);
23   __ subp(rcx, Immediate(sizeof(intptr_t)));
24   __ Push(Operand(rbx, rcx, times_1, FrameDescription::frame_content_offset()));
25   __ bind(&inner_loop_header);
26   __ testp(rcx, rcx);
27   __ j(not_zero, &inner_push_loop);
28   __ addp(rax, Immediate(kPointerSize));
29   __ bind(&outer_loop_header);
30   __ cmpp(rax, rdx);
31   __ j(below, &outer_push_loop);
32   // MORE CODE HERE ...
33   ...
34   __ InitializeRootRegister();
35   __ ret(0);
36 }
```

Figure 7. Contemporary VMs rely heavily on assembly code for implementing OSR. This excerpt is from the `Deoptimizer::TableEntryGenerator::Generate` function for x64 in the V8 head revision at the time of writing (https://github.com/v8/v8/blob/01590d660d6c8602b616a82816c4aea2a251be63/src/x64/deoptimizer-x64.cc#L136). V8 invokes this routine when it has already generated the baseline frames in temporary buffers, and each `FrameDescription` object contains the size and the content of a frame. This snippet is a two-level loop. The outer loop iterates through each `FrameDescription`, and the inner loop reads the content of the frame word by word and pushes it onto the current stack. Line 24 uses the `PUSH` instruction which modifies the stack pointer. Since the stack pointer is constantly changing, all other operands, including the loop counters and the pointer to `FrameDescription`, must be held in registers and cannot be spilled. If V8 had had access to a SWAPSTACK primitive, this procedure could have avoided assembly entirely.

```
function sum(f, t) {
    var s = 0;
    for (var i = f; i <= t; i++)
        s = s + i;
    return s;
}
var result = sum(1, 10);
print(result);
```

(a) JavaScript

```
.funcdef @optfunc1_sum VERSION @optfunc1_sum.v1 <@optfunc1_sum.sig> {
  %entry(<@tagref64> %osrParam_f <@tagref64> %osrParam_t
         <@tagref64> %osrParam_s <@tagref64> %osrParam_i):
    // Compensation code
    %raw_f          = COMMINST @uvm.tr64.to_fp (%osrParam_f)
    %raw_t          = COMMINST @uvm.tr64.to_fp (%osrParam_t)
    %raw_s          = COMMINST @uvm.tr64.to_fp (%osrParam_s)
    %raw_i          = COMMINST @uvm.tr64.to_fp (%osrParam_i)
    BRANCH %forHead(%raw_f %raw_t %raw_s %raw_i)

  %forHead(<@double> %f <@double> %t <@double> %s <@double> %i):
    %i_le_t         = FOGE  <@double> %t %i
    BRANCH2 %i_le_t %forBody(%f %t %s %i) %forEnd(%f %t %s %i)

  %forBody(<@double> %f <@double> %t <@double> %s <@double> %i):
    %s2             = FADD <@double> %s %i
    %i2             = FADD <@double> %i @CONST_1
    BRANCH %forHead(%f %t %s2 %i2)

  %forEnd(<@double> %s)
    %tagged_s       = COMMINST @uvm.tr64.from_fp (%s)
    RET <@tagref64> %tagged_s
}
```

(b) Mu IR

Figure 8. Result of JS-Mu compiling a JS program. Sub-figure (a) is a simple JS program that sums over a range. Sub-figure (b) is the optimized Mu IR code generated when optimization is triggered at the loop header. The IR code is adjusted to match the latest Mu IR specification [21]. Auto-generated variable names are simplified for readability. The `%entry` block contains compensation code, which initializes the values of local variables from parameters, and jumps to the loop header that triggered optimization. The compensation code also removes the tags of the values to make them plain `double` type. Therefore, within the loop of `%forHead` and `%forBody`, all variables have been specialized to the `double` type, and no conversion to or from the tagged reference type (`@tagref64`) is present. This is equivalent to the LLVM IR code which Clang could have generated from an equivalent C program with static types.

```
 1 yg_stack_swap:
 2   push rbp
 3   push rbx
 4   push r12
 5   push r13
 6   push r14
 7   push r15
 8
 9   mov  rax, rdx
10
11   lea  rcx, [_yg_ss_cont+rip]
12   push rcx
13
14   mov  [rdi], rsp
15   mov  rsp, [rsi]
16   ret
17
18 _yg_ss_cont:
19   pop  r15
20   pop  r14
21   pop  r13
22   pop  r12
23   pop  rbx
24   pop  rbp
25   ret
```

Figure 9. Swapstack implementation in `libyugong`. This figure shows the assembly code for x64 on GNU/Linux. The `yg_stack_swap` function is called by the swapper, and must be called using the C calling convention with the signature "`uintptr_t yg_stack_swap(void** swapper, void** swappee, uintptr_t value)`". This function passes and receives value of the `uintptr_t` type. The first two parameters (`RDI` and `RSI`) are the locations where the stack pointer of the current stack is saved and where the stack pointer of the swappee is loaded from. The third parameter is the `uintptr_t` value to be passed to the other stack. When called using the C calling convention, the caller-saved registers must have already been saved by the caller compiled by a compliant C compiler. Lines 2–7 then save all callee-saved registers; line 9 moves the third argument (in `RDX`) to the return value register (`RAX`); lines 11–12 push the address of `_yg_ss_cont`; lines 14–15 save the current stack pointer and load the stack pointer of the swappee stack; and line 16 finally returns to the swappee. In `libyugong`, the top of all suspended stacks is laid out as in Figure 4a, and the stack pointer points to the address of the assembly snippet `_yg_ss_cont` shown here if it receives a `uintptr_t` value. When the swapper executes the `RET` instruction in line 16, the thread continues at line 19, restoring the callee-saved registers saved by the swappee stack and returns to the resumption point of the top frame of the swappee. The resumption point is usually the next instruction after the call to `yg_stack_swap`. But when OSR happens and stack frames are popped and pushed, `libyugong` will fix the stack top layout to match Figure 4a so that any thread can swap to any unbound stack the same way.