Lightweight On-Stack Replacement in Languages with Unstructured Loops

Matt D'Souza
mwdsouza@uwaterloo.ca
University of Waterloo
Ontario, Canada

Gilles Duboscq
gilles.m.duboscq@oracle.com
Oracle Labs
Zurich, Switzerland

Abstract

On-stack replacement (OSR) is a popular technique used by just in time (JIT) compilers. A JIT can use OSR to transfer from interpreted to compiled code in the middle of execution, immediately reaping the performance benefits of compilation. This technique typically relies on loop counters, so it cannot be easily applied to languages with unstructured control flow. It is possible to reconstruct the high-level loop structures of an unstructured language using a control flow analysis, but such an analysis can be complicated, expensive, and language-specific. In this paper, we present a more lightweight strategy for OSR in unstructured languages which relies only on detecting backward jumps. We design a simple, language-agnostic API around this strategy for language interpreters. We then discuss our implementation of the API in the Truffle framework, and the design choices we made to make it efficient and correct. In our evaluation, we integrate the API with Truffle's LLVM bitcode interpreter, and find the technique is effective at improving start-up performance without harming warmed-up performance.

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

Keywords: on-stack replacement, unstructured loops, Truffle, bytecode interpreter, partial evaluation

ACM Reference Format:
Matt D'Souza and Gilles Duboscq. 2021. Lightweight On-Stack Replacement in Languages with Unstructured Loops. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '21), October 19, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486606.3486782

VMIL '21, October 19, 2021, Chicago, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9109-2/21/10…$15.00
https://doi.org/10.1145/3486606.3486782

1 Introduction

In systems with a method-based just in time (JIT) compiler, an application will not reach peak performance until its hot code paths are compiled. If the application contains long-running methods, it can take even longer to warm up: the JIT can compile such methods, but any ongoing activations cannot simply switch to the optimized code; the method is doomed to run slowly until it finishes executing.

On-stack replacement (OSR) can alleviate this problem. OSR is a general technique to transfer between different versions of a method in the middle of execution. By using OSR to switch from interpreted to optimized code, a runtime system can improve the start-up performance for certain classes of programs. Usually, the runtime system uses loop counters to trigger OSR, but this is not easy in languages with unstructured control flow (such as JVM bytecode [17] or LLVM bitcode [16]), since they do not have explicit loops. Mosaner et al. [19] address this problem by reconstructing loops using a control flow analysis.

In this paper, we present a simpler approach to OSR for unstructured languages written in an interpreter framework. Our work relies on the insight that backward jumps in unstructured code generally fulfil the same role as loops in the implementation of an OSR system. Usually, back-edges are trivial to detect, which enables a more lightweight strategy for OSR in unstructured language interpreters (Section 3). In fact, systems like HotSpot [20] and PyPy [3] already use backward jumps for OSR. We present a simple API to bring our strategy to interpreters written in a partially evaluating language framework (Section 4). We implemented this API in the Truffle framework [25] and discuss some relevant design choices which make it perform well on Truffle (Section 5). In our evaluation, we modified Truffle's LLVM interpreter to support our flavour of OSR, and find that the technique is effective at improving the start-up performance of applications with long-running loops (Section 6).

Concretely, we make the following contributions:

- a summary of an existing technique for OSR using backward jumps,
- a simple API for language implementation frameworks to provide to interpreters,
- an implementation of the API in the Truffle framework,
- an evaluation of the technique's efficacy on Truffle's LLVM interpreter.

2 Background

In this section, we provide background context for our work. First, we discuss on-stack replacement (OSR) as it is used for improving warm-up. Then, we briefly introduce unstructured languages. We then give a short overview of the Truffle framework used to implement our technique. Finally, we discuss an existing technique for unstructured OSR on Truffle.

2.1 On-Stack Replacement (OSR)

On-stack replacement is a common technique used in systems with a JIT to transfer execution between different versions of code [10, 13]. Though OSR has a wide range of applications [7, 15], in this paper we are concerned with using OSR to speed up a program by transferring from interpreted code to compiled code. In the case of transferring execution from compiled code back to the interpreter, we will use the term *deoptimization*.

At a high level, a system which uses OSR to improve warm-up performance:

1. Detects when a method runs for a long time
2. Schedules the method for compilation
3. Transfers execution to the optimized code (when compilation completes), where it finishes executing the method

OSR is usually designed around loops. An OSR system can identify long-running methods by counting loop iterations. If a loop runs for enough iterations, it would likely benefit from OSR, so the system will schedule the method for compilation. When compiling a method, there is the question of selecting a transfer point: a location in the method where the interpreter can transfer to the compiled code. Compilation is usually performed asynchronously, so the transfer point should be a point the interpreter is likely to reach again after compilation completes (so that the transfer can actually occur). Since OSR candidates are already running in long loops, loop headers are commonly chosen as the transfer point.

A particularly challenging aspect of OSR is transferring state between versions. The optimized code may use a completely different frame layout, in which case the interpreter frame must be transformed to the expected layout [6]. If the compiled code has the potential to deoptimize (e.g., if the JIT does speculative compilation), a related challenge is reconstructing interpreter state from the compiled state [8, 12].

2.2 Unstructured Languages

For the purposes of this paper, an *unstructured* language is a language without explicit looping abstractions. Instead of `while` or `for` loops, such a language may use `goto` statements to perform jumps to arbitrary program locations.

Unstructured control flow is generally seen in bytecode or assembly languages with a low level of abstraction. An interpreter for such a language can be implemented as a dispatch loop, like the one seen in Listing 1. Dispatch loops use an instruction index (`idx`) to iterate through a sequence of instructions (`instructions`) and execute each instruction sequentially. Special instructions manipulate the instruction index to implement control flow (like `JUMP_IF_NOT_ZERO`, which jumps to the index indicated by the second operand if the first operand is nonzero).

```
class DispatchLoop {
  byte[] instructions;
  Object execute(frame) {
    idx = 0;
    while(true) {
      switch(instructions[idx]) {
        case INC:
          frame.set(op1, frame.get(op1) + 1);
          idx = idx + 2;
        case JUMP_IF_NOT_ZERO:
          if (frame.get(op1) != 0) {
            idx = op2;
          } else {
            idx = idx + 3;
          }
        ... // more operators
      }
    }
  }
}
```

Listing 1. Dispatch loop pseudocode for a simple bytecode language with unstructured control flow. The `frame` contains program state.

```
class AddNode extends Node {
  Node left;
  Node right;
  public int execute(Frame f) {
    return left.execute(f) + right.execute(f)
  }
}
```

Listing 2. A simple `AddNode` in Truffle which executes its two operands and returns their sum.

2.3 The Truffle Framework

Truffle is a framework for building high-performance language implementations [25]. A Truffle language implementation is written in Java, typically as an abstract syntax tree (AST) interpreter. These interpreters define `Node`s with methods defining how to execute them (like in Listing 2). Though Truffle has historically been designed for ASTs, it also supports dispatch loop interpreters (such as [14, 22]), wherein each method is implemented as a monolithic dispatch `Node`.

In the remainder of this section, we discuss two parts of Truffle which are relevant to our work.

2.3.1 Partial Evaluation.

When running Truffle with the Graal compiler [25], it uses partial evaluation (PE) to optimize the interpretation of ASTs [11, 24]. In PE, the compiler aggressively constant-folds interpreter code, treating the AST being compiled as constant. For example, PE can replace a dispatch loop for a bytecode sequence $BC_1, \ldots, BC_n$ with code which sequentially executes BC_1 through BC_n, removing the loop dispatch overhead.

Partial evaluation enables several optimizations, including escape analysis [5, 23], wherein the compiler detects when an object does not escape its allocating context. If an object does not escape, the compiler can avoid heap allocation, elide synchronization, and use scalar replacement [4, 23] to replace objects with the scalars they comprise.

Scalar replacement is especially important for optimizing Truffle `Frame`s, which contain the state of an interpreted program. With scalar replacement, the interpreted program's local variables (normally accessed through the `Frame`'s getter and setter methods) can be treated like regular Java variables by the compiler. This effectively eliminates the overhead of `Frame` accesses, and allows the compiler more flexibility to optimize [25]. Our implementation includes a state transfer mechanism designed to support scalar replacement of `Frames`.

A key aspect of the Truffle framework is to allow language implementers to take advantage of deoptimization. One example is the `@CompilationFinal` annotation which can be used to mark fields that PE should consider final when constant folding. Changing these fields is permitted as long as the implementer inserts deoptimization directives to invalidate code compiled using old constants.

2.3.2 Loop-Based OSR.

When a Truffle interpreter uses the built-in `LoopNode` class to implement loops, it gets OSR for free. The `LoopNode` executes a loop body node repeatedly until it indicates that the loop should exit. Internally, the `LoopNode` tracks the number of loop iterations, and requests compilation once an OSR threshold is exceeded. Once a compiled version of the `LoopNode` is available, it will call it, passing its `Frame` (the object containing all program state) along as argument.

`LoopNodes` can be used when loops are reducible [9] and the interpreter knows which code is contained in a loop. This is often the case for structured languages, since they use explicit loop syntax (e.g., `while` loops) and do not permit jumps into the middle of a loop. Unstructured languages generally cannot use `LoopNodes` because the loops are not easy to determine (and control flow may be irreducible).

2.4 Unstructured OSR Using Loop Reconstruction

Performing OSR in an unstructured language is difficult because there are no explicit loops. Mosaner et al. [19] address this problem by reconstructing loops from unstructured code.

```
1  # inputs: x and y
2  SET result 0
3  outer:
4    SET temp y
5    inner:
6      INC result
7      DEC temp
8      JUMP_IF_NOT_ZERO temp inner
9    DEC x
10   JUMP_IF_NOT_ZERO x outer
11 RETURN result
```

Listing 3. A pseudocode method to multiply two numbers. The language does not contain structured loops.

The technique consists of a few steps:

1. First, construct a control flow graph of the program.
2. Next, perform a depth-first traversal over the basic blocks to detect loops.
3. Then, traverse the basic blocks inside each loop to determine nesting and successor relations (these are necessary for technical reasons).
4. During execution, whenever the interpreter dispatches to an instruction marked as a loop header, collect loop counts and perform loop-based OSR as usual.[1]

This technique is complex, using a control flow analysis with multiple traversals. The need to detect irreducible control flow (i.e., multiple entry points to a loop body) further increases the complexity of the implementation [9, 19]. Since loop reconstruction must happen before the code runs, this requires extra start-up time when loading a new application. In this work, we present a simpler solution which does not require a control flow analysis.

3 Using Back-Edges for OSR

In this section we discuss the high-level technique. We use Listing 3 as a running example; this method computes the product of inputs `x` and `y` using repeated addition.

The key idea behind the approach is to use backward jumps (or *back-edges*) instead of loops for OSR. Conceptually, back-edges fulfil a similar role to loops in structured OSR. This approach is already used by the Java HotSpot virtual machine [20], and is similar to how PyPy performs meta-tracing [3]. To our knowledge, we are the first to apply this technique to a partially-evaluating method-based framework like Truffle.

3.1 Selecting OSR Targets

First, an OSR system should detect methods which run for a long time in the interpreter. We count backward jumps for this purpose. In Listing 3, there are back-edges on lines 8 and 10. When these back-edges are taken, we increment a counter

[1] In Truffle, loops are reified as `LoopNodes`, so OSR is handled transparently.

for the method. If the counter exceeds a pre-defined threshold, the method is "hot" and we select it for OSR compilation. Dispatch loop interpreters (see Section 2.2) can easily detect back-edges by comparing the current and next instruction indices, so this technique can often be implemented with little effort.[2]

For the purposes of triggering OSR, all backward jumps in a method use the same counter. Intuitively, the counter is important for detecting a method's hotness, but detecting the hotness of individual loops is not as important. A single counter is also simpler to implement and introduces less overhead for the interpreter than a per-target counter.

3.2 Compiling OSR Targets

When compiling a method for OSR, the system should select a point (or points) in the code where it can transfer control to the OSR target. The location should be somewhere the interpreter is likely to hit again, so that it can perform the transfer. We use the targets of back-edges for this purpose. The runtime system optimizes and compiles the method starting at the given code location.

For example, in Listing 3, if a jump on line 8 triggers OSR compilation, we will compile an OSR target starting from `inner`. Once compilation succeeds, the next time the interpreter jumps back to `inner` it can transfer execution to the compiled code.

In some situations, using just a single transfer point is insufficient. Consider if we instead compile an OSR target starting from `outer`: the interpreter will not be able to transfer execution until `outer` is hit again, even if the inner loop runs for a long time. To account for cases like this, we permit multiple transfer points. Once the back-edge threshold is exceeded, we request compilation for each unique back-edge target we encounter (as is done in [20]). Thus, in Listing 3, we would only be "stuck" in the inner loop until the OSR system can compile an OSR target starting from `inner`.

A performance concern with OSR is the additional pressure it puts on the compilation system. Compiling a target for each back-edge is, inevitably, a wasted effort since only one target will ever actually be used. Also, since unstructured languages do not require back-edges to correspond to loops, we could waste effort compiling OSR targets for locations the interpreter is unlikely to hit again. We can avoid this issue by maintaining separate counters for each target. However, since the languages we evaluated do not contain such back-edges, our implementation uses a single counter. One way our implementation minimizes wasted compilations is with polling: once the threshold is reached, instead of potentially requesting compilation at every back-edge, we only request compilation periodically. This reduces the likelihood that compilation would be requested for an infrequent back-edge.

[2]This approach assumes that there exists a total ordering for instructions (e.g., by bytecode index or line number).

```
interface UnstructuredOSRNode {
  Object executeOSR(osrFrame, target,
      interpreterState);
  void setMetadata(metadata);
  Metadata getMetadata();
}
boolean pollOSR() {...}
Object tryOSR(parentFrame, target,
      interpreterState, beforeOSR) {...}
```

Listing 4. Our API for unstructured OSR.

3.3 Transferring Control to an OSR Target

Finally, once a compiled OSR target is available, we can transfer execution to it. Since the interpreter already detects back-edges, and the OSR targets start from back-edge targets, it is convenient to transfer control when taking a back-edge.

For example, in Listing 3, suppose we compile an OSR target starting from `inner`. When we detect a back-edge to `inner`, we can check if an OSR target exists; if it does, we can transfer control to the OSR target. All of the current method's state (i.e., `x`, `y`, `result`, and `temp`) should be copied over during this step. We cannot simply reuse the interpreter activation, because the generic interpreter and dedicated OSR target usually represent the method's state differently. While it is logically the same representation, in the OSR target, the partial evaluator has specialized it to the specific method and entry-point.

4 An API for Language Interpreters

In this section, we provide an overview of the high-level API we designed to be used in interpreters for unstructured languages, and provide an example integration. We also comment on how this API interacts with a system which uses partial evaluation.

4.1 The API

The API consists of the `UnstructuredOSRNode` interface and the helper functions in Listing 4. A dispatch loop node must do a few things to integrate with this API.

First, it must implement the `UnstructuredOSRNode` interface. The `executeOSR` method should define how to resume execution of the node, typically by running the dispatch loop starting from `target`. The OSR system will call this method when it performs OSR. Since the `target` is a compile-time constant, the partial evaluator can convert `executeOSR` into code executing a sequence of instructions starting from index `target`. The `setMetadata` and `getMetadata` methods should simply proxy accesses to a field defined by the node. The OSR system needs a place to store metadata related to each dispatch node (e.g., a back-edge counter, a map of OSR targets), so this is a natural place to store it.[3]

Second, the node should detect back-edges during dispatch and notify the OSR system using the `pollOSR` helper. This method returns `true` once the method is hot enough to warrant OSR compilation. If it returns `true`, the interpreter should call `tryOSR`. This helper transparently handles the entire OSR process for the user. It counts back-edges, requests compilation, and transfers to compiled OSR code (which calls `executeOSR`). If OSR succeeds, it returns a non-`null` value.[4] The main parameters to `tryOSR` are the current execution state (`parentFrame`) and the destination of the back-edge (`target`). There are two optional parameters: `interpreterState` can contain internal interpreter state which gets passed to `executeOSR` (e.g., data pointers); `beforeOSR` is a callback which can be invoked before calling OSR code (e.g., to notify instrumentation hooks).

4.2 Example Integration

To illustrate the simplicity of integration, we modify the interpreter from Listing 1 to use our API. Listing 5 contains the changes necessary to support OSR.

First, we modify the `DispatchLoop` class to implement the `UnstructuredOSRNode` interface. For code reuse, we extract the dispatch loop into a helper (lines 10–27) which can be called by both `executeOSR` and the existing `execute` method (lines 5 and 8). We also add a `metadata` field (line 3) and proxy accesses to it through `getMetadata` and `setMetadata` (omitted for brevity).

Then, we modify the dispatch loop to detect backward jumps and call `pollOSR` (line 16). If the call returns `true`, we try to perform OSR (line 17). If the result is non-`null`, then OSR occurred, and we can return early with the result (lines 18–20). If the PE framework supports compiler directives to test the current execution context (e.g., `IN_INTERPRETER`), they can be used to ensure back-edge detection only runs in the interpreter. Since `IN_INTERPRETER` is `false` in compiled code, partial evaluation can constant-fold the condition on line 16 to `false`, and hence lines 16–21 completely disappear in the compiled code.

5 Truffle Implementation

The desire for a simple OSR interface is sometimes in conflict with the complex performance and correctness constraints of a system like Truffle. In this section, we discuss some of the design challenges and tradeoffs of our Truffle implementation.

```
1  class DispatchLoop implements
       UnstructuredOSRNode{
2    byte[] instructions;
3    Metadata metadata;
4    Object execute(frame) {
5      return executeFromIdx(frame, 0);
6    }
7    Object executeOSR(osrFrame, target, ...) {
8      return executeFromIdx(osrFrame, target);
9    }
10   Object executeFromIdx(frame, idx) {
11     while(true) {
12       switch(instructions[idx]) {
13         ...
14         case JUMP_IF_NOT_ZERO:
15           if (frame.get(op1) != 0) {
16             if (op2 < idx && IN_INTERPRETER &&
                     pollOSR()){
17               result = tryOSR(frame, op2,...);
18               if (result != null) {
19                 return result;
20               }
21             }
22             idx = op2;
23           } else { ... }
24         ...
25       }
26     }
27   }
28 }
```

Listing 5. Modification of Listing 1 to support OSR.

5.1 Managing Metadata

The metadata for our implementation consists of a back-edge counter, a map of OSR targets (indexed by starting index), and info about the frame layout (for copying).

The vast majority of methods do not require OSR, so it is wasteful to allocate all of these objects for each method. Accordingly, our solution lazily initializes metadata. First, we partially initialize the metadata object with a counter when a back-edge is reported. Then, in the rare case that OSR is requested, we fully initialize the OSR targets map and frame layout metadata. Lazy initialization can be perilous in the presence of multiple threads [21], so we take extra care to ensure that fields are safely initialized with respect to Java's memory model [18].

5.2 Implementing `pollOSR` and `tryOSR`

In this section we discuss the design of our helper functions; we provide pseudocode for the method in Listing 6.

The `pollOSR` method first reads the method's OSR metadata and increments its counter (lines 2–3). It then inspects the result, returning `false` if the OSR threshold has not been reached (line 4). If the OSR threshold is reached, `pollOSR`

[3] In a language like Java, interfaces cannot declare fields, so the accessors define a "logical" field the implementing class must provide storage for.

[4] In the Truffle framework, `null` is never a valid value; we could alternatively define a marker object to indicate "no result".

```
1  boolean pollOSR() {
2    metadata = getOrInitializeMetadata()
3    count = ++metadata.count;
4    return (count >= THRESHOLD) &&
5           (count % POLLING_INTERVAL == 0);
6  }
7
8  Object tryOSR(parentFrame, target,
       interpreterState, beforeOSR) {
9    osrTarget = getMetadata().getCompileTarget(
         target, interpreterState);
10   if (osrTarget.isCompiling()) return null;
11   beforeOSR(target);
12   return osrTarget.call(parentFrame);
13 }
```

Listing 6. Pseudocode for `pollOSR` and `tryOSR`.

returns `true` every `POLLING_INTERVAL` back-edges (line 5)[5]. Since compilation will take much longer than a single loop iteration, it would not be helpful to try OSR on every single back-edge; we poll to avoid unnecessary work.

In `tryOSR`, we request compilation for the given `target` or retrieve the call target if it already exists (line 9). If the OSR target is still being compiled, we return `null` (line 10); otherwise, we call the `beforeOSR` callback and then jump to OSR code (lines 11–12). While many OSR implementations use a tail call—replacing the interpreter activation with a compiled OSR activation—Truffle does not have such low-level control over the stack, so we push the OSR activation on top of the interpreter activation.

A key goal in this design is to minimize the amount of work done in the common case. An interpreter will call `pollOSR` on every back-edge, but only rarely call `tryOSR`, because few methods actually need OSR. Thus, we perform minimal work inside `pollOSR`, avoiding locking, allocation, and volatile fields[6]. In the rare case that `tryOSR` is called, we take a slower path: we use locking, allocate the `interpreterState` object and `beforeOSR` callback, and use a thread-safe map so that other threads may reuse the OSR compilation. We tolerate the extra allocation and synchronization overhead since this path occurs infrequently.

5.3 Transferring State

Transferring state is challenging in many OSR systems, since the two code versions may use different frame layouts; the need to reconstruct state after deoptimization further complicates things. By calling the OSR target through Truffle's existing call mechanisms, all of these complications are handled by the framework. We benefit greatly from Truffle here; in another system, this process could be much more complex.

[5]Our `THRESHOLD` is a multiple of our `POLLING_INTERVAL` (1024), so `pollOSR` returns `true` when it first hits `THRESHOLD`.

[6]This sacrifices counter consistency, but the tradeoff is acceptable to keep a fast common path.

```
osrFrame = parentFrame
...
while(osrFrame.get("n") > 0) {
  osrFrame.set("result",
    osrFrame.get("result") * osrFrame.get("n")
  );
  osrFrame.set("n", osrFrame.get("n") - 1);
}
return osrFrame.get("result");
```

Listing 7. Pseudocode for OSR with parent frame reuse.

```
n = parentFrame.get("n")
result = parentFrame.get("result")
...
while (n > 0) {
  result = result * n;
  n = n - 1;
}
return result;
```

Listing 8. Pseudocode for OSR with parent frame copying.

However, the naïve approach of passing the parent `Frame` object to the OSR code is not ideal. Recall that the compiler uses scalar replacement to replace a `Frame` object with the set of variables it comprises (see Section 2.3.1). This transformation is only safe when the `Frame` is allocated inside the compiled method and does not become reachable outside of it. If we simply pass the interpreter `Frame` to the OSR code, the `Frame` escapes the OSR code and is not eligible for scalar replacement. As a result, variable accesses get proxied through the `Frame` object, which is slower and less amenable to optimization (e.g., Listing 7). Instead, when possible, we copy each variable from the parent `Frame` into a new OSR `Frame`. Since this new `Frame` does not escape, the compiler can successfully perform scalar replacement, removing all `Frame` indirection entirely (e.g., Listing 8).

We add two hooks to the interface in Listing 4. The first hook, `copyIntoOSRFrame`, copies state into the OSR `Frame`. By default, it copies over each variable, but it may be overridden for finer control over the copy. For example, some implementations may wish to `null` out references in the parent `Frame` so the garbage collector can reclaim objects earlier. The second hook, `restoreParentFrame`, can be used to restore state back into the parent `Frame`. By default, it does nothing, since most languages likely do not need the `Frame` after returning from OSR.

Copying introspects the `Frame` layout, but the layout can change dynamically (e.g., if a new variable is introduced). We use `@CompilationFinal` directives to tell the compiler to treat the layout as a constant (and insert deoptimization points in case it changes). We provide this copying code in a helper to simplify integration.

5.4 Handling `Frame` Introspection

Because we copy the parent `Frame` contents into the OSR `Frame`, once OSR begins executing, the two can get out of sync. We must be careful when this happens, since Truffle provides mechanisms to introspect `Frames`.

Truffle allows `Frames` to be materialized, which lets interpreters pass and store them like regular objects. Because of materialization, it is possible that when we perform OSR, some code (e.g., a debugger) has a reference to the parent `Frame`. If it inspects this `Frame` during OSR, the variables may be stale and not reflect the actual state of the program. To remedy this, we do not copy the parent `Frame` if it has materialized. Instead, we reuse it.

Additionally, Truffle provides a stack walking utility which returns a stack trace containing the `Frame` for each ongoing method. Since our OSR implementation pushes a new OSR activation onto the stack (instead of replacing the interpreter activation), we modify stack walking to return the newer OSR `Frame` and ignore the parent `Frame`.

5.5 Compilation & Invalidation

Since Truffle compilation is speculative, compiled code can be invalidated when an assumption turns out to be incorrect, or when an unexpected code path is hit. Our implementation must take this dynamic nature of compilation into account.

Truffle can dynamically modify an AST during execution, for instance, to add instrumentation or to type-specialize an operation. If an AST changes, any compiled code which includes that AST must be invalidated. Our implementation registers callbacks to detect changes in OSR nodes so that we can invalidate any compiled OSR targets.

It is also common to lazily initialize the fields of an AST. Since modifying a field can trigger invalidation (e.g., if it is marked `@CompilationFinal`), we encountered scenarios where compiled OSR code would try to initialize a field and immediately invalidate itself, deoptimizing back to the interpreter. We add another hook to Listing 4, `prepareOSR`, which the OSR system calls before compilation. This hook allows language implementations to eagerly initialize any fields and avoid unnecessary invalidation inside OSR code.

6 Evaluation

In this section, we evaluate the efficacy of our technique. Our primary concern is whether it improves start-up performance for programs with long-running methods, but we also assess its impact on warmed-up (steady state) performance. We also briefly discuss our experience using the API.

6.1 Setup

We evaluate our technique using Sulong, Truffle's LLVM bitcode interpreter [22]. We compare our OSR strategy with the loop reconstruction technique from [19].

We reuse benchmarks from [19]: several microbenchmarks from the *Computer Language Benchmarks Game*[7] (mandelbrot, nbody, fannkuchredux, revcomp, binarytrees, and pidigits) as well as the gzip[8] and whetstone[9] benchmarks. We include benchmarks which enjoyed a performance improvement from loop reconstruction, but also a couple of benchmarks which did not; it is desirable for our technique to have minimal performance impact on these benchmarks.

For each benchmark, we perform 10 runs in each of the following configurations:

- Back-Edge: OSR using back-edges (our technique)
- Loop: OSR using loop reconstruction (from [19])
- None: No OSR

Each run executes the benchmark for at least 50 consecutive iterations to give the system time to warm up.

We perform all benchmarks on a system with an Intel Core i7-10700 processor with 16 virtual cores at 2.90GHz. It has 32GB of memory and runs Ubuntu 20.04. We run Sulong on GraalVM Enterprise Edition 21.2.0 in a native image.

6.2 Start-up Performance

To evaluate the start-up performance of our technique, we consider the execution time of the first 10 iterations of each benchmark. In Figure 1 we plot the warm-up curves for each of the benchmarks which saw a performance improvement from OSR in [19]. We omit the other benchmarks, since they do not exhibit interesting start-up behaviour. We also omit None from the whetstone plot to keep the scale readable; its first two iterations take 99 seconds on average, and its later iterations converge to roughly the same times as Back-Edge and Loop.

We find that our technique is at least as effective as loop reconstruction for improving start-up times. The first iteration of each benchmark using Back-Edge is anywhere from 3× (fannkuchredux) to 35× (whetstone) faster than None due to OSR. Later iterations for either OSR technique improve marginally as regular whole-method compilation kicks in. Occasionally, due to issues with deoptimization, the OSR techniques briefly slow down (e.g., iteration 3 of nbody) before re-optimizing.

In 4 of the 8 benchmarks, the early iterations of Back-Edge are slightly faster than Loop. For example, the first iteration of whetstone takes almost 50% more time with Loop than Back-Edge (4.3s versus 2.9s). There are many confounding factors in Truffle compilations which make it difficult to confidently state a root cause. We suspect that Back-Edge benefits from not having to perform a control flow analysis at start-up. Another possible reason is that Back-Edge can compile beyond the end of a loop—even through other loops—whereas Loop may need to compile multiple loops

[7]https://benchmarksgame-team.pages.debian.net/benchmarksgame
[8]http://people.csail.mit.edu/smcc/projects/single-file-programs/
[9]http://www.netlib.org/benchmark/whetstone.c

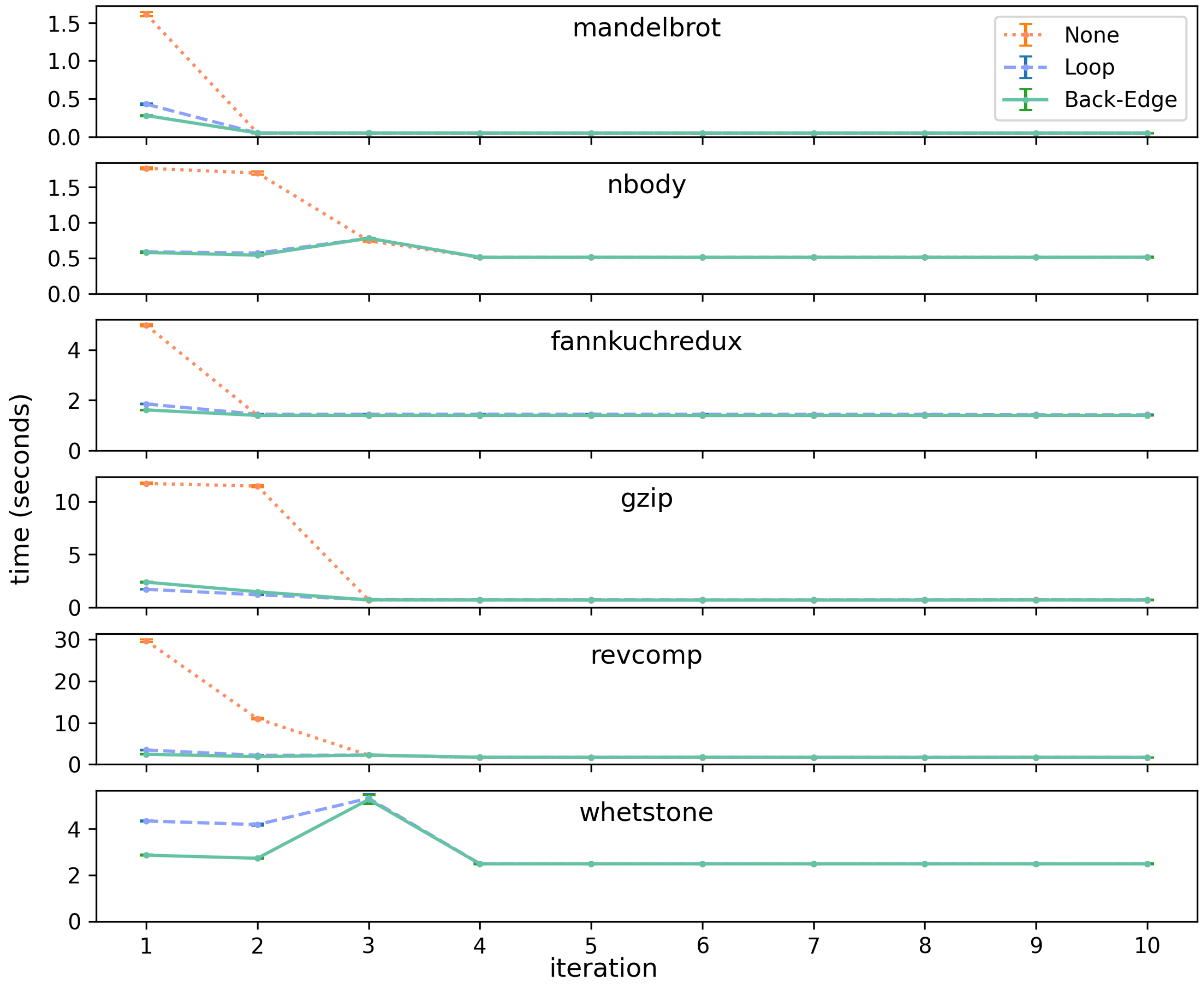

Figure 1. Warm-up curves for benchmarks where OSR affects start-up performance. We plot the mean (of 10 iterations) and depict the standard deviation using error bars. The binarytrees and pigidits benchmarks do not trigger OSR, so we omit their curves.

separately. In the gzip benchmark, Loop outperforms Back-Edge. We see Back-Edge spends twice as long compiling OSR targets, which may explain the discrepancy.

6.3 Warmed-up Performance

Though OSR is a technique to help warm-up time, it should not negatively affect performance after warm-up, either. To assess the impact of our technique on warmed-up performance, we compare warmed-up times across configurations. We measure warmed-up time as the mean execution time over the last 10 iterations. To ensure the program is warmed up, we manually check the warm-up curves and verify that execution time does not change significantly across these iterations.

We present mean warmed-up times in Table 1. Across all benchmarks, including ones which do not benefit from OSR, Back-Edge has comparable (and sometimes better) warmed-up performance compared to None. We suspect this is because PE can completely eliminate back-edge detection code from compiled code (as discussed in Section 4), resulting in code which looks practically the same as None. In contrast, Loop sometimes lowers the warmed-up performance relative to None (the difference is significant in 3 of 8 benchmarks). As mentioned in [19], this is likely because Loop changes the Truffle AST structure, which can cause compiler optimizations to behave differently. Our solution can thus be seen as a less intrusive OSR technique with respect to warmed-up performance.

Table 1. Warmed-up performance. For each configuration, we present the mean warmed-up time (with standard deviation) and the warmed-up time relative to None (lower is better).

	None			Loop			Back-Edge		
	mean (s)	SD	rel	mean (s)	SD	rel	mean (s)	SD	rel
mandelbrot	0.047	0.001	-	0.060	0.001	1.268	0.047	0.000	0.999
nbody	0.510	0.002	-	0.509	0.002	0.999	0.510	0.002	1.000
fannkuchredux	1.391	0.002	-	1.423	0.001	1.023	1.391	0.002	1.000
gzip	0.701	0.009	-	0.713	0.015	1.018	0.698	0.014	0.996
revcomp	1.690	0.013	-	1.683	0.007	0.996	1.688	0.011	0.999
whetstone	2.485	0.012	-	2.494	0.014	1.003	2.486	0.012	1.000
binarytrees	1.965	0.038	-	1.971	0.080	1.003	1.950	0.030	0.992
pidigits	1.378	0.019	-	1.382	0.032	1.004	1.378	0.015	1.001

6.4 Usability

From our experience, our Truffle API does not require much effort to integrate, and hides much of the complexity that OSR entails. Our integration with Sulong [22] is less than 60 lines of code, whereas the existing loop reconstruction OSR of [19] is over 300 lines. The author who wrote the integration had no prior experience with the Sulong system, but found it straightforward to integrate with the Back-Edge API. We also implemented our technique for Java-on-Truffle [14], which does not have an existing OSR mechanism. The integration is less than 150 lines and enjoys comparable performance benefits.

7 Related Work

Our work is closely related to a few different lines of research. We briefly discuss them here.

OSR was popularized by the SELF VM [13], where it was used to debug optimized code using deoptimization [12]. OSR is also commonly used in Java virtual machines (JVMs). The HotSpot VM [20] uses back-edges in JVM bytecode to perform OSR. Our work uses HotSpot's technique, packaging it in an API for language implementers. Jikes RVM (another JVM) also implements OSR [10]; the authors cite the engineering complexity of OSR as a reason it is not more widely used. Later works attempt to tame this complexity through formalization [15] or by providing OSR in language-independent frameworks [6, 15]. Truffle [25] falls into the latter category, giving structured languages OSR for free through its `LoopNode` interface. Truffle's LLVM interpreter supports unstructured OSR by reconstructing loops [19]; our technique adds a more general, lightweight API for unstructured languages.

Tracing JITs such as Dynamo [1] are also designed around improving the performance of loops. When they detect a hot loop, they collect a *trace* from one iteration and use the trace to compile an optimized version of the loop. RPython, another language implementation framework, supports a modified form of tracing called *meta-tracing* which traces bytecode dispatch loops [3]. Much like our technique, they use back-edges to detect, compile, and optimize hot loops in the interpreted language. They perform constant folding over the instruction stream and index, which can be seen as a limited form of the partial evaluation used in our technique. Pycket is an interesting application of the RPython framework to functional languages [2]. Since looping in functional languages is commonly implemented using function calls (i.e., recursion), detecting loops for functional languages requires a different strategy altogether.

8 Conclusion

On-stack replacement can be an effective way to improve start-up speeds, but its myriad technical challenges are a barrier to its adoption. For unstructured languages, the lack of explicit loops is one such challenge. In this paper, we presented a technique for OSR which uses backward jumps. We argue that back-edges fit the design goals of an OSR system just as well as loops do, and design a simple API around the technique. We implement the API in the Truffle framework, and discuss some of the design challenges we faced. In our evaluation, we compare our technique with a loop reconstruction approach to OSR, and find the technique is as good (if not better) at improving start-up performance without harming warmed-up performance.

There are many aspects of our design which could be explored further. For example, though our metadata storage scheme attempts to minimize memory usage, it still incurs a small overhead for each dispatch node. We would like to evaluate the extent of this overhead, and potentially explore other storage strategies. Additionally, we would like to experiment with different compilation settings. Our evaluation uses second-tier compilation, but it would be interesting to evaluate other schemes (e.g., multi-tier). Graal's compilation queue currently does not treat OSR compilations any differently; we suspect that we can further improve start-up performance by using a scheme which prioritizes OSR compilations. We leave these topics for future work.

Acknowledgments

This research was supported by the GraalVM team at Oracle Labs. We would like to especially thank Christian Humer and Roland Schatz for their invaluable feedback throughout the design and development of this project.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.* 1–12. https://doi.org/10.1145/349299.349303

[2] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G Siek, and Sam Tobin-Hochstadt. 2015. Pycket: a tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 22–34. https://doi.org/10.1145/2784731.2784740

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems.* 18–25. https://doi.org/10.1145/1565824.1565827

[4] Steve Carr and Ken Kennedy. 1994. Scalar replacement in the presence of conditional control flow. *Software: Practice and Experience* 24, 1 (1994), 51–77. https://doi.org/10.1002/spe.4380240104

[5] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. *ACM SIGPLAN Notices* 34, 10 (1999), 1–19. https://doi.org/10.1145/320385.320386

[6] Daniele Cono D'Elia and Camil Demetrescu. 2016. Flexible on-stack replacement in LLVM. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization.* 250–260. https://doi.org/10.1145/2854038.2854061

[7] Daniele Cono D'Elia and Camil Demetrescu. 2018. On-stack replacement, distilled. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 166–180. https://doi.org/10.1145/3192366.3192396

[8] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation without regret: reducing deoptimization meta-data in the Graal compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools.* 187–193. https://doi.org/10.1145/2647508.2647521

[9] Ana M Erosa and Laurie J Hendren. 1994. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94).* IEEE, 229–240. https://doi.org/10.1109/ICCL.1994.288377

[10] Stephen J Fink and Feng Qian. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 241–252. https://doi.org/10.1109/CGO.2003.1191549

[11] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. https://doi.org/10.1023/A:1010095604496

[12] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation.* 32–43. https://doi.org/10.1145/143103.143114

[13] Urs Hölzle and David Ungar. 1994. A third-generation Self implementation: Reconciling responsiveness with performance. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications.* 229–243. https://doi.org/10.1145/191080.191116

[14] Oracle Labs. 2021. *Java on Truffle: Introducing a New Way to Run Java.* Retrieved July 31, 2021 from https://www.graalvm.org/java-on-truffle/

[15] Nurudeen A Lameed and Laurie J Hendren. 2013. A modular approach to on-stack replacement in LLVM. *ACM SIGPLAN Notices* 48, 7 (2013), 143–154. https://doi.org/10.1145/2451512.2451541

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86. https://doi.org/10.5555/977395.977673

[17] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification.* Pearson Education.

[18] Jeremy Manson, William Pugh, and Sarita V Adve. 2005. The Java Memory Model. *ACM SIGPLAN Notices* 40, 1 (2005), 378–391. https://doi.org/10.1145/1047659.1040336

[19] Raphael Mosaner, David Leopoldseder, Manuel Rigger, Roland Schatz, and Hanspeter Mössenböck. 2019. Supporting On-Stack Replacement in Unstructured Languages by Loop Reconstruction and Extraction. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes.* 1–13. https://doi.org/10.1145/3357390.3361030

[20] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, Vol. 1. 1–12.

[21] Bill Pugh et al. 2008. *The "Double-Checked Locking is Broken" Declaration.* Retrieved July 27, 2021 from https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

[22] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing low-level languages to the JVM: Efficient execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages.* 6–15. https://doi.org/10.1145/2998415.2998416

[23] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization.* 165–174. https://doi.org/10.1145/2544137.2544157

[24] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 662–676. https://doi.org/10.1145/3062341.3062381

[25] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software.* 187–204. https://doi.org/10.1145/2509578.2509581