Flexible On-Stack Replacement in LLVM

Daniele Cono D'Elia Camil Demetrescu
Dept. of Computer, Control, and Management Engineering
Sapienza University of Rome, Italy
{delia,demetres}@dis.uniroma1.it

Abstract

On-Stack Replacement (OSR) is a technique for dynamically transferring execution between different versions of a function at run time. OSR is typically used in virtual machines to interrupt a long-running function and recompile it at a higher optimization level, or to replace it with a different one when a speculative assumption made during its compilation no longer holds.

In this paper we present a framework for OSR that introduces novel ideas and combines features of existing techniques that no previous solution provided simultaneously. New features include OSR with compensation code to adjust the program state during a transition and the ability to fire an OSR from arbitrary locations in the code. Our approach is platform-independent as the OSR machinery is entirely encoded at a compiler's intermediate representation level.

We implement and evaluate our technique in the LLVM compiler infrastructure, which is gaining popularity as Just-In-Time (JIT) compiler in virtual machines for dynamic languages such as Javascript, MATLAB, Python, and Ruby. As a case study of our approach, we show how to improve the state of the art in the optimization of the `feval` instruction, a performance-critical construct of the MATLAB language.

Categories and Subject Descriptors D.3 [*Processors*]: Compilers

Keywords On-stack replacement, just-in-time compilation, code optimization, deoptimization, LLVM.

1. Introduction

The LLVM compiler infrastructure [11] provides a Just-In-Time compiler called MCJIT that is currently being used for generating optimized code at run-time in virtual machines for dynamic languages. MCJIT is employed in both industrial and research projects, including Webkit's Javascript engine, the open-source Python implementation Pyston, the Rubinius project for Ruby, Julia for high-performance technical computing, McVM for MATLAB, CXXR for the R language, Terra for Lua, and the Pure functional programming language. The MCJIT compiler shares the same optimization pipeline with static compilers such as `clang`, and it provides dynamic features such as native code loading and linking, as well as a customizable memory manager.

A piece that is currently missing in MCJIT is a feature to enable on-the-fly transitions between different versions of a running program's function. This feature is commonly known as *On-Stack-Replacement* (OSR) and is typically used in high-performance virtual machines, such as HotSpot and the Jikes RVM for Java, to interrupt a long-running function and recompile it at a higher optimization level. OSR can be a powerful tool for dynamic languages, for which most effective optimization decisions can typically be made only at run-time, when critical information such as type and shape of objects becomes available. In this scenario, OSR becomes useful also to perform deoptimization, i.e., when the running code has been speculatively optimized and the assumption used for the optimization does not hold anymore, the optimized function is interrupted and the execution continues in a safe version of the code.

Currently VM builders using MCJIT are required to have a deep knowledge of the internals of LLVM in order to mimic a transitioning mechanism. In particular, they can rely on two experimental intrinsics, *Stackmap* and *Patchpoint*, to inspect the details of the compiled code generated by the back-end and to patch it manually with a sequence of assembly instructions. In particular, a Stackmap records the location of live values at a particular instruction address and during the compilation it is emitted into the object code within a designated section; a Patchpoint instead allows to reserve space at an instruction address for run-time patching and can be used to implement an inline caching mechanism [14].

Lameed and Hendren propose McOSR [10], a technique for OSR that stores the live values in a global buffer, recompiles the current function, and then loads in it the saved state when the execution is resumed. McOSR was designed for

CGO'16, March 12–18, 2016, Barcelona, Spain

http://dx.doi.org/10.1145/2854038.2854061

the legacy JIT – no longer included in LLVM since release 3.6 – and has several limitations that we discuss in Section 3.

Contributions. In this paper we propose a general-purpose, target-independent implementation of on-stack replacement. Specific goals of our approach include:

- The ability for a function reached via OSR to fire an OSR itself: this would allow switching from a base function f to an optimized function f', and later on to a further optimized version f'', and so on.
- Supporting deoptimization, i.e., transitions from an optimized function to a less optimized function from which it was derived.
- Supporting transitions at arbitrary points, including those that would require adjusting the transferred program state to resume the execution in the OSR target function.
- Supporting OSR targets either generated at run-time (e.g., using profiling information) or already known at compilation time.
- Hiding from the front-end that generates the different function versions all the implementation details for handling OSR transitions between them at specific points.

We implemented the proposed approach in OSRKit, a prototype library for LLVM IR manipulation based on MCJIT with the following design goals:

- Encoding OSR transitions in terms of pure IR code only, avoiding manipulations at machine-code level.
- Providing a front-end with a clean interface to specify glue code for OSR transitions requiring adjustments to the program state.
- Incurring a minimal level of intrusiveness in terms of both the instrumentation of the code generated by the front-end and the impact of OSR points on native code quality.
- Relying on LLVM's compilation pipeline to generate the most efficient native code for an instrumented function.

While the general ideas we propose have been prototyped in LLVM, we believe that they could be applied to other toolchains as well. To investigate the potential of our approach, we show how to optimize the `feval` construct – a major source of inefficiency in MATLAB execution engines [9, 15]. We present an extension of the MATLAB McVM runtime [3] based on OSRKit to enable aggressive specialization mechanisms for `feval` that were not supported by extant techniques [9]. An experimental evaluation of our technique reveals that the OSR machinery injected by OSRKit has a small level of intrusiveness and the optimizations enabled by our approach can yield significant speedups in practical scenarios. We devise an artifact[1], endorsed by the joint PPoPP-CGO 2016 Artifact Evaluation committee, that will allow the interested reader to repeat the experiments presented in this work and to get acquainted with OSRKit[2].

[1] Our artifact is available in the ACM Digital Library and also at `http://www.dis.uniroma1.it/~demetres/artifacts/cgo2016/`.

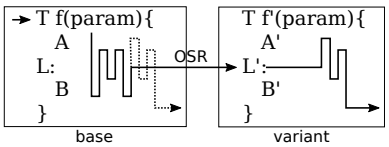

Figure 1. On-stack replacement dynamics: control is transferred via OSR from a point L of a base function f to a point L' in a variant f' of f.

Structure of the paper. The remainder of this paper is organized as follows. In Section 2 we present our OSR technique and in Section 3 we outline its implementation in LLVM. Section 4 illustrates our `feval` case study in McVM. In Section 5, we present our experimental study and discuss implications of injecting OSR points in LLVM IR programs. Related work is discussed in Section 6 and concluding remarks are given in Section 7.

2. Overview

In this section we provide an overview of our ideas. The key to platform independence in our work is to express the entire OSR machinery at intermediate code representation level, without resorting to machine-level code manipulation or special intrinsics of the intermediate language such as Stackmap and Patchpoint in LLVM IR.

Consider the generic OSR scenario shown in Figure 1. A base function f is executed and it can either terminate normally (dashed lines), or an OSR event may transfer control to a variant f', which resumes the execution. The decision of whether an OSR should be fired at a given point L of f is based on an *OSR condition*. A typical example in JIT-based virtual machines is a profile counter reaching a certain hotness threshold, which indicates that f is taking longer than expected and is worth optimizing. Another example is a guard testing whether f has become unsafe and execution needs to fall back to a safe version f'. This scenario includes deoptimization of functions generated with aggressive speculative optimizations.

Several OSR implementations adjust the stack so that execution can continue in f' with the current frame [2, 8, 19]. This requires manipulating the program state at machine-code level and is highly ABI- and compiler-dependent. A simpler approach, which we follow in this article, consists in creating a new frame every time an OSR is fired, essentially regarding an OSR transition as a function call [10, 14].

Our implementation targets two general scenarios: 1) *resolved OSR*: f' is known before executing f as in the deoptimization example discussed above; 2) *open OSR*: f' is generated when the OSR is fired, supporting deferred and

[2] OSRKit is available at `https://github.com/dcdelia/tinyvm`.

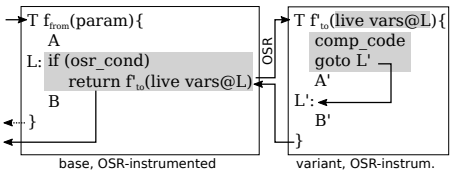

Figure 2. Resolved OSR scenario.

profile-guided compilation strategies. In both cases, f is instrumented before its execution to incorporate the OSR machinery. We call such OSR-instrumented version f_{from}.

In the resolved OSR scenario (see Figure 2), instrumentation consists of adding a check of the OSR condition and, if it is satisfied, a tail call that fires the OSR. The called function is an instrumented version of f', which we call f'_{to}. We refer to f'_{to} as the *continuation function* for an OSR transition. The assumption is that f'_{to} produces the same side-effects and return value that one would obtain by f if no OSR was performed. Differently from f', f'_{to} takes as input all live variables of f at L, executes an optional *compensation code* to fix the computation state (`comp_code`), and then jumps to a point L' from which execution can continue.

Compensation code adds flexibility to our framework, as it extends the range of points where OSR transitions can be fired. In fact, the OSR practice often makes the conservative assumption that execution can always continue with the very same program state as the base function. This assumption can however be restrictive, as it may reduce the number of program locations eligible for OSR (i.e., one has to wait to a point where the states would realign). Our solution provides a front-end with means to encode a glue code, tailored to the specific optimizations involved between two function versions, to adjust the program state and perform an OSR transition. This code can be used, for instance, to modify the heap, or to compute variables that are live at L' but not at L.

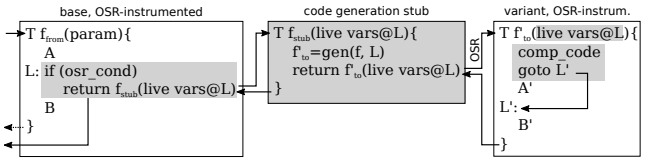

Figure 3. Open OSR scenario.

The open OSR scenario is similar, with one main difference (see Figure 3): instead of calling f'_{to} directly, f_{from} calls a stub function f_{stub}, which first creates f'_{to} and then calls it. Function f'_{to} is generated by a function `gen` that takes the base function f and the OSR point L as input. The reason for having a stub in the open OSR scenario, rather than directly instrumenting f with the code generation machinery, is to minimize the extra code injected into f. Indeed, instrumentation may interfere with optimizations, e.g., by increasing register pressure and altering code layout and instruction cache behavior.

```
int isord(long* v, long n, int (*c)(void*,void*)) {
    for (long i=1; i<n; i++)
        if (c(v+i-1,v+i)>0) return 0;
    return 1;
}
```

Figure 4. Example for OSR instrumentation in LLVM.

Discussion. Instrumenting functions for OSR at a higher level than machine code yields several benefits:

1. *Platform independence*: the OSR instrumentation code is lowered to native code by the compiler back-end, which handles the details of the target ABI.
2. *Global optimizations*: lowering OSR instrumentation code along with application code can generate faster code than local binary instrumentation. For instance, dead code elimination can suppress from f'_{to} portions of code that would no longer be needed when jumping to the landing pad L', producing smaller code and enabling better register allocation and instruction scheduling.
3. *Debugging and Profiling*: preserving ABI conventions in the native code versions of f_{from}, f_{stub}, and f'_{to} helps debuggers and profilers to more precisely locate the current execution context and collect more informative data.
4. *Abstraction*: being entirely encoded using high-level language constructs (assignments, conditionals, function calls), the approach is amenable to a clean instrumentation API that abstracts the OSR implementation details, allowing a front-end to focus on where to insert OSR points independently of the final target architecture.

A natural question is whether encoding OSR at a higher level of abstraction can result in poorer performance than binary code approaches. We address this issue in Section 3, where we analyze the OSR machine code generated for an x86-64 target, and in Section 5, where OSR performance is measured on classic benchmarks.

3. OSR in LLVM

In this section we discuss one possible embodiment of the OSR approach of Section 2 in LLVM. Our discussion is based on a simple running example that illustrates a profile-driven optimization scenario. We start from a simple base function (`isord`) that checks whether an array of numbers is ordered according to some criterion specified by a comparator (see Figure 4). Our goal is to instrument `isord` so that, whenever the number of loop iterations exceeds a certain threshold, control is dynamically diverted to a faster version generated on the fly by inlining the comparator.

The IR code shown in this section[3] has been generated with `clang` and instrumented with OSRKit, a library we prototyped to help VM builders deploy OSR in LLVM. OS-

[3] Virtual register names and labels in the LLVM-produced IR code shown in this paper have been refactored to make the code more readable.

```
define i32 @isordfrom(
  i64* %v, i64 %n, i32 (i8*, i8*)* nocapture %c) {
entry:
  %t0 = icmp sgt i64 %n, 1
  br i1 %t0, label %loop.body, label %exit

loop.header:
  %t1 = icmp slt i64 %i1, %n
  br i1 %t1, label %loop.body, label %exit

loop.body:
  %i = phi i64 [%i1, %loop.header], [1,%entry]
  %p.osr = phi i64 [%p.osr1, %loop.header],
                   [1000, %entry]
  %p.osr1 = add nsw i64 %p.osr, -1
  %osr.cond = icmp eq i64 %p.osr, 0
  br i1 %osr.cond, label %osr,
                   label %loop.body.cont
loop.body.cont:
  %t2 = getelementptr inbounds i64* %v, i64 %i
  %t3 = add nsw i64 %i, -1
  %t4 = getelementptr inbounds i64* %v, i64 %t3
  %t5 = bitcast i64* %t4 to i8*
  %t6 = bitcast i64* %t2 to i8*
  %t7 = tail call i32 %c(i8* %t5, i8* %t6)
  %t8 = icmp sgt i32 %t7, 0
  %i1 = add nuw nsw i64 %i, 1
  br i1 %t8, label %exit, label %loop.header

exit:
  %res = phi i32 [1, %entry], [1, %loop.header]
                 [0, %loop.body.cont],
  ret i32 %res

osr:
  %val = bitcast i32 (i8*, i8*)* %c to i8*
  %osr.res = call i32 @isordstub(i8* %val,
      i64* %v, i64 %n, i32 (i8*, i8*)* %c, i64 %i)
  ret i32 %osr.res
}
```

Figure 5. LLVM IR version of base function `isord` (Figure 4) instrumented for open OSR. The OSR is fired at the beginning of the loop body after 1000 iterations. Additions resulting from the instrumentation are in grey.

RKit provides a number of useful abstractions that include open and resolved OSR instrumentation of IR base functions without breaking the SSA (Static Single Assignment) form, liveness analysis, generation of OSR continuation functions, and mapping of LLVM values between different versions of a program along with compensation code generation.

OSR Instrumentation in IR. To defer the compilation of the continuation function until the comparator is known at run time, we used OSRKit to instrument `isord` with an open OSR point at the beginning of the loop body, as shown in Figure 5. Portions added to the original code by OSR instrumentation are highlighted in grey. New instructions are placed at the beginning of the loop body to increment a hotness counter `p.osr` and jump to an OSR-firing block if the counter reaches the threshold (1000 iterations in this example). The OSR block contains a tail call to the target generation stub, which receives as parameters the four live variables at the OSR point (`v`, `n`, `c`, `i`). OSRKit allows the stub

```
define i32 @isordstub(
   i8* %val, i64* %v_osr, i64 %n_osr,
   i32 (i8*, i8*)* nocapture %c_osr, i64 %i_osr) {
entry:
  %cont.func = call
     ; generator returns ptr to isordto
     i32 (i64*, i64, i32 (i8*, i8*)*, i64)*
        (i8*, i8*, i8*, i8*)* inttoptr

     ; generator function address is 4357824
     (i64 4357824 to
           i32 (i64*, i64, i32 (i8*, i8*)*, i64)*
                 (i8*, i8*, i8*, i8*)*)

     ; hard-coded parameters passed to generator:
     ;  46993664 = addr of isord IR function
     ;  46995056 = addr of basic block at loop.body
     ;  47005408 = addr of code generation env
     (i8* inttoptr (i64 46993664 to i8*),
      i8* inttoptr (i64 46995056 to i8*),
      i8* inttoptr (i64 47005408 to i8*), i8* %val)

  %osr.res = call i32 %cont.func(i64* %v_osr,
    i64 %n_osr, i32 (i8*, i8*)* %c_osr, i64 %i_osr)
  ret i32 %osr.res
}
```

Figure 6. IR stub that generates the continuation function when an open OSR is fired by `isordfrom` (Figure 5).

to receive the run-time value `val` of an IR object that can be used to produce the continuation function – in our example, the pointer to the comparator function to be inlined. The stub (see Figure 6) calls a code generator that: 1) builds an optimized version of `isord` by inlining the comparator, and 2) uses it to create the continuation function `isordto` shown in Figure 7. The stub passes to the code generator four parameters: 1) a pointer to the `isord` IR code, 2) a pointer to the basic block in `isord` from which the OSR is fired, 3) a user-defined object to support code generation in MCJIT, and 4) the stub's `val` parameter (the first three are hard-wired by OSRKit). The stub terminates with a tail call to `isordto`. To generate the continuation function from the optimized version created by the inliner, OSRKit replaces the function entry point, removes dead code, replaces live variables with the function parameters, and fixes ϕ-nodes accordingly. Additions resulting from the IR instrumentation are in grey, while removals are struck-through.

x86-64 Lowering. Figure 8 shows the x86-64 code generated by the LLVM back-end for `isordfrom` and `isordto`. For the sake of comparison with the native code that would be generated for the original non-OSR versions, additions resulting from the IR instrumentation are in grey, while removals are struck-through. Notice that the OSR intrusiveness in `isordfrom` is minimal, consisting of just two assembly instructions with register and immediate operands. As a result of induction variable canonicalization in the LLVM back-end, loop index `i` and hotness counter `p.osr` are fused in register `%r12`. We also note that tail call optimization is applied in the OSR-firing block, resulting in no stack growth during an OSR. The continuation function `isordto` is iden-

```
define i32 @isordto(
  i64* nocapture readonly %v_osr,
  i64 %n_osr, i32 (i8*, i8*)* %c_osr, i64 %i_osr) {

osr.entry: ; no compensation code needed...
  br label %loop.body

entry:
  %t1 = icmp sgt i64 %n_osr, 1
  br i1 %t1, label %loop.body, label %exit

loop.header:
  %t2 = icmp slt i64 %i1, %n_osr
  br i1 %t2, label %loop.body, label %exit

loop.body:
  %i = phi i64 [ %i1, %loop.header ],
               [ 1, %entry ],
               [ %i_osr, %osr.entry ]
  %t3 = add nsw i64 %i, -1
  %t4 = getelementptr inbounds i64* %v_osr, i64 %t3
  %t5 = load i64* %t4, align 8, !tbaa !1
  %t6 = getelementptr inbounds i64* %v_osr, i64 %i
  %t7 = load i64* %t6, align 8, !tbaa !1
  %t8 = icmp sgt i64 %t5, %t7
  %i1 = add nuw nsw i64 %i, 1
  br i1 %t8, label %exit, label %loop.header

exit:
  %res = phi i32 [ 1, %entry ],
                 [ 0, %loop.body ],
                 [ 1, %loop.header ]
  ret i32 %res
}
```

Figure 7. Faster variant of `isord` (Figure 4) in LLVM IR with comparator inlining, instrumented as OSR continuation function. Instrumentation additions are in grey. The original function entry block is unreachable after instrumentation and is eliminated (struck-through code fragments).

```
isordfrom:
   pushq %r15
   pushq %r14
   pushq %r12
   pushq %rbx
   pushq %rax
   movq %rdx, %r14 #c
   movq %rsi, %r15 #n
   movq %rdi, %rbx #v
   movl $1, %r12d  #i
   cmpq $1, %r15
   jle .LBB0_1
.LBB0_4: # %loop.body
   cmpq $1001, %r12
   je .LBB0_7
   movq %rbx, %rdi
   leaq 8(%rbx), %rbx
   movq %rbx, %rsi
   callq *%r14
   movl %eax, %ecx
   xorl %eax, %eax
   testl %ecx, %ecx
   jg .LBB0_6
   incq %r12
   cmpq %r15, %r12
   jl .LBB0_4
   movl $1, %eax
   jmp .LBB0_6
.LBB0_1:
   movl $1, %eax
.LBB0_6: # %exit
   addq $8, %rsp
   popq %rbx
   popq %r12
   popq %r14
   popq %r15
   retq
.LBB0_7: # %osr
   movq %r14, %rdi # c
   movq %rbx, %rsi # v
   movq %r15, %rdx # n
   movq %r14, %rcx # c
   movq %r12, %r8  # i
   addq $8, %rsp
   popq %rbx
   popq %r12
   popq %r14
   popq %r15
   jmp isordstub

isordto:
   movl $1, %edx
   cmpq $1, %rsi
   jle .LBB0_1
.LBB0_4: # %loop.body
   movq -8(%rdi,%rdx,8),%rcx
   xorl %eax, %eax
   cmpq (%rdi,%rdx,8),%rcx
   jg .LBB0_5
   incq %rdx
   cmpq %rsi, %rdx
   jl .LBB0_4
.LBB0_1:
   movl $1, %eax
.LBB0_5: # %exit
   retq
```

Figure 8. OSR-instrumented functions `isordfrom` (base) and `isordto` (faster continuation) after IR-to-x86-64 lowering in LLVM. Additions resulting from the IR instrumentation are in grey, while removals are struck-through.

tical to the specialized version of `isord` with inlined comparator, except that the loop index is passed as a parameter in `%rdx` and no preamble is needed since OSR jumps directly in the loop body.

Comparison with McOSR. McOSR [10] is a library for inserting open OSR points designed specifically for the legacy LLVM JIT, and encodes the OSR machinery entirely in IR as OSRKit does. When an OSR is fired, live variables are stored into a pool of globals allocated by the library. McOSR then invokes a user-defined method to transform f into f' and calls f with empty parameters. The new entrypoint inserted by McOSR in f checks a global flag to discriminate if the function is being invoked in an OSR transition or as a regular call: in the first case, the state is restored from the pool of global variables before jumping to the OSR landing pad.

OSRKit improves upon McOSR in a number of aspects. The presence of a new entrypoint has the potential to disrupt many optimizations: McOSR tries to mitigate this issue by promptly recompiling f again once the execution is resumed and f has returned, but only future invocations of f would benefit from it. In contrast, OSRKit generates an optimized, dedicated OSR continuation function to resume the execution: lessons from the Jikes RVM [6] suggest that our approach is likely to yield better performance. Also, we transfer live variables as arguments to the continuation function, possibly using registers, which is likely to be more efficient than spilling them to a pool of globals. Due to the complexity in preserving the SSA form when updating the IR, McOSR allows the insertion of OSR points only at loop headers (in particular, those with exactly two predecessor blocks), while OSRKit can encode them at arbitrary program locations.

Notice also that OSRKit introduces a number of features that are absent from McOSR, including: support for compensation code and resolved OSR points; compatibility with MCJIT's design; support for maintaining multiple versions of the same function, which can be very useful in the presence of speculative optimizations and deoptimization.

4. Optimizing `feval` in McVM

In this section we show how OSRKit can be used in a production VM to implement aggressive optimizations for dynamic languages. We focus on MATLAB's `feval` construct, a widely used built-in higher-order function that applies the function passed as first parameter to the remaining arguments (e.g., `feval(g,x,y)` computes `g(x,y)`). This fea-

ture is used in many classes of numerical computations that benefit from having functions as parameters.

A previous study by Lameed and Hendren [9] shows that the overhead of an `feval` call is significantly higher than a direct call, especially in JIT-based execution environments such as McVM [3] and the proprietary MATLAB JIT accelerator by Mathworks. In fact, the presence of an `feval` instruction can disrupt the results of intra- and inter-procedural level for type and array shape inference analyses, which are key factors for efficient code generation. Furthermore, since `feval` invocations typically require a fallback to an interpreter, parameters passed to an `feval` are typically boxed to make them more generic.

Our case study presents a novel technique for optimizing `feval` in the McVM virtual machine, a complex research project developed at McGill University. McVM is publicly available [21] and includes: a front-end for lowering MATLAB programs to an intermediate representation called IIR that captures the high-level features of the language; an interpreter for running MATLAB functions and scripts in IIR format; a manager component to perform analyses on IIR; a JIT compiler based on LLVM for generating native code for a function, lowering McVM IIR to LLVM IR; a set of helper components to perform fast vector and matrix operations using optimized libraries such as ATLAS, BLAS and LAPACK. McVM implements a function versioning mechanism based on type specialization, which is the main driver for generating efficient code [3].

4.1 Current Approaches

Lameed and Hendren [9] proposed two dynamic techniques for optimizing `feval` instructions in McVM: *JIT-based* and *OSR-based* specialization. Both attempt to optimize a function f that contains instructions of the form $\texttt{feval}(g, ...)$, leveraging information about g and the type of its arguments observed at run-time. The optimization produces a specialized version f' where $\texttt{feval}(g, x, y, z, ...)$ instructions are replaced with direct calls of the form $g(x, y, z, ...)$.

The two approaches differ in the points where code specialization is performed. In JIT-based specialization, f' is generated when f is called. In contrast, the OSR-based method interrupts f as it executes, generates a specialized version f', and resumes from it.

Another technical difference, which has substantial performance implications, is the representation level at which optimization occurs in the two approaches. When a function f is first compiled from MATLAB to IIR, and then from IIR to IR, the functions it calls via `feval` are unknown and the type inference engine is unable to infer the types of their returned values. Hence, these values must be kept boxed in heap-allocated objects and handled with slow generic instructions in the IR representation of f (suitable for handling different types). The JIT method works on the IIR representation of f and can resort to the full power of type analysis to infer the types of the returned values of g, turning the slow generic instructions of f into fast type-specialized instructions in f'. On the other hand, OSR-based specialization operates on the IR representation of f, which prevents the optimizer from exploiting type inference. As a consequence, for f' to be sound, the direct call to g must be guarded by a condition that checks if the type of its parameters remain the same as observed at the time when f was interrupted. If the guard fails, or the `feval` target g changes, the code falls back to executing the original `feval` instruction.

JIT-based specialization is less general than OSR-based specialization, as it only works if the `feval` argument g is one of the parameters of f, but is substantially faster due to the benefits of type inference.

4.2 A New Approach

In this section, we present a new approach that combines the flexibility of OSR-based specialization with the efficiency of the JIT-based method, answering an open question raised by Lameed and Hendren [9]. The key idea is to lift the f-to-f' optimization performed by the OSR-based specialization from IR to IIR level. This makes it possible to perform type inference in f', generating a much more efficient code. The main technical challenge of this idea is that the program's state in f at the OSR point may be incompatible with the state of f' from which execution continues. Indeed, some variables may be boxed in f and unboxed in f'. Hence, compensation code is needed to adjust the state by performing live variable unboxing during the OSR.

Implementation in McVM. We implemented our approach in McVM[4], extending it with four main components:

1. An analysis pass to identify optimization opportunities for `feval` instructions in the IIR of a function.
2. An extension for the IIR compiler to track the *variable map* between IIR and IR objects at `feval` sites.
3. An OSR inserter based on OSRKit to inject open OSR points in the IR for IIR locations annotated during the analysis pass.
4. An `feval` optimizer triggered at OSR points, which uses:
   (a) a profile-driven IIR generator to replace `feval` calls with direct calls;
   (b) a helper component to lower the optimized IIR function to IR and construct a state mapping;
   (c) a code caching mechanism to handle the compilation of the continuation functions.

We remark that our implementation heavily depends on OSRKit's ability to handle compensation code.

Optimizer. The optimizer is called as `gen` function in the open OSR stub (see Figure 3) created by the OSR inserter. It receives the IR version f^{IR} of function f, the basic block of

[4] As a by-product of our project, we ported the MATLAB McVM virtual machine from the LLVM legacy JIT to the new MCJIT toolkit. Our code is available at `https://github.com/dcdelia/mcvm`.

```
define void @odeEuler_OSR(
  i64 %0, i64 %1, i8* %2, i8* %3, i8* %4,
  i64 %5, i8* %6, double %7,
  { i8*, i8*, i64 }* %8, i8* %9) {

osr.entry:
  %castUNKtoMF64 = call double
      @"MatrixF64Obj::getScalarVal"(i8* %2)
  %castUNKtoMF64_2 = call double
      @"MatrixF64Obj::getScalarVal"(i8* %4)
  %envLookupFory = call i8*
      @"Environment::lookup"(i8* %9, i8* inttoptr
      (i64 32152960 to i8*))
  %10 = alloca [16 x i8]
  %11 = alloca [24 x i8]
  br label %31
```

Figure 9. Compensation code for `odeEuler` benchmark. McVM-specific instructions are highlighted in grey.

f^{IR} where the OSR was fired, and the native code address of the `feval` target function g. As a first step, the optimizer looks up the IR code of g by its address and checks whether a previously compiled version of f specialized with g was previously cached. If not, a new function f_{opt}^{IIR} is generated by cloning the IIR representation f^{IIR} of f and by replacing all `feval` calls to g in f_{opt}^{IIR} with direct calls.

As a next step, the optimizer asks the IIR compiler to lower f_{opt}^{IIR} to f_{opt}^{IR}. During the process, the compiler stores the variable map between IIR and IR objects at the direct call replacing the `feval` instruction that triggered the OSR.

Using this map and the one stored during the lowering of f^{IIR}, the optimizer constructs a state mapping between f^{IR} and f_{opt}^{IR}. In particular, for each value in f_{opt}^{IR} live at the continuation block we determine whether we can assign to it a live value passed at the OSR point, or a compensation code is required to set its value.

Notice that, since the type inference engine yields more accurate results for f_{opt}^{IIR} compared to f^{IIR}, the IIR compiler can in turn generate efficient specialized IR code for representing and manipulating IIR variables, and compensation code is typically required to unbox or downcast some of the live values passed at the OSR point.

Once a state mapping has been constructed, the optimizer asks OSRKit to generate the continuation function for the OSR transition and then executes it.

An example of compensation code is reported in Figure 9. In order to correctly resume the execution at the first instruction in basic block `%31`, the entrypoint of `odeEuler`'s continuation function executes a sequence of instructions that: 1) convert to `double` two live variables – i.e., function arguments `%2` and `%4` – that are represented as boxed values in the unoptimized function, 2) look up in McVM's environment at `%9` the pointer to the object instantiated for the symbol description stored at address `0x32152960`, and 3) allocate on the stack two buffers of 16 and 24 bytes, respectively.

Discussion. The ideas presented in this section advance the state of the art of `feval` optimization in MATLAB runtimes. Similarly to OSR-based specialization, we do not place restrictions on the functions that can be optimized. On the other hand, we work at IIR (rather than IR) level as in JIT-based specialization, which allows us to perform type inference on the code with direct calls. Working at IIR level eliminates the two main sources of inefficiency of OSR-based specialization: 1) we can replace generic instructions with specialized instructions, and 2) the types of g's arguments do not need to be cached or guarded as they are statically inferred. These observations are confirmed in practice by experiments on benchmarks from the MATLAB community, as we will show in Section 5.2.

Benchmark	Description
b-trees	Adaptation of a GC bench for binary trees
fannkuch	Fannkuch benchmark on permutations
fasta	Generation of DNA sequences
fasta-redux	Generation of DNA sequences (with lookup table)
mbrot	Mandelbrot set generation
n-body	N-body simulation of Jovian planets
rev-comp	Reverse-complement of DNA sequences
sp-norm	Eigenvalue calculation with power method

Table 1. Description of the `shootout` benchmarks.

5. Experimental Evaluation

In this section we present a preliminar experimental study of OSRKit aimed at addressing the following questions:

Q1 How much does a never-firing OSR point impact code quality? What kind of slowdown should we expect?

Q2 What is the run-time overhead of an OSR transition, for instance to a clone of the running function?

Q3 What is the overhead of OSRKit for inserting OSR points and creating a stub or a continuation function?

Q4 What kind of benefits can we expect by using OSR in a production environment based on LLVM?

5.1 Benchmarks and Setup

We address questions Q1-Q3 by analyzing the performance of OSRKit on a selection of the `shootout` benchmarks [7] running in a proof-of-concept virtual machine we developed in LLVM. In particular, we focus on single-threaded benchmarks that do not rely on external libraries to perform their core computations. Benchmarks and their description are reported in Table 1; four of them (`b-trees`, `mbrot`, `n-body` and `sp-norm`) are evaluated against two workloads of different size.

We generate the IR modules for our experiments with `clang` starting from the C version of the `shootout` suite. To cover scenarios where OSR machinery is inserted in programs with different optimization levels, we consider two versions: 1) *unoptimized*, where the only LLVM optimization we perform is `mem2reg` to promote stack references to

registers and construct the SSA form; 2) *optimized*, where we apply `opt -O1` to the unoptimized version.

For question Q4, we analyze the impact of the optimization technique presented in Section 4.2 on the running time of a few numeric benchmarks, namely `odeEuler`, `odeMidpt`, `odeRK4`, and `sim_anl`. The first three benchmarks [16] solve an ordinary differential equation for heat treating simulation using the Euler, midpoint, and Range-Kutta method, respectively; the last benchmark minimizes the six-hump camelback function with the method of simulated annealing[5].

All the experiments were performed on an octa-core 2.3 Ghz Intel Xeon E5-4610 v2 with 256+256KB of L1 cache, 2MB of L2 cache, 16MB of shared L3 cache and 128 GB of DDR3 main memory, running Debian Wheezy 7, Linux kernel 3.2.0, LLVM 3.6.2 (Release build, compiled using gcc 4.7.2), 64 bit.

For each benchmark we analyze CPU time performing 10 trials preceded by an initial warm-up iteration; reported confidence intervals are stated at 95% confidence level.

5.2 Results

Q1: Impact on Code Quality. In order to measure how much a never-firing OSR point might impact code quality, we analyzed the source-code structure of each benchmark and profiled its run-time behavior to identify performance-critical sections for OSR point insertion. The distinction between open and resolved OSR points is nearly irrelevant in this context: we choose to focus on open OSR points, passing `null` as the `val` argument for the stub (see Section 3).

For iterative benchmarks, we insert an OSR point in the body of their hottest loops. We classify a loop as hottest when its body is executed for a very high cumulative number of iterations (e.g., from millions up to billions) and it either calls the method with the highest *self* time in the program, or it performs the most computational-intensive operations for the program in its own body. These loops are natural candidates for OSR point insertion: for instance, the Jikes RVM inserts yield points on backward branches to trigger method recompilation through OSR or thread preemption for garbage collection. In the `shootout` benchmarks, the number of such loops is typically 1 (2 for `spectral-norm`).

For recursive benchmarks, we insert an OSR point in the body of the method that accounts for the largest *self* execution time in the program. Such an OSR point might be useful to trigger recompilation of the code at a higher degree of optimization, enabling for instance multiple levels of inlining for non-tail-recursive functions. The only analyzed benchmark showing a recursive pattern is `b-trees`.

Results for the unoptimized and optimized versions of the benchmarks are reported in Figure 10 and Figure 11, respectively. For both scenarios we observe that the overhead is very small, i.e., less than 1% for most benchmarks and less than 2% in the worst case. For some benchmarks, code might run slightly faster after OSR point insertion due to instruction cache effects. The number of times the OSR condition is checked for each benchmark is reported in Table 2.

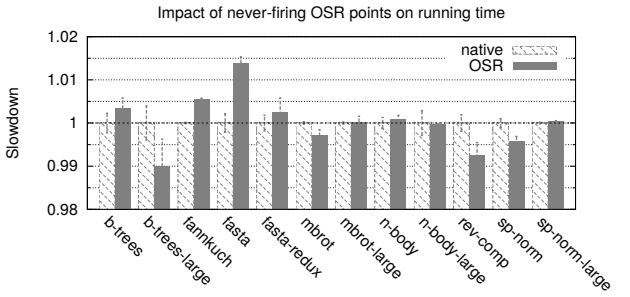

Figure 10. Q1: Impact on running time of never-firing OSR points inserted inside hot code portions (unoptimized code).

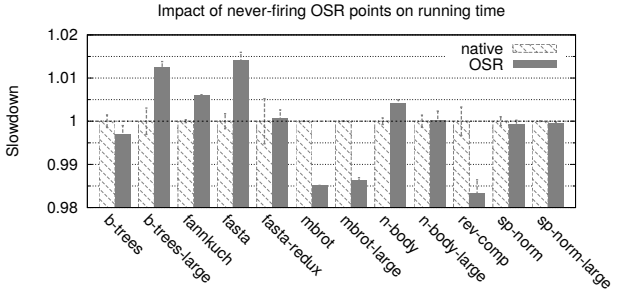

Figure 11. Q1: Impact on running time of never-firing OSR points inserted inside hot code portions (optimized code).

Q2: Overhead of OSR Transitions. Table 2 reports estimates of the average cost of performing an OSR transition to a clone of the running function. For each benchmark we compute the time difference between the scenarios in which an always-firing and a never-firing resolved OSR point is inserted in the code, respectively; we then normalize this difference against the number of fired OSR transitions.

Hot code portions for OSR point insertion have been identified as in Q1. Depending on the characteristics of the hot loop, we either transform its body into a separate function and instrument its entrypoint, or, when the loop calls a method with a high self time, we insert an OSR point at the beginning of that method.

Normalized differences reported in the table represent a reasonable estimate of the average cost of firing an OSR transition. Reported numbers are in the order of nanoseconds, and might be negative due to instruction cache effects.

Q3: OSR Machinery Generation. We now discuss the overhead of the OSRKit library for inserting OSR machinery in the IR of a function. Table 3 reports for each benchmark the number of IR instructions in the instrumented function

[5] `http://www.mathworks.com/matlabcentral/fileexchange/33109-simulated-annealing-optimization`

Benchmark	Fired OSRs (M)	*Unoptimized code* Live values	*Unoptimized code* Avg time (ns)	*Optimized code* Live values	*Optimized code* Avg time (ns)
b-trees	605	2	1.731	3	0.974
b-trees-large	2 690	2	1.749	3	1.423
fannkuch	399	0	1.793	0	0.621
fasta	400	2	2.335	2	2.699
fasta-redux	400	4	2.306	4	2.269
mbrot	256	15	5.016	15	3.628
mbrot-large	1 024	15	5.268	15	4.637
n-body	50	3	2.952	3	6.929
n-body-large	500	3	2.953	3	6.953
rev-comp	6	8	-10.158	8	8.267
sp-norm	1 210	2	0.772	2	-0.030
sp-norm-large	19 360	2	0.778	2	-0.003

Table 2. Cost of OSR transitions to the same function. For each benchmark we report the number of fired OSR transitions (rounded to millions), the number of live values passed at the OSR point, and the average time for a transition.

Benchmark	|IR|	*Open OSR* (μs) Insert point	*Open OSR* (μs) Gen. stub	*Resolved OSR* (μs) Insert point	Generate f'_{to} Total	Generate f'_{to} Avg/inst
b-trees	13	15.40	28.32	14.31	76.13	5.86
fannkuch	50	14.16	18.66	12.84	208.03	4.16
fasta	38	12.93	27.07	13.01	250.39	6.59
fasta-redux	55	13.79	23.44	9.32	258.36	4.70
mbrot	77	15.96	27.39	15.30	384.61	4.99
n-body	19	14.31	19.73	11.58	88.73	4.67
rev-comp	145	16.31	39.99	13.90	810.84	5.59
sp-norm	28	15.31	27.50	12.41	154.54	5.52

Table 3. Q3: OSR machinery insertion in optimized code. Time measurements are expressed in microseconds. Results for unoptimized code are very similar and thus not reported.

and the time spent in the IR manipulation. Locations for OSR points are chosen as in Q1, and the target function is a clone of the source function.

For open OSR points, we report the time spent in inserting the OSR point in the function and in generating the stub; both operations do not depend on the size of the function. For resolved OSR points, we report the time spent in inserting the OSR point and in generating the f'_{to} function.

Not surprisingly, constructing a continuation function takes longer than the other operations (i.e., up to 1 ms vs. 20-40 us), as it involves cloning and manipulating the body of the target function and thus depends on its size: Table 3 hence comes with an additional column in which time is normalized against the number of IR instructions in the target function.

Discussion. Experimental results presented in this section suggest that inserting an OSR point is unlikely to degrade the quality of generated code (Q1). The time required to fire an OSR transition is negligible (i.e., order of nanoseconds, Q2), while the cost of OSR-point insertion and of generating a continuation function is likely to be dominated by the cost of its compilation (Q3). For a front-end, the choice whether to insert an OSR point into a function for dynamic optimization merely depends on the trade-off between the expected benefits in terms of execution time and the overheads from generating and JIT-compiling an optimized version of the function; compared to these two operations, the cost of OSR-related operations is negligible.

Benchmark	Base (cached)	Optimized (JIT)	Optimized (cached)	Direct (by hand)
odeEuler	1.046	2.796	2.800	2.828
odeMidpt	1.014	2.645	2.660	2.685
odeRK4	1.005	2.490	2.582	2.647
sim_anl	1.009	1.564	1.606	1.612

Table 4. Q4: Speedup comparison for `feval` optimization.

Q4: Optimizing `feval` ***in MATLAB.*** We report the speed-ups enabled by our technique in Table 4, using the running times for McVM's `feval` default dispatcher as baseline. As the dispatcher typically JIT-compiles the invoked function, we also analyzed running times when the dispatcher calls a previously compiled function. In the last column, we show speed-ups from a modified version of the benchmarks in which each `feval` call is replaced by hand with a direct call to the function in use for the specific benchmark.

Unfortunately, we are unable to compute direct performance metrics for the solution by Lameed and Hendren since its source code has not been released. Figures in their paper [9] show that for the very same MATLAB programs the speed-up of the OSR-based approach is on average within 30.1% of the speed-up of hand-coded optimization (ranging from 9.2% to 73.9%); for the JIT-based approach, the average grows to 84.7% (ranging from 75.7% to 96.5%).

Our optimization technique yields speed-ups that are very close to the upper bound given from by-hand optimization; in the *worst case* (`odeRK4` benchmark), we observe a 94.1% when the optimized code is generated on the fly, which becomes 97.5% when a cached version is available. Compared to their OSR-based approach, the compensation entry block is a key driver of improved performance, as the benefits from a better type-specialized whole function body outweigh those from performing a direct call using boxed arguments and return values in place of the original `feval`.

6. Related Work

Early Approaches. OSR has been pioneered in the SELF programming language implementations [8] to enable source-level debugging of optimized code, which requires deoptimizing the code back to the original version. To reconstruct the source-level state, the compiler generates *scope descriptors* recording locations or values of arguments and locals. Execution can be interrupted only at certain interrupt points where its state is guaranteed to be consistent (i.e., method

prologues and backward branches in loops), allowing optimizations between interrupt points. SELF also implements a deferred compilation mechanism [2] for branches that are unlikely to occur at run-time.

Java Virtual Machines. The success of the Java language has drawn more attention to the design and implementation of OSR techniques, as bytecode interpreters began to work along with JIT compilers. In the high-performance HotSpot Server JVM [12] performance-critical methods are identified using method-entry and backward-branches counters; when the OSR threshold is reached, the runtime transfers the execution from the interpreter frame to an OSR frame and thus to compiled code. Deoptimization is performed when class loading invalidates inlining or other optimization decisions: execution is rolled forward to a safe point, at which the native frame is converted into an interpreter frame.

The Jikes RVM uses an OSR mechanism [6] that extracts a scope descriptor from a thread suspended at a method's entrypoint or backward branch, creates specialized code to setup the stack frame for the optimized compiled code and resumes the execution at the desired program counter. OSR is used as part of a profile-driven deferred compilation mechanism. A more general solution is proposed in [18], with the OSR implementation decoupled from program code to ease more aggressive specializations triggered by events external to the executing code (e.g., class loading). Execution state information is maintained in a variable map that is incrementally updated across a set of compiler optimizations.

In the Graal VM, which is centered on the principle of speculative optimizations, the execution falls back to the interpreter during deoptimization, while a runtime function restores the stack frames in the interpreter using the metadata associated with the deoptimization point [4, 5, 22].

Prospect. Prospect [20] is an LLVM-based framework for parallelizing a sequential application. The IR is instrumented through two LLVM passes to enable switching at run-time between a slow and a fast variant of the code, which are both compiled statically. Helper methods are used to save and eventually restore registers, while stack-local variables are put on a separate `alloca` stack rather than on the stack frame so that the two variants result into similar and thus interchangeable stack layouts.

Other Related Work. In tracing JIT compilers deoptimization techniques are used to safely leave an optimized trace when a guard fails. SPUR [1] is a trace-based JIT compiler for Microsoft's Common Intermediate Language (CIL) with three levels of JIT-ting plus a transfer-tail JIT used to bridge the execution from an instruction in a block generated at the second or third level to a safe point for deoptimization to the first JIT level. In RPython, guards are implemented as a conditional jump to a trampoline that analyzes resume information for the guard and executes compensation code to leave the trace; resume data is compactly encoded by sharing parts of the data structure between subsequent guards [17]. A similar approach is used in LuaJIT, where sparse snapshots are taken to enable state restoration when leaving a trace [13].

7. Conclusions

In this paper, we have presented an OSR framework that introduces novel ideas and combines features of extant techniques that no previous solution provided simultaneously. Relevant aspects include platform independence [10], generation of highly optimized continuation functions [6], and performing deoptimization without the need for an interpreter as fallback [1].

Two novel features we propose are OSR with compensation code, which allows extending the range of points where OSR transitions can be fired, and the ability to inject OSR points at arbitrary locations. Using these features, we have shown how to improve the state of the art of `feval` optimization in MATLAB virtual machines. In our implementation, encoding compensation code is currently delegated to the front-end. Future work may investigate automatic ways to build it for certain classes of compiler optimizations.

We have investigated the feasibility of our approach in LLVM, showing that it is efficient in practice. We expect it to be fully portable to other VM frameworks, as the OSR semantics is encoded entirely at intermediate code level without using any special LLVM feature.

Acknowledgements. We wish to thank Jan Vitek, Petr Maj, Karl Millar, and Olivier Flückiger for many enlightening discussions. We are especially grateful to Jan for sparking our interest in this exciting line of research. We would also like to thank Tomas Kalibera and the anonymous CGO and CGO-PPoPP AE referees for their many useful comments.

Live Version of this Paper. We complement the traditional scholarly article publication model by maintaining a live version of this paper at `https://www.authorea.com/users/55853/articles/66046` and `https://github.com/camild/article-llvm-osr`. The live version incorporates continuous feedback by the community, providing post-publication fixes, improvements, and extensions.

References

[1] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A Trace-based JIT Compiler for CIL. In *OOPSLA '10*, pages 708–725, New York, NY, USA, 2010. ACM.

[2] C. Chambers and D. Ungar. Making Pure Object-oriented Languages Practical. In *OOPSLA '91*, pages 1–15, New York, NY, USA, 1991. ACM.

[3] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB Through Just-in-time Specialization. In *CC'10/ETAPS'10*, pages 46–65, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An Intermediate Representation for

Speculative Optimizations in a Dynamic Compiler. In *VMIL '13*, pages 1–10, New York, NY, USA, 2013. ACM.

[5] G. Duboscq, T. Würthinger, and H. Mössenböck. Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler. In *PPPJ '14*, pages 187–193, New York, NY, USA, 2014. ACM.

[6] S. J. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *CGO '03*, pages 241–252. IEEE Computer Society, 2003.

[7] B. Fulgham and I. Gouy. The Computer Language Benchmarks Game. `http://benchmarksgame.alioth.debian.org/`. Accessed: 2016-01-13.

[8] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *PLDI '92*, pages 32–43, New York, NY, USA, 1992. ACM.

[9] N. A. Lameed and L. J. Hendren. Optimizing MATLAB Feval with Dynamic Techniques. In *DLS '13*, pages 85–96, New York, NY, USA, 2013. ACM.

[10] N. A. Lameed and L. J. Hendren. A Modular Approach to On-stack Replacement in LLVM. In *VEE '13*, pages 143–154, New York, NY, USA, 2013. ACM.

[11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04*, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[12] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ Server Compiler. In *JVM'01*, Berkeley, CA, USA, 2001. USENIX Association.

[13] M. Pall. LuaJIT 2.0 intellectual property disclosure and research opportunities. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`. Accessed: 2016-01-13.

[14] F. Pizlo. Introducing the WebKit FTL JIT. `https://www.webkit.org/blog/3362/` Accessed: 2016-01-13.

[15] S. Radpour, L. Hendren, and M. Schäfer. Refactoring MATLAB. In *CC'13*, pages 224–243, Berlin, Heidelberg, 2013. Springer-Verlag.

[16] G. Recktenwald. *Numerical Methods with MATLAB: Implementations and Applications*. Featured Titles for Numerical Analysis Series. Prentice Hall, 2000.

[17] D. Schneider and C. F. Bolz. The Efficient Handling of Guards in the Design of RPython's Tracing JIT. In *VMIL '12*, pages 3–12, New York, NY, USA, 2012. ACM.

[18] S. Soman and C. Krintz. Efficient and General On-Stack Replacement for Aggressive Program Specialization. In *PLC'06*, pages 925–932, 2006.

[19] T. Suganuma, T. Yasue, and T. Nakatani. A Region-based Compilation Technique for Dynamic Compilers. *ACM Trans. Program. Lang. Syst.*, 28(1):134–174, Jan. 2006.

[20] M. Süsskraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer. Prospect: A Compiler Framework for Speculative Parallelization. In *CGO '10*, pages 131–140, New York, NY, USA, 2010. ACM.

[21] The McLab project. Sable McVM. `https://github.com/Sable/mcvm`. Accessed: 2016-01-13.

[22] 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 *Onward! 2013*, pages 187–204, New York, NY, USA, 2013. ACM.