A Modular Approach to On-Stack Replacement in LLVM

Nurudeen Lameed Laurie Hendren

Sable Research Group
School of Computer Science
McGill University, Montréal, Québec, Canada
{nlamee,hendren}@cs.mcgill.ca
http://www.sable.mcgill.ca/mclab

Abstract

On-stack replacement (OSR) is a technique that allows a virtual machine to interrupt running code during the execution of a function/method, to re-optimize the function on-the-fly using an optimizing JIT compiler, and then to resume the interrupted function at the point and state at which it was interrupted. OSR is particularly useful for programs with potentially long-running loops, as it allows dynamic optimization of those loops as soon as they become hot.

This paper presents a modular approach to implementing OSR for the LLVM compiler infrastructure. This is an important step forward because LLVM is gaining popular support, and adding the OSR capability allows compiler developers to develop new dynamic techniques. In particular, it will enable more sophisticated LLVM-based JIT compiler approaches. Indeed, other compiler/VM developers can use our approach because it is a clean modular addition to the standard LLVM distribution. Further, our approach is defined completely at the LLVM-IR level and thus does not require any modifications to the target code generation.

The OSR implementation can be used by different compilers to support a variety of dynamic optimizations. As a demonstration of our OSR approach, we have used it to support dynamic inlining in McVM. McVM is a virtual machine for MATLAB which uses a LLVM-based JIT compiler. MATLAB is a popular dynamic language for scientific and engineering applications that typically manipulate large matrices and often contain long-running loops, and is thus an ideal target for dynamic JIT compilation and OSRs. Using our McVM example, we demonstrate reasonable overheads for our benchmark set, and performance improvements when using it to perform dynamic inlining.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

General Terms Experimentation, Languages, Performance

Keywords On-Stack Replacement, LLVM, MATLAB, JIT Compiler Optimization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'13, March 16–17, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1266-0/13/03. . . $15.00

1. Introduction

Virtual machines (VMs) with Just-in-Time (JIT) compilers have become common place for a wide variety of languages. Such systems have an advantage over static compilers in that compilation decisions can be made on-the-fly and they can adapt to the characteristics of the running program. On-stack replacement (OSR) is one approach that has been used to enable on-the-fly optimization of functions/methods [10, 12, 18, 21]. A key benefit of OSR is that it can be used to interrupt a long-running function/method (without waiting for the function to complete), and then restart an optimized version of the function at the program point and state at which it was interrupted.

LLVM is an open compiler infrastructure that can be used to build JIT compilers for VMs [1, 14]. It supports a well-defined code representation known as the LLVM IR, as well as supporting a large number of optimizations and code generators. LLVM has been used in production systems, as well as in many research projects. For instance, MacRuby is an LLVM-based implementation of Ruby on Mac OS X core technologies[1]; Rubinus[2] is another implementation of Ruby based on LLVM JIT. Unladen-swallow is a fast LLVM implementation of Python[3]. VMKit[4] is an LLVM-based project that works to ease the development of new language VMs, and which has three different VMs currently developed (Java, .Net, and a prototype R implementation). A common theme of these diverse projects is that they could benefit from further on-the-fly optimizations, but unfortunately LLVM does not support OSR-based optimizations. Indeed, we agree with the developers of VMKit who believe that using OSR would enable them to speculate and develop runtime optimizations that can improve the performance of their VMs[5]. Thus, given the value of and need for OSR and the wide-spread adoption of LLVM in both industry and academia, our paper aims to fill this important void and provide an approach and modular implementation of OSR for LLVM.

Implementing OSR in a non-Java VM and general-purpose compiler toolkits such as LLVM requires novel approaches. Some of the challenges to implementing OSR in LLVM include:

(1) At what point should the program be interrupted and how should such points be expressed within the existing design of LLVM, without changing the LLVM IR?

(2) The static single-assignment (SSA) nature of the LLVM IR requires correct updates of control flow graphs (CFGs) of LLVM

[1] http://macruby.org/

[2] http://rubini.us

[3] http://code.google.com/p/unladen-swallow/

[4] Previously http://vmkit.llvm.org and now http://vmkit2.gforge.inria.fr

[5] Private communication with the authors, October 2012.

code, thus program transformations to handle OSR-related control flow must be done carefully and fit into the structure imposed by LLVM.

(3) LLVM generates a fixed address for each function; how then should the code of a new version of the running function be made accessible at the old address without recompiling the callers of the function? This was actually a particularly challenging issue to solve.

(4) The OSR implementation must provide a clean integration with LLVM's capabilities for function inlining.

(5) As there are many users of LLVM, the OSR implementation should not require modifications to the existing LLVM installations. Ideally the OSR implementation could just be added to an LLVM installation without requiring any recompilation of the installation.

We addressed these and other challenges by developing a modular approach to implementing OSR that fits naturally in the LLVM compiler infrastructure.

To illustrate a typical use of our OSR implementation, we have used the implementation to support a selective dynamic inlining optimization in a MATLAB VM. MATLAB [15] is a popular platform for programming scientific applications [17]. It is a dynamic language designed for manipulation of matrices and vectors, which are common in scientific applications [9]. The dynamic features of the language, such as dynamic typing and loading, contribute to its appeal but also prevent efficient compilation. MATLAB programs often have potentially long-running loops, and because its optimization can benefit greatly from on-the-fly information such as types and array shapes, we believe that it is an ideal language for OSR-based optimizations. Thus, we wanted to experiment with this idea in McVM/McJIT [7, 16], an open source VM and JIT for MATLAB, which is built upon LLVM.

The main contributions of this paper are:

Modular OSR in LLVM: We have designed and implemented OSR for LLVM. Our approach provides a clean API for JIT compiler writers using LLVM and clean implementation of that API, which integrates seamlessly with the standard LLVM distribution and that should be useful for a wide variety of applications of OSR.

Integrating OSR with inlining in LLVM: We show how we handle the case where the LLVM inliner inlines a function that contains OSR points.

Using OSR in McJIT for selective dynamic inlining: In order to demonstrate the effectiveness of our OSR module, we have implemented an OSR-based dynamic inliner that will inline function calls within dynamically hot loop bodies. This has been completely implemented in McVM/McJIT.

Experimental measurements of overheads/benefits: We have performed a variety of measurements on a set of 16 MATLAB benchmarks. We have measured the overheads of OSRs and selective dynamic inlining. This shows that the overheads are usually acceptable and that dynamic inlining can result in performance improvements.

The rest of the paper is organized as follows. In Section 2, we classify OSR techniques according to their runtime transition capabilities. In Section 3, we outline the application programming interface (API) and demonstrate the usage our OSR module, from a JIT compiler writer's point of view. In Section 4, we describe the implementation of our API and the integration of inlining. In Section 5, we present a case study of using the OSR support to implement a selective and dynamic inlining of function calls in long-running loops in the McVM JIT compiler for MATLAB. Section 6 reviews some related work upon which we are building. We conclude the paper and highlight some future work in Section 7.

2. OSR Classification

The term OSR is used in the literature [3, 10, 12, 18, 21] to describe a variety of similar, but different, techniques for enabling an on-the-fly transition from one version of running code to another semantically equivalent version. To see how these existing techniques relate to each other, and to our proposed OSR implementation, we propose a classification of OSR transitions, as illustrated in Figure 1.

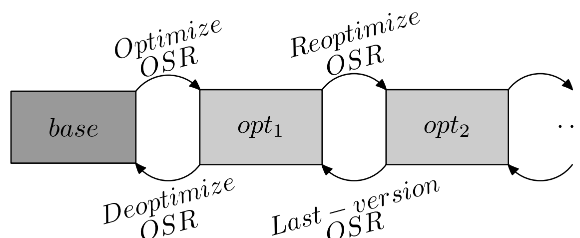

Figure 1. OSR Classification

In most systems with OSR support, the execution of the running code often begins with interpretation or the execution of the code compiled by a non-optimizing base-line compiler. We refer to this version of the running code as the *base* version. This is shown in the darker shaded block of Figure 1.

We call an OSR transition from the *base* version to more optimized code (such as opt_1 in Figure 1) an *Optimize OSR*. The OSR support in the Java HotSpot server compiler [18] uses this kind of transition.

Some virtual machines allow an OSR transition from optimized code such as opt_1 in Figure 1 to unoptimized code (the *base* version). We call this a *Deoptimize OSR* transition. This was the original OSR transition pioneered by Hölzle et al [12] to allow online debugging of optimized code in the SELF [6] virtual machine.

Systems such as the Jikes RVM [3], V8 VM[6], and JavaScriptCore[7] support both *Optimize OSR* and *Deoptimize OSR* transitions. Once a system has deoptimized back to the base code, it can potentially trigger another *Optimize OSR*, perhaps at a higher-level of optimization.

We call a transition from optimized code such as opt_1 to more optimized code such as opt_2 in Figure 1 a *Reoptimize OSR*. Further, we call an OSR transition from more optimized code (e.g., opt_2) to the last version of less optimized code (e.g., opt_1) a *Last-version OSR*.

The OSR technique presented in this paper supports both OSR transitions to a more optimized version and deoptimizations to the last version. Thus, if one starts with the base code, our OSR machinery can be used to perform an *Optimize OSR* transition. From that state, our OSR machinery can be used either as a *Deoptimize OSR* transition to return to the base code (which is the last version of the code), or as a *Reoptimize OSR* to transition to an even more optimized version. Our OSR implementation always caches the last version of the code, so it can also be used to support a *Last-version OSR* to transition from a higher-level of optimization to the previous level.

We now present the API of our OSR implementation[8].

[6] https://developers.google.com/v8/

[7] http://trac.webkit.org/wiki/JavaScriptCore

[8] Available at http://www.sable.mcgill.ca/mclab/mcosr/

3. The OSR API

The key objective of this work was to build a modular system with a clean interface that is easy to use for VM and JIT compiler writers. In this section, we present the API of our OSR module and how JIT compiler developers who are already building JITs/VMs with LLVM can use our module to add OSR functionality to their existing JITs. We provide some concrete examples, based on our McJIT implementation of OSR-based dynamic inlining.

Figure 2(a) represents the structure of a typical JIT developed using LLVM. *LLVM CodeGen* is the front-end that produces LLVM IR for the JIT. The JIT compiler may perform transformations on the IR via the *LLVM Optimizer*. This is typically a collection of transformation and optimization passes that are run on the LLVM IR. The output (i.e., the transformed LLVM IR) from the optimizer is passed to the target code generator, *Target CodeGen*, that produces the appropriate machine code for the code in LLVM IR.

In Figure 2(b), we show a JIT (such as that shown in Figure 2(a)) that has been retrofitted with OSR support components (the shaded components). We describe the functions of *Inserter* and *OSR Pass* shown in Figure 2(b) shortly. In Section 4, we present the implementation of these components and how they interact with the JIT to provide OSR support to the JIT.

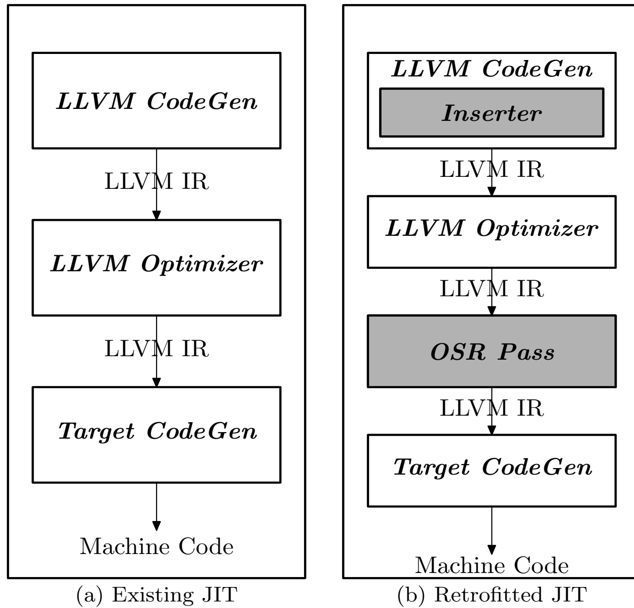

Figure 2. Retrofitting an existing JIT with OSR support

3.1 Adding the OSR Point Inserter

To support OSR, a JIT compiler must be able to mark the program points (henceforth called OSR points) where a running program may trigger OSR. A developer can add this capability to an existing JIT by modifying the compiler to call the *genOSRSignal* function, provided by our API, to insert an OSR point at the beginning of a loop during the LLVM code generation of the loop. The LLVM IR is in SSA form. As will be shown later, an OSR point instruction must be inserted into its own basic block, which must be preceded by the loop header block containing all the ϕ nodes. This ensures that if OSR occurs at runtime, the continuation block can be efficiently determined.

In addition to marking the spot of an OSR point, the JIT compiler writer will want to indicate what transformation should occur if that OSR point triggers at runtime. Thus, the *genOSRSignal* function requires an argument which is a pointer to a *code transformer* function - i.e. the function that will perform the required transformation at runtime when an OSR is triggered. A JIT developer that desires different transformations at different OSR points can simply define multiple code transformers, and then insert OSR points with the desired transformation for each point. A valid transformer is a function pointer of the type *Transformer* that takes two arguments as shown below.

```
typedef unsigned int OSRLabel;
typedef bool
(*Transformer) (llvm::Function*, OSRLabel);
```

The first argument is a pointer to the function to be transformed. The second argument is an unsigned integer representing the label of the OSR point that triggered the current OSR event. The code of the transformer is executed if the executing function triggers an OSR event at a corresponding label. A user may specify a *null* transformer if no transformation is required. As an example of a transformation, our OSR-based dynamic inliner (Section 5.1) uses the transformer shown in Figure 3. It inlines all call sites annotated with label *osrPt*.

```
1 bool  inlineAnnotatedCallSites (llvm::Function *F,
2                                  osr::OSRLabel osrPt) {
3   ...
4   llvm::McJITInliner  inliner (FIM, osrPt, TD);
5    inliner.addFunction ( inlineVersion );
6     inliner.inlineFunctions  ();
7    ...
8 }
```

Figure 3. A Code Transformer

To illustrate with a concrete example of inserting OSR points, our OSR-based dynamic inlining implementation uses the code snippet shown in Figure 4 to insert conditional OSR points after generating the loop header block containing only ϕ nodes. In the code snippet (lines 6 – 12), a new basic block *osr* is created and the call to *genOSRSignal* inserts an OSR point into the block. The rest of the code inserts a conditional branch instruction into *target* and completes the generation of the LLVM IR for the loop.

```
1  ...
2 // get the loop header block --- the target
3 llvm::BasicBlock* target = builder.GetInsertBlock ();
4 llvm::Function* F = target->getParent();
5 // create the osr instruction block
6 llvm::BasicBlock* osrBB =
7   llvm::BasicBlock::Create(F->getContext(), "osr", F);
8 // now create an osr pt and register a transformer
9 llvm:: Instruction * marker =
10     osr::Osr::genOSRSignal(osrBB,
11                   inlineAnnotatedCallSites ,
12                   loopInitializationBB );
13 ...
14 // create the osr condition instruction
15 llvm::Value *osrCond = builder.CreateICmpUGT(counter,
16             getThreshold( context ), "ocond");
17 builder.CreateCondBr(osrCond, osrBB, fallThru );
18  ...
```

Figure 4. Sample Code for Inserting an OSR Point

3.2 Adding the OSR Transformation Pass

After modifying the JIT with the capability to insert OSR points, the next step is to add the creation and running of the OSR transformation pass. When the OSR pass is run on a function with OSR points, the pass automatically instruments the function by adding

the OSR machinery code at all the OSR points (note that the JIT-compiler developer only has to invoke the OSR pass, the pass itself is provided by our OSR module).

The OSR pass is derived from the LLVM function pass. Figure 5 shows a simplified interface of the pass. An LLVM front-end, that is, an LLVM code generator, can use the following code snippet to create and run the OSR pass on a function F after the original LLVM optimizer in Figure 2(b) finishes.

```
llvm::FunctionPass* OIP = osr::createOSRInfoPass();
OIP->runOnFunction(*F);
```

The OSR pass can also be added to an LLVM function pass manager.

```
namespace osr {
  class OSRInfoPass : public llvm::FunctionPass {
  public:
    OSRInfoPass();
    virtual bool runOnFunction(llvm::Function& F);
    virtual const char* getPassName() const
    { return "OSR Info Collection Pass";} ...
  };
  llvm::FunctionPass* createOSRInfoPass();
}
```

Figure 5. The OSR Pass Interface

3.3 Initialization and Finalization

To configure the OSR subsystem during the JIT's start-up time, the JIT developer must add a call to the method *Osr::init*. This method initializes the data structures and registers the functions used later by the OSR subsystem. The JIT developer must also add a call to the method `void Osr:releaseMemory()` to de-allocate the memory allocated by the OSR system. The code snippet in Figure 6 show how an existing JIT can initialize and release the memory used by the OSR subsystem. As shown in line 4, the arguments to `Osr::init` are: a JIT execution engine and the module. The execution engine and the module are used to register the functions used by the system.

```
1  int main(int argc, const char** argv) {
2    ...
3    // initialize the OSR data structures ...
4    Osr:: init (EE, module);
5
6    ... // JIT's Code
7
8    // free up the memory used for OSR ...
9    Osr::releaseMemory();
10   ...
11   return 0;
12 }
```

Figure 6. Initialization and Finalization in the JIT's *main* function

4. Implementation

In the previous section, we outlined our API which provides a simple and modular approach to adding OSR support to LLVM-based JIT compilers. In this section, we present our implementation of the API. Our implementation assumes that the application is single-threaded. We first discuss the main challenges that influenced our implementation decisions, and our solution to those challenges.

4.1 Implementation Challenges

Our first challenge was how to mark OSR points. Ideally, we needed an instruction to represent an OSR point in a function. However, adding a new instruction to LLVM is a non-trivial process and requires rebuilding the entire LLVM system. It will also require users of our OSR module to recompile their existing LLVM installations. Hence, we decided to use the existing call instruction to mark an OSR point. This also gives us some flexibility as the signature of the called function can change without the need to rebuild any LLVM library.

A related challenge was to identify at which program points OSR instructions should be allowed. We decided that the beginning of loop bodies were ideal points because we could ensure that the control flow and phi-nodes in the IR could be correctly patched in a way that does not disrupt other optimization phases in LLVM.

The next issue that we considered was portability. We decided to implement at the LLVM IR, rather than at a lower level, for portability. This is similar to the approach used in Jikes research VM [10], which uses byte-code, rather than machine code to represent the transformed code. This approach also fits well with the extensible LLVM pass manager framework.

A very LLVM-specific challenge was to ensure that the code of the new version is accessible at the old address without recompiling all the callers of the function. Finding a solution to this was really a key point in getting an efficient and local solution.

Finally, when performing an OSR, we need to save the current state (i.e., the set of live values) of an executing function and restore the same state later. Thus, the challenge is how to restore values while at the same time keeping the SSA-form CFG of the function consistent.

We now explain our approach which addresses all these challenges. In particular, we describe the implementation of *Inserter* and *OSR Pass* shown in Figure 2(b).

4.2 OSR Point

In Section 3.1, we explained how a developer can add the capability to insert OSR points to an existing JIT. Here we describe the representation of OSR points.

We represent an OSR point with a call to a native function named *@__osrSignal*. It has the following signature.

```
declare void @__osrSignal(i8*, i64)
```

The first formal parameter is a pointer to some memory location. A corresponding argument is a pointer to the function containing the call instruction. This is used to simplify the integration of inlining; we discuss this in detail in Section 4.5. The second formal parameter is an unsigned integer. A function may have multiple OSR points; the integer uniquely identifies an OSR point.

The OSR module maintains a table named OSR function table (*oft*). The table maps a function in LLVM IR onto a set of OSR-point entries. The set can grow or shrink dynamically as new OSR points are added (e.g., after a dynamic inlining) and old OSR points removed (e.g., after an OSR). An entry e in the set is an ordered pair.

$$e = (osr_call_inst, code_transformer)$$

The first member of the pair — *osr_call_inst* — is the call instruction that marks the position of an OSR point in a basic block. The second is the *code_transformer* (Section 3.1).

4.3 The OSR Pass

The OSR pass in Figure 2(b) is a key component of our OSR implementation. As shown in Figure 5, the OSR transformation pass is derived from the LLVM *FunctionPass* type. Like all LLVM function passes, the OSR pass runs on a function via its *runOnFunction* (Figure 5) method.

The pass first inspects a function's *oft* entry to determine whether the function has at least one OSR point. It returns immediately if the function has no OSR points. Otherwise, it instruments the function

at each OSR point. Figure 7 shows a simplified CFG of a loop with no OSR points. The basic block labelled *LH1* is the loop header. *LB* contains the code for the body of the loop; and the loop exits at *LE*.

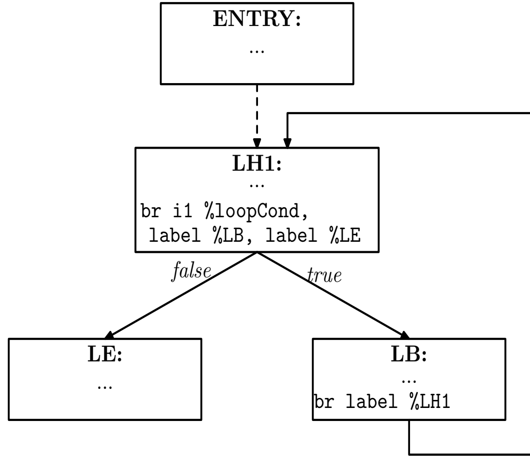

Figure 7. A CFG of a loop with no OSR points.

Figure 8 shows a simplified CFG for the loop in Figure 7 with an OSR point. This represents typical code an LLVM front-end will generate with OSR enabled. Insertion of OSR points is performed by *Inserter* shown in Figure 2(b). The loop header block (now *LH0* in the Figure 8) terminates with a conditional branch instruction that evaluates the Boolean flag *%osrCond* and branches to either the basic block labelled *OSR* or to *LH1*. *LH1* contains the loop termination condition instruction. *LB* contains the code for the body of the loop; the loop exits at *LE*.

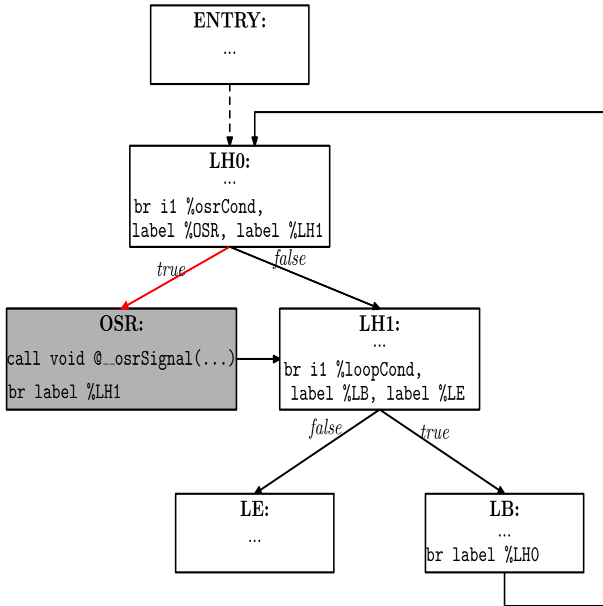

Figure 8. The CFG of the loop in Figure 7 after inserting an OSR point.

The OSR compilation pass performs a liveness analysis on the SSA-form CFG to determine the set of live variables at a loop header such as *LH0* in Figure 8. It creates, using the LLVM cloning support, a copy of the function named the *control version*. As we explain later in this section, this is used to support the transition from one version of the function to another at runtime. It also creates a descriptor [10, 12] for the function. The descriptor contains useful information for reconstructing the state of a function during an OSR event. In our approach, a descriptor is composed of:

- a pointer to the current version of the function;
- a pointer to the control version of the function;
- a map of variables from the original version of the function onto those in the control version; and
- the sets of the live variables collected at all OSR points.

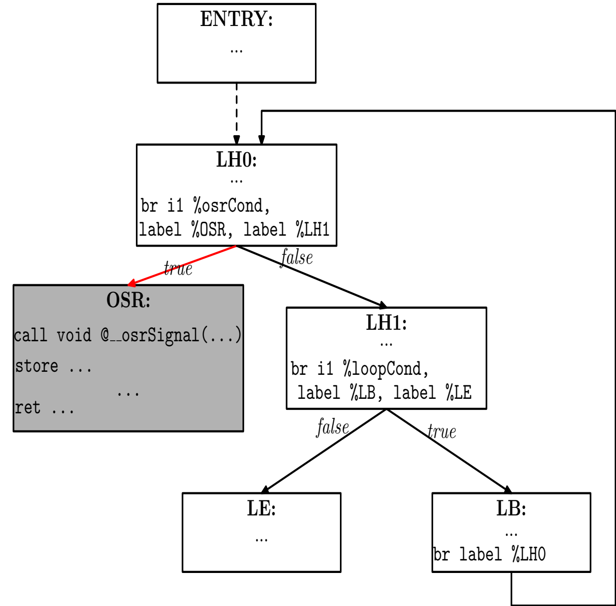

Figure 9. The transformed CFG of the loop in Figure 8 after running the OSR Pass.

After running the OSR pass on the loop shown in Figure 8, the CFG will be transformed into that shown in Figure 9. Notice that in the transformed CFG, the OSR block now contains the code to save the runtime values of the live variables and terminates with a return statement. We now describe in detail the kinds of instrumentation added to an OSR block.

4.3.1 Saving Live Values

To ensure that an executing function remains in a consistent state after a transition from the running version to a new version, we must save the current state of the executing function. This means that we need to determine the live variables at all OSR points where an OSR transition may be triggered. Dead variables are not useful.

As highlighted in Section 3, we require that the header of a loop with an OSR point always terminates with a conditional branch instruction of the form:

```
br i1 %et, label %osr, label %cont
```

This instruction tests whether the function should perform OSR. If the test succeeds (i.e., *%et* is set to `true`), the succeeding block beginning at label *%osr* will be executed and OSR transition will begin. However, if the test fails, execution will continue at the continuation block, *%cont*. This is the normal execution path.

In *%osr* block, we generate instructions for saving the runtime value of each live variable computed by the liveness analysis. The code snippet in Figure 10 shows a typical *osr* block in a simplified form.

```
1  osr:
2    call void @__osrSignal(f, i64 1)
3    store double %7, double* @live
4    store double %8, double* @live1
5    ...
6    store i32 1, i32* @osr_flag
7    call void @__recompile(f, i32 1)
8    call void @f( ... )
9    call void @__recompileOpt(f)
10   ret void
```

Figure 10. OSR Instrumentation

The call to *@__osrSignal(f, i64 1)* in line 2 marks the beginning of the block. Following this call is a sequence of `store` instructions. Each instruction in the sequence saves the runtime value of a live variable into a global variable *@live**. The last `store` instruction stores the value 1 into *@osr_flag*. If *@osr_flag* is non-zero at runtime, then the executing function is performing an OSR transition. We explain the functions of the instructions in lines 7 – 10 later.

The saved variables are mapped onto the variables in the control version. This is a key step as it allows us to correctly restore the state of the executing function during an OSR.

4.4 Restoration of State and Recompilation

The protocol used to signify that a function is transitioning from the executing version to a new version, typically, a more optimized version[9], is to set a global flag. The flag is reset after the transition.

At runtime, the running function executes the code to save its current state. It then calls the compiler to recompile itself and, if a code *transformer* is present, the function is transformed before recompilation. The compiler retrieves the descriptor of the function and updates the running version using the *control* version as illustrated in Figure 11.

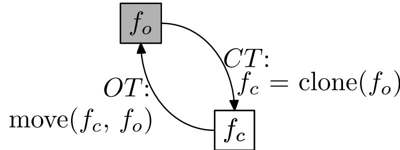

Figure 11. State Management Cycle

Let f_o denote the original version of the LLVM IR of the running function, and f_c denote the control version that was generated by cloning the original version. We denote the set of all the live variables of f_o at the program point p_o with $V_o(p_o)$. Similarly, $V_c(p_c)$ denotes the state of the control version at the matching program point p_c. Because f_c is a copy of f_o, it follows that

$$V_o(p_o) \equiv V_c(p_c).$$

Figure 11 illustrates the state management cycle of the running function. The function starts with version f_o. At compilation time[10] (shown as event CT in Figure 11), we clone f_o to obtain f_c. We then compile f_o. At runtime, when an OSR (event OT in Figure 11) is triggered by the running function, we first remove the instructions in f_o and then *move* the code (LLVM IR) of f_c into f_o, transform/optimize as indicated by the OSR transform, and then recompile f_o and execute the machine code of f_o.

[9] It may also transition from an optimized version to a less optimized version depending on the application.

[10] This includes the original compilation and all subsequent recompilations due to OSR.

This technique ensures that the machine code of the running function is always accessible at the same address. Hence, there is no need to recompile its callers: the machine code of the transformed f_o is immediately available to them at the old entry point of the running function.

To locate the continuation program point p_o ($p_o \equiv p_c$), the compiler recovers the OSR entry of the current OSR identifier; using the variable mappings in the descriptor, finds the instruction that corresponds to the current OSR point. From this, it determines the basic block of the instruction. And because the basic block of an OSR point instruction has one and only one predecessor, the compiler determines the required target, p_o.

4.4.1 Restoration of State

To restore the state of the executing function, we create a new basic block named *prolog* and generate instructions to load all the saved values in this block; we then create another basic block that merges a new variable defined in the *prolog* with that entering the loop via the loop's entry edge. We ensure that a loop header has only two predecessors and because LLVM IR is in SSA form, the new block consists of ϕ nodes with two incoming edges: one from the initial loop's entry edge and the other from *prolog*. The ϕ nodes defined in the merger block are used to update the users of an instruction that corresponds to a saved live variable in the previous version of the function.

Figure 12 shows a typical CFG of a running function before inserting the code for recovering the state of the function. The basic block *LH1* defines a ϕ node for an induction variable ($\%i$ in Figure 12) of a loop in the function. The body of the loop, *LB*, contains a `add` instruction that increments the value of $\%i$ by 1.

Assuming that we are recovering the value of $\%i$ from the global variable $@live_i$, Figure 13 shows the CFG after inserting the blocks for restoring the runtime value of $\%i$. In this figure, *prolog* contains the instruction that will load the runtime value of $\%i$ from the global variable $@live_i$ into $\%_i$; similarly, the basic block *prolog.exit* contains a ϕ instruction ($\%_m_i$) that merges $\%_i$ from *prolog* and the value 1 from *ENTRY*. This variable (i.e., $\%_m_i$) replaces the incoming value (1) from *ENTRY* in the definition of $\%i$ in the loop header (*LH1*) as shown in Figure 13. Notice that the incoming block *ENTRY* has been replaced with *prolog.exit* (*PE*) in the definition of $\%i$ in *LH1*.

Fixing the CFG to keep the SSA form consistent is non-trivial. A simple replacement of a variable with a new variable does not work. Only variables dominated by the definitions in the merger block need to be replaced. New ϕ nodes might be needed at some nodes with multiple incoming edges (i.e., only those that are in the dominance frontier of the merger block). Fortunately, the LLVM framework provides an SSA Updater that can be used to update the SSA-form CFG. We exploited the SSA Updater to fix the CFG.

To complete the state restoration process, we must fix the control flow to ensure that the function continues at the correct program point. For this, we insert a new entry block named *prolog.entry* that loads *@osr_flag* and tests the loaded value for zero to determine, during execution, whether the function is completing an osr transition or its being called following a recent completion of an OSR. The content of the new entry block is shown in the following code snippet.

```
1  prolog.entry :
2    %osrPt = load i32* @osr_flag
3    %cond = icmp eq i32 %osrPt, 0
4    br i1 %cond, label %entry, label %prolog
```

If $\%osrPt$ is non-zero, the test succeeds and the function is completing an OSR; it will branch to $\%prolog$. In $\%prolog$, all the live values will be restored and control will pass to the target block: the loop header where execution will continue. However, if $\%osrPt$

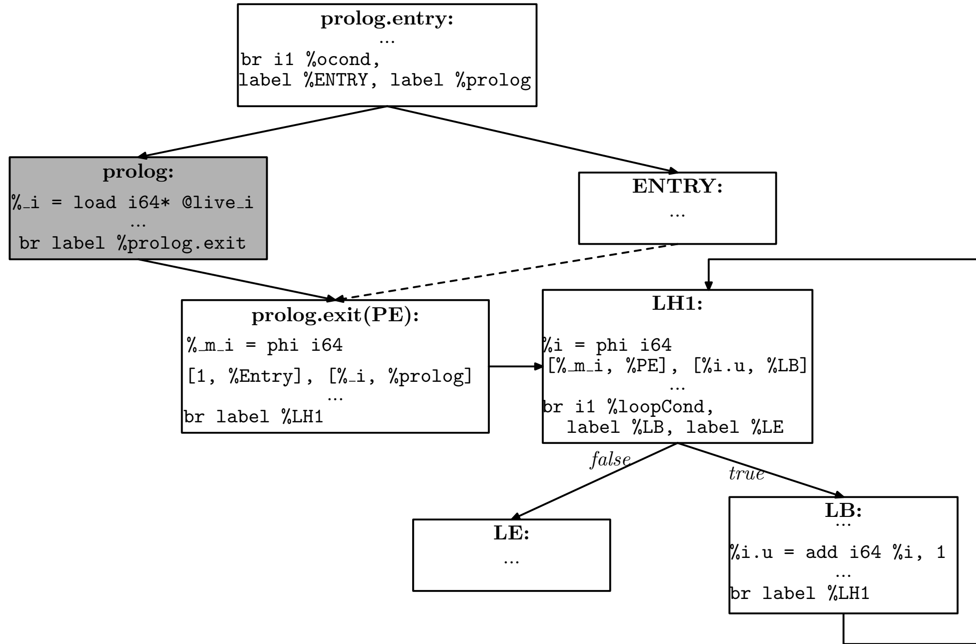

Figure 13. The CFG of the loop represented by Figure 12 after inserting the state recovery blocks.

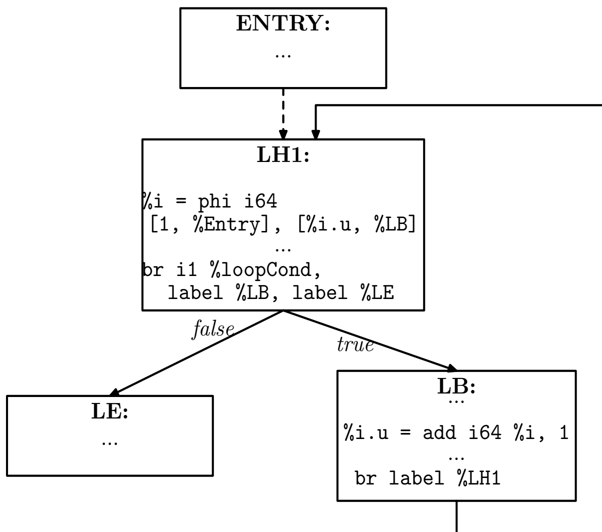

Figure 12. A CFG of a loop of a running function before inserting the blocks for state recovery.

is zero, the function is not currently making a transition: it is being called anew. It will branch to the original entry basic block, where its execution will continue.

As shown in Figure 13, the basic block *prolog.entry* terminates with a conditional branch instruction. The new version of the running function will begin its execution from *prolog.entry*. After executing the block, it will continue at either *prolog* or *ENTRY* (the original entry block of the function) depending on the runtime value of *%cond*.

4.4.2 Recompilation

We now return to the instructions in lines 7 – 10 of Figure 10. The instruction in line 7 calls the compiler to perform OSR and recompile f using the code transformer attached to OSR point 1. After that, function f will call itself (as shown in line 8), but this will execute the machine code generated for its new version. This works because the LLVM recompilation subsystem replaces the instruction at the entry point of function f with a jump to the entry point of the new version. During this call, the function completes OSR and resumes execution. The original call will eventually return to the caller any return value returned by the recursive call.

Normally after an OSR, subsequent calls (if any) of f executes the code in the *prolog.entry*, which tests whether or not the function is currently performing an OSR. However, this test succeeds only during an OSR transition; in other words, the execution of the code in *prolog.entry* after an OSR has been completed is redundant. To optimize away the *prolog.entry*, we again call the compiler (line 9 in Figure 10) but this time, the compiler only removes the *prolog.entry* and consequently, other dead blocks, and recompile f. In Section 5.2, we compare the performance of our benchmarks when the *prolog.entry* is eliminated with the performance of the same benchmarks when the *prolog.entry* is not eliminated.

4.5 Inlining Support

Earlier, we discussed the implementation of OSR points and how the OSR transformation pass handles OSR points. However, we did not specify how we handled OSR points inserted into a function from an inlined call site. A seamless integration of inlining optimization poses further challenges. When an OSR event is triggered at runtime, the runtime system must retrieve the code transformer attached to the OSR point from the *oft* entry of the running function. How then does the system know the original function that defined an inlined OSR point? Here we explain how our approach handles inlining.

Remember that an OSR point instruction is a call to a function. The first argument is a pointer to the enclosing function. There-

fore, when an OSR point is inlined from another function, the first argument to the inlined OSR point (i.e., a call instruction) is a function pointer to the inlined function. From this, we can recover the *transformer* associated with this point by inspecting *oft* using this pointer. We can then modify these OSR points by changing the first argument into a pointer to the current function and assign a new ID to each inlined OSR point. We must also update the *oft* entry of the caller to reflect these changes.

We distinguish two inlining strategies: static and dynamic. In static inlining, a call site is expanded before executing the *caller*. This expansion may introduce a new OSR point from the *callee* into the caller and invalidates all the state information collected for the existing OSR points. We regenerate this information after any inlining process.

Dynamic inlining concerns inlining of call sites in a running function during the execution of the function after observing, for some time, its runtime behaviour. Typically, we profile a program to determine *hot* call sites and inline those subject to some heuristics. We used OSR support to implement dynamic inlining of call sites in long-running loops. We discuss this implementation next.

5. Case Study: Dynamic Inlining

In this section, we present an example application of our OSR approach to support selective dynamic inlining in McJIT. We selected this as our first application of OSR because inlining impacts OSR since it must properly deal with OSR points in the inlined functions. Moreover, inlining can increase the opportunity for loop vectorization and provide larger scopes for subsequent optimizations.

5.1 The McJIT dynamic inliner

In our approach to dynamic inlining, we first modified McJIT to identify potential inlining candidates. In our case, a call is considered an inlining candidate if the body of the called function is less than 20 basic blocks, or it is less than 50 basic blocks and it has an interpreter environment associated with the body. McJIT generates LLVM IR for each function in a program. The LLVM IR generated by McJIT may contain calls to the interpreter for special cases and for those cases the symbol environment set-up code facilitates the interaction with the interpreter. In our case, inlining can reduce the interpreter environment overheads.

We then modified McJIT so that loops which contain potential inlining candidates are instrumented with with a hotness counter and a conditional which contains an OSR point (where the OSR point is associated with a new McJIT inlining transformer). When an OSR triggers (i.e. the hotness counter reaches a threshold), the McJIT inlining transformation will inline all potential inlining candidates associated with that OSR point.

There are many strategies for determining which loops should be given an OSR point, and a JIT developer can define any strategy that is suitable for his/her situation. For McJIT, we defined two such general strategies, as follows:

CLOSEST Strategy: The LLVM front-end is expected to insert OSR points only in the loop that is closest to the region that is being considered for optimization. For example, to implement a dynamic inlining optimization using this strategy, an OSR point is inserted at the beginning of the closest loop enclosing an interesting call site. This strategy is useful for triggering an OSR as early as possible, i.e., as soon as that closest enclosing loop becomes hot.

OUTER Strategy: The LLVM front-end is expected to insert an OSR point at the beginning of the body of the outer-most loop of a loop nest containing the region of interest. This approach is particularly useful for triggering many optimizations in a loop nest with a single OSR event. In the case of dynamic inlining, one OSR will trigger inlining of all inlining candidates within the loop nest. The potential drawback of this strategy is that the OSR will not trigger until the outermost loop becomes hot, thus potentially delaying an optimization.

In Figure 14, we illustrate the difference between the two strategies using an hypothetical loop nest. We use a call site to represent an interesting region for optimization.

A loop is represented with a box. The box labelled L_0 denotes the outer-most loop of the loop nest. The nest contains four loops and has a depth of 3. Loops L_1 and L_3 are at the same nesting level. And L_2 is nested inside L_1. The loop nest has three call sites: C_0 in loop L_0, C_2 in loop L_2, and C_3 in loop L_3. Figure 14(a) shows the loop nest with no OSR points.

With the outer-most-loops strategy, an OSR point will be inserted only at the beginning of the outer-most loop, L_0 as shown in Figure 14(b). However, if the strategy is closest-enclosing loops, the front-end will insert an OSR point at the beginning of loops L_0, L_2, and L_3 as shown in Figure 14(c). Although C_2 is inside L_1, no OSR points are inserted into L_1 because L_1 is not the closest-enclosing loop of C_2.

As shown in the figure, the outer-most-loops strategy causes only one OSR point to be inserted into the entire loop nest, while the closest-enclosing-loops strategy causes three OSR points to be inserted. Thus, depending on the optimization performed during an OSR event, the choice of strategy can make a difference in performance.

In our VM, a user specifies an OSR strategy from the command line when invoking the VM, like the following example.

```
./mcvm -jit_enable true -jit_osr_enable true
       -jit_osr_strategy outer.
```

This command starts McVM with OSR enabled with *outer* strategy. In our JIT, the default strategy is *outer*.

When the OSR triggers it calls the McJIT inliner transformation. Our McJIT inliner calls the LLVM basic-inliner library to do the actual inlining. However, the McJIT inliner must also do some extra work because it must inline the correct version of *callee* function body. The key point is that if the *callee* has an OSR point, it must not inline the version of the callee which has already been instrumented with the code to store values of the live variables at this OSR point. If this version is inlined into the *caller* — the function that is performing OSR— the instrumentation becomes invalid as the code does not correctly save the state of the caller at that inlined OSR point. We resolved this problem by recovering the *control* version of the called function (*callee*) and modifying the call site. We change the function called by the call instruction to the control version of the callee. For instance, if the inlined call site is `call void @f(...)`, and the control version of f is f', then the call site will be changed to `call void @f'(...)`. Note that the control version has an identical OSR point but is not instrumented to save the runtime values of live variables at that program point. For consistency, the function descriptor of the function is updated after inlining as outlined earlier.

5.2 Experimental Evaluation

We used our McJIT dynamic inliner to study the overheads of OSR and the potential performance benefit of inlining. We used a collection of MATLAB benchmarks from a previous MATLAB research project and other sources [9, 19, 20], Table 1 gives a short description of each benchmark. All the benchmarks have one or more loops, the table also lists the total number of loops and max loop depth for each benchmark.

The configuration of the computer used for the experimental work is:

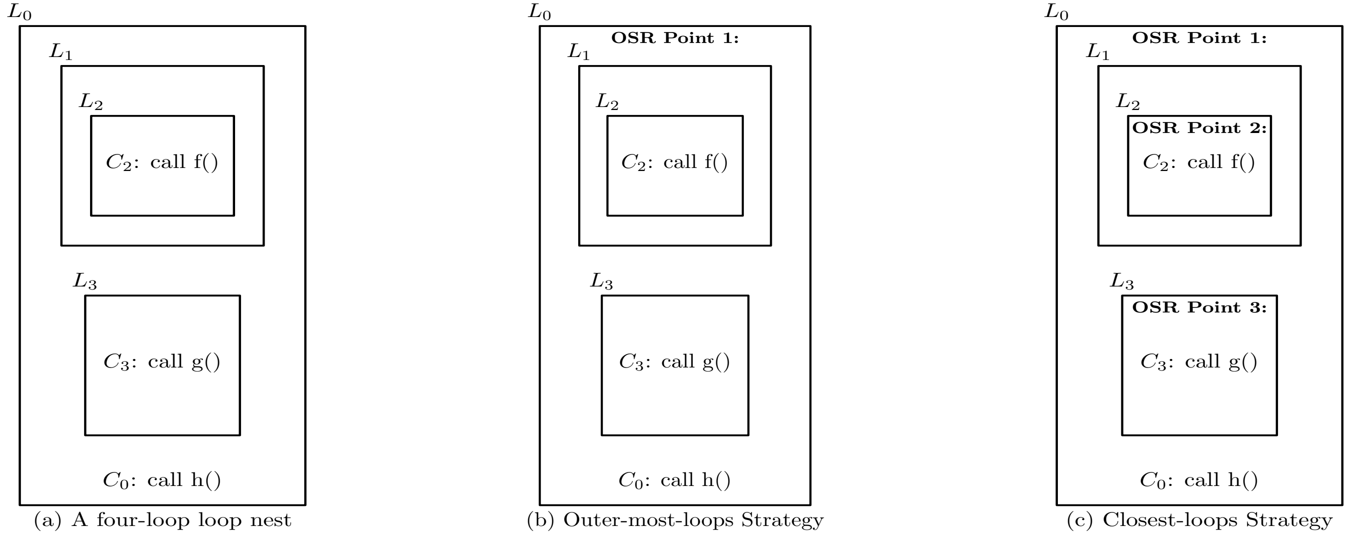

Figure 14. A loop nest showing the placement of OSR points using the closest or outer-most Strategy

BM	Description	# Loops	Max Depth
adpt	adaptive quadrature using Simpsons rule	4	2
capr	capacitance of a transmission line using finite difference and and Gauss-Seidel iteration.	10	2
clos	transitive closure of a directed graph	4	2
crni	Crank-Nicholson solution to the one dimensional heat equation	7	2
dich	Dirichlet solution to Laplaces equation	6	3
diff	Youngs two-slit diffraction experiment	13	4
edit	computes the edit distance of two strings	7	2
fdtd	3D FDTD of a hexahedral cavity with conducting walls	1	1
fft	fast fourier transform	6	3
fiff	finite-difference solution to the wave equation	13	4
mbrt	Mandelbrot set	3	2
nb1d	N-body problem coded using 1d arrays for the displacement vectors	6	2
nfrc	computes a Newton fractal in the complex plane -2..2,-2i..2i	3	2
nnet	neural network learning AND/OR/XOR functions	11	3
schr	solves 2-D Schroedinger equation	1	1
sim	Minimizes a function with simulated annealing	2	2

Table 1. The Benchmarks

Processor: Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz
RAM: 16 GB;
Cache Memory: L1 32KB, L2 256KB, L3 12MB;
Operating System: Ubuntu 12.04 x86-64;
LLVM: version 3.0; and McJIT: version 1.0.

Our main objectives were:

- To measure the overhead of OSR events on the benchmarks over the outer-most and closest-loop strategies. The overhead includes the cost of instrumentation and performing OSR transitions. We return to this in Section 5.2.1.
- To measure the impact of selective inlining on the benchmarks. We discuss this in detail in Section 5.2.2.

We show the results of our experiments in Table 2(a) and Table 2(b). For these experiments, we collected the execution times (shown as **t(s)** in the tables) measured in seconds, for 7 runs of each benchmark. To increase the reliability of our data, we discarded the highest and the lowest values and computed the average of the remaining 5 values. To measure the variation in the execution times, we computed the standard deviation (STD) (shown as **std**) of the 5 values for each benchmark under 3 different categories. All the results shown in both tables were collected using the outer-most-loops strategy, with the default LLVM code-generation optimization level.

The column labelled **Normal** gives the average execution times and the corresponding STDs of the benchmarks ran with OSR disabled, while the column labelled **With OSR** gives similar data when OSR was enabled. Column **With OSR** in Table 2(b) shows the results obtained when dynamic inlining plus some optimizations enabled by inlining were on.

	Normal(N)		With OSR(O)		#OSR		Ratio
BM	t(s)	std	t(s)	std	I	T	O/N
adpt	17.94	0.06	17.84	0.08	1	1	0.99
capr	11.61	0.01	11.63	0.02	2	2	1.00
clos	16.96	0.01	16.96	0.01	0	0	1.00
crni	7.20	0.04	7.40	0.04	1	1	1.03
dich	13.92	0.01	13.92	0.00	0	0	1.00
diff	12.73	0.07	12.80	0.09	0	0	1.01
edit	6.58	0.03	6.66	0.09	1	0	1.01
fdtd	12.14	0.03	12.16	0.05	0	0	1.00
fft	13.95	0.05	14.05	0.03	1	1	1.01
fiff	8.02	0.01	8.05	0.01	1	1	1.00
mbrt	9.05	0.11	9.22	0.11	1	1	1.02
nb1d	3.44	0.02	3.47	0.01	0	0	1.01
nfrc	9.68	0.05	10.00	0.04	2	2	1.03
nnet	5.41	0.02	5.59	0.03	2	1	1.03
schr	11.40	0.01	11.42	0.03	0	0	1.00
sim	15.26	0.03	15.92	0.07	1	1	1.04
GM							1.01

(a) OSR Overhead

	Normal(N)		With OSR(O)		#OSR				Ratio
BM	t(s)	std	t(s)	std	I	T	FI	CA	O/N
adpt	17.94	0.06	17.85	0.06	1	1	1	F	0.99
capr	11.61	0.01	11.69	0.02	2	2	2	T	1.01
clos	16.96	0.01	17.18	0.22	0	0	0	F	1.01
crni	7.2	0.04	6.73	0.24	1	1	1	T	**0.93**
dich	13.92	0.01	13.94	0.01	0	0	0	F	1.00
diff	12.73	0.07	12.74	0.04	0	0	0	F	1.00
edit	6.58	0.03	6.66	0.07	1	0	0	F	1.01
fdtd	12.14	0.03	12.13	0.03	0	0	0	F	1.00
fft	13.95	0.05	13.91	0.02	1	1	2	F	1.00
fiff	8.02	0.01	8.26	0.03	1	1	1	F	1.03
mbrt	9.05	0.11	9.06	0.03	1	1	1	F	1.00
nb1d	3.44	0.02	3.47	0.01	0	0	0	F	1.01
nfrc	9.68	0.05	4.26	0.02	2	2	5	T	**0.44**
nnet	5.41	0.02	5.71	0.03	2	1	1	F	1.05
schr	11.4	0.01	11.45	0.05	0	0	0	F	1.00
sim	15.26	0.03	14.72	0.09	1	1	1	F	**0.96**
GM									0.95

(b) Dynamic Inlining using OSR

Table 2. Experimental Results (lower execution ratio is better)

The number of OSR points instrumented at JIT compilation time is shown under **I** of the column labelled **#OSR**; while the number of OSR events triggered at runtime is shown under the column labelled **T** of **#OSR**. The execution ratio for a benchmark is shown as the ratio of the average execution time when OSR was enabled to the average execution time when OSR was disabled (this is the default case). Columns **O/N** of Table 2(a) and **O/N** of Table 2(b) show, respectively, the ratio for each benchmark when OSR only was enabled and when OSR and inlining were enabled. The last row of Table 2(a) and Table 2(b) shows the average execution ratio (the geometric mean (GM)) over all the benchmarks. In Table 2(b), we show the number of functions inlined under **FI**. The column labelled **CA** indicates whether at least one function in the benchmark is called again after it has completed an OSR event.

The STDs of our data sets range from 0.00 to 0.24, showing that the execution times are quite reliable. We now discuss the results of our experiments in detail.

5.2.1 Cost of Code Instrumentation and OSR

Because our approach is based on code instrumentation, we wanted to measure the overhead of code instrumentation and triggering OSRs. This will allow us to assess the performance and develop an effective instrumentation strategy.

Column **O/N** of Table 2(a) shows that the overheads range from about 0 to 4%; this is also the range for the closest-enclosing-loops strategy, suggesting that the overheads under the two strategies are close. Out of the 16 benchmarks, 10 have at least one OSR point; and 8 of these 10 benchmarks triggered one or more OSR events. We have not shown the table of the results for the closest-enclosing loops because out of the 8 benchmarks that triggered an OSR event, the outer-most and the closest-enclosing loops are different only in 3 benchmarks: *mbrt*, *nfrc*, and *sim*. The execution ratios for these benchmarks under the closest-enclosing-loops strategy are: 1.00 for *mbrt*, 1.02 for *nfrc*, and 1.04 for *sim*. The *mbrt* and *nfrc* benchmarks have lower execution ratios under the closest-enclosing-loops strategy. It is not entirely clear whether the closest-enclosing-loops strategy is more effective than the outer-most-loops strategy; although, with these results, it appears that using the closest-loops strategy results in lower overheads. The choice between these two will depend largely on the kinds of the optimizing transformations expected at OSR points. We return to this discussion in Section 5.2.2, where we examine the effectiveness of our dynamic inlining optimization.

We investigated the space performance and found that, depending on the strategy, the three benchmarks (*mbrt*, *nfrc* and *sim*) compiled up to 3% more instructions under the closest-enclosing-loops strategy. This is hardly surprising; the OSR overhead depends on the number of OSR points instrumented and the number of OSR points triggered at runtime. The size of the instrumentation code added at an OSR point in a function depends on the size of the live variables of the function at that point, and this varies depending on the position of the OSR point in a loop nest. The outer-most loop is likely to have the smallest set of live variables.

Although the overhead peaked at 4%, the average overhead over all the benchmarks (shown as GM in Table 2(a)) is 1%. Thus, we conclude that on average, the overhead is reasonable and practical for computation-intensive applications. As we continue to develop effective optimizations for MATLAB programs, we will work on techniques to use OSR points in locations where subsequent optimizations are likely to offset this cost and therefore increase performance.

5.2.2 Effectiveness of Selective Inlining With OSR

Our objective here is to show that our approach can be used to support dynamic optimization. So, we measured the execution times of the benchmarks when dynamic inlining is enabled. When an OSR is triggered, we inline call sites in the corresponding loop nest. Column **With OSR** of Table 2(b) shows the results of this experiment.

The results show significant improvements for *crni*, *nfrc* and *sim*. This shows that our dynamic inlining is particularly effective for this class of programs. Further investigation revealed that these benchmarks inlined multiple small functions and several of these functions fall back to the McVM's interpreter to compute some complicated expressions. McJIT's interactions with the interpreter is facilitated by setting up a symbol environment for binding variables at runtime. Our dynamic inlining enables optimization that eliminates the environment set-up instructions in the inlined code. This is the main cause of performance improvement in *nfrc* and *sim*, and is impossible to do without inlining.

Only the *fiff* and *nnet* show a real decrease in performance when using the outer-most-loop strategy with inlining. We found that the function inlined by *nnet* contains some expensive cell array operations, which our optimizer is currently unable to handle. The

benchmark also triggered OSR event once, but performed three OSR instrumentation phases: two at the compilation time and one re-instrumentation during the only OSR event.

We wanted to assess the impact of recompilation to optimize the *prolog.entry* block added during an OSR event; so we turned off recompilation after OSR and re-collected the execution times for the benchmarks. Out of the 9 benchmarks that performed inlining, only 3 benchmarks contain at least a further call to a function that completed an OSR. These are the rows with the value "**T**" against the column labelled **CA** in Table 2(b). The results for these benchmarks under the no-recompilation after OSR is: 1.01 for *capr*, 0.95 for *crni*, and 0.45 for *nfrc*. These results suggest that the recompilation to remove the *prolog.entry* contributes to the increase in performance for *capr* and *nfrc*. The basic block has the potential to disrupt LLVM optimizations and removing it might lead to better performance. The recompilation after OSR does not result in a slowdown for the other benchmarks.

In Section 5.2.1, we mentioned that the kinds of the optimizing transformations can guide the choice of strategy that lead to better performance. Considering the 3 benchmarks with a loop nest where the outer-most and closest-enclosing loops are different, that is, *mbrt*, *nfrc* and *sim*, we found that the outer-most-loop strategy outperforms the closest-enclosing-loop strategy. In particular, the *sim* benchmark results in about 5% performance degradation. These results support our claim.

We recorded the average performance improvement over all the benchmarks (shown as GM in Table 2(b)) of 5%. We conclude that our OSR approach is effective, in that efficiently supports this optimization, and that it works smoothly with inlining. To see further benefits of OSR for MATLAB, we shall develop more sophisticated optimizations that leverage the on-the-fly dynamic type and shape information that is very beneficial for generating better code.

6. Related Work

Hölzle et al [12] used an OSR technique to dynamically deoptimize running optimized code to debug the executing program. OSR techniques have been in used in several implementations of the Java programming language, including Jikes research VM [3, 10] and HotSpot [18] to support adaptive recompilation of running programs. A more general-purpose approach to OSR for the Jikes VM was suggested by Soman and Krintz [21] which decouples OSR from the program code. Our approach is more similar to the original Jikes approach in that we also implement OSR points via explicit instrumentation and OSR points in the code. However, we have designed our OSR points and OSR triggering mechanism to fit naturally into the SSA-form LLVM IR and tool set. Moreover, the LLVM IR is entirely different from Java byte-code and presents new challenges to OSR implementation at the IR level (Section 4). Our approach is also general-purpose in the sense that the OSR can potentially trigger any optimization or de-optimization that can be expressed as an LLVM transform.

Recently, Süsskraut et al [23] developed a tool in LLVM for making a transition from a slow version of a running function to a fast version. Like Süsskraut et al, our system is based on LLVM. However, there are significant differences in the approaches. While their system creates two versions of the same function statically, and transitions from one version to another at runtime, our proposed solution instruments and recompiles code dynamically at runtime. This is more suitable for an adaptive JIT. Secondly, the approach used by Süsskraut et al stores the values of local variables in a specially allocated area that is always accessible when an old stack frame is destroyed and a new stack frame is created for the executing function. This requires a special memory management facility beyond that provided by LLVM. In contrast to their approach, our approach does not require a special allocation because the stack frame is not destroyed until OSR transition is completed. The recursive call of the executing function essentially extends the old stack frame. We only have to copy the old addresses and scalar values from the old stack frame onto the new stack frame. Finally, another notable difference between our approach and that taken by Süsskraut et al is that their approach requires instrumenting the caller to support OSR in a called function. This may result in high instrumentation overhead. In our approach, we do not instrument a caller to support OSR in a callee.

Inlining is an important compiler optimization. It has been used successfully in many production compilers, especially compilers for object-oriented programming languages. Several techniques for effective inlining were introduced in the several implementations of SELF [6, 13]. SELF-93 [13] uses heuristics to determine the root method for recompilation by traversing the call stack. It then in-lines the traversed call stack into the root method. The HotSpot Server VM [18] uses a similar inlining strategy.

Online profile-directed inlining has been explored in many VMs [2, 4, 5, 8, 11, 22]. The Jikes research VM [3] considers the effect of inlining in its cost-benefit model for recompilation by raising the expected benefit of recompiling a method with a frequently executed call site. Suganuma et al report that for inlining decisions for non-tiny methods, heuristics based solely on online profile data outperforms those based on offline, static data [22]. Online profile-directed inlining in a MATLAB compiler has not been reported in the literature. We expect that by using online profiling information to identify hot call sites and guide inlining decisions, inlining of the most critical call sites will boost performance.

7. Conclusions and Future Work

In this paper, we have introduced a modular approach to implementing OSR for LLVM-based JIT compilers, and demonstrated the approach by implementing selective dynamic inlining for MATLAB. Our approach should be very easy for others to adopt because it is based on the LLVM and is implemented as an LLVM pass. Furthermore, we found a solution which does not require any special data structures for storing stack frame values, nor any instrumentation in the callers of functions containing OSR points. It also does not introduce any changes to LLVM which would require rebuilding the LLVM system. Finally, our approach also provides a solution for the case where a function body containing OSR points is inlined, in a way that maintains the OSR points and adapts them to the inlined context.

We used our OSR strategy in the McJIT implementation, and using this implementation, we demonstrated the feasibility of the approach by measuring the overheads of the OSR instrumentation for two OSR placement strategies: outer-most loops and closest-enclosing loops. On our benchmark set, we found overheads of 0 to 4%. Further, we used the OSR machinery to implement dynamic incremental function inlining. On our benchmarks, we found some performance improvements and slight degradations, with several benchmarks showing good performance improvements.

Our ultimate goal is to use OSR to handle recompilation of key loops, taking advantage of type knowledge to apply more sophisticated loop optimizations, including parallelizing optimizations which can leverage GPU and multicores. Thus, as McJIT and MATLAB-specific optimizations develop, we plan to use OSR to enable such optimizations. In addition to our own future uses of our OSR implementation, we also hope that other groups will also use our OSR approach in LLVM-based JITs for other languages, and we look forward to seeing their results.

Acknowledgments

This work was supported in part by NSERC and FQRNT. We thank all the anonymous reviewers for their helpful comments and sug-

gestions on this version, and previous versions, of this paper. We particularly would like to thank the VEE reviewer who suggested that we include a categorization for the kinds of OSR transitions.

References

[1] LLVM. http://www.llvm.org/.

[2] A. Adl-Tabatabai, J. Bharadwaj, D. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. StarJIT: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7(1):19–31, Feb 2003.

[3] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Syst. J.*, 44(2):399–417, 2005.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapenó JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 47–65, New York, USA, 2000. ACM.

[5] M. Arnold, M. Hind, and B. G. Ryder. Online Feedback-Directed Optimization of Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 111–129, New York, USA, 2002. ACM.

[6] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 1–15, New York, USA, 1991. ACM.

[7] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. In *International Conference on Compiler Construction*, pages 46–65, March 2010.

[8] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 13–26, New York, USA, 2000. ACM.

[9] Cleve Moler. *Numerical Computing with MATLAB*. SIAM, 2004.

[10] S. J. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society.

[11] K. Hazelwood and D. Grove. Adaptive Online Context-Sensitive Inlining. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, pages 253–264, Washington, DC, USA, 2003. IEEE Computer Society.

[12] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[13] U. Hölzle and D. Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications*, OOPSLA '94, pages 229–243, New York, NY, USA, 1994. ACM.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society.

[15] MathWorks. *MATLAB Programming Fundamentals*. The MathWorks, Inc., 2009.

[16] McLAB. The McVM Virtual Machine and its JIT Compiler, 2012. http://www.sable.mcgill.ca/mclab/mcvm_mcjit.html.

[17] C. Moler. The Growth of MATLAB™ and The MathWorks over Two Decades, 2006. http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf.

[18] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

[19] Press, H. William and Teukolsky, A. Saul and Vetterling, T. William and Flannery, P. Brian. *Numerical Recipes : the Art of Scientific Computing*. Cambridge University Press, 1986.

[20] L. D. Rose, K. Gallivan, E. Gallopoulos, B. A. Marsolf, and D. A. Padua. FALCON: A MATLAB Interactive Restructuring Compiler. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 269–288, London, UK, 1996. Springer-Verlag.

[21] S. Soman and C. Krintz. Efficient and General On-Stack Replacement for Aggressive Program Specialization. In *Software Engineering Research and Practice*, pages 925–932, 2006.

[22] T. Suganuma, T. Yasue, and T. Nakatani. An Empirical Study of Method In-lining for a Java Just-In-Time Compiler. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 91–104, Berkeley, CA, USA, 2002. USENIX Association.

[23] M. Süsskraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer. Prospect: A Compiler Framework for Speculative Parallelization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 131–140, New York, USA, 2010. ACM.